



Бесплатная электронная книга

УЧУСЬ

C++

Free unaffiliated eBook created from
Stack Overflow contributors.

#C++

.....	1
1: C ++	2
.....	2
.....	2
Examples	2
.....	2
.....	3
.....	4
.....	4
C-Style / Block	5
.....	6
.....	6
.....	6
.....	7
.....	8
.....	8
.....	8
.....	8
.....	9
.....	9
.....	10
C ++	11
.....	12
2: C	14
.....	14
Examples	14
.....	14
.....	14
.....	14
3: Const Correctness	15
.....	15
.....	

Examples.....	15
.....	15
Const Correct Class Design.....	16
Const	19
Const Correctness	20
const CV-Qualified Member Functions:.....	20
const :.....	22
4: constexpr.....	25
.....	25
.....	25
Examples.....	25
constexpr.....	25
constexpr.....	27
if.....	29
5: decltype.....	31
.....	31
Examples.....	31
.....	31
.....	31
6: Loops.....	33
.....	33
.....	33
.....	33
Examples.....	33
.....	33
.....	36
.....	39
.....	39
Do-while.....	40
Loop: Break and Continue.....	41
-	42

7: Perfect Forwarding	45
.....	45
Examples	45
.....	45
8: RAII:	47
.....	47
Examples	47
.....	48
/ ScopeExit	49
ScopeSuccess (c ++ 17)	49
ScopeFail (c ++ 17)	51
9: RTTI:	53
Examples	53
.....	53
dynamic_cast	53
typeid	54
, c ++	54
10: SFINAE ()	55
Examples	55
enable_if	55
.....	55
void_t	57
trailing decltype	59
SFINAE	59
enable_if_all / enable_if_any	60
is_detected	62
.....	63
11: static_assert	66
.....	66
.....	66
.....	66

Examples.....	66
static_assert.....	66
12: std :: function: , ...	68
Examples.....	68
.....	68
std :: function, std :: bind.....	68
std :: function lambda std :: bind.....	69
«»	70
std :: function	71
std :: tuple.....	73
13: std :: set std :: multiset.....	75
.....	75
.....	75
Examples.....	75
.....	75
.....	76
.....	77
.....	78
.....	78
~.....	79
.....	79
.....	80
.....	80
14:	83
.....	83
Examples.....	83
.....	83
.....	84
auto, const	84
.....	85
(C ++ 14).....	85
~.....	86

15:		88
Examples.....		88
-		88
16:		90
.....		90
Examples.....		90
O (log n).....		90
17:		92
.....		92
.....		92
Examples.....		92
.....		92
18:		94
.....		94
Examples.....		94
[[]]		94
[[]]		95
[[]] [[(«»)]]		96
[[Nodiscard]]		97
[[Maybe_unused]]		97
19: / c ++		99
.....		99
Examples.....		99
.....		99
20:		101
Examples.....		101
.....		101
.....		101
.....		102
.....		102
21: -		103
.....		

Examples.....	103
.....	103
C-	103
std :: bitset.....	103
.....	103
C- -	103
std :: bitset.....	104
.....	104
C- -	104
std :: bitset.....	104
.....	104
C- -	104
std :: bitset.....	105
n- x.....	105
C- -	105
std :: bitset.....	105
.....	105
C- -	105
std :: bitset.....	105
.....	106
C- -	106
.....	106
, 2.....	107
- :	107
22: C ++.....	109
.....	109
Examples.....	109
.....	109
23:	110
.....	

.....110

Examples.....110

 C ++ 11 110

 ~.....111

 112

 113

 114

24: 117

.....117

Examples.....117

 117

 std :: tuple.....118

 std :: array.....119

 std :: pair.....119

 120

 121

 Object Consumer.....123

 std :: vector.....123

 124

25: 125

.....125

Examples.....125

 125

26: 126

.....126

.....126

.....126

Inline 126

.....127

.....127

Examples.....127

-	127
.....	127
.....	128
?.....	128
27:	129
.....	129
.....	129
Examples.....	129
:	129
Lambda auto.....	129
.....	130
28:	131
.....	131
.....	131
Examples.....	131
.....	131
()	132
29:	134
.....	134
Examples.....	134
.....	134
.....	134
.....	135
30:	138
.....	138
Examples.....	138
.....	138
.....	139
.....	140
31:	141
Examples.....	141
.....	141

.....	141
32:	143
Examples.....	143
.....	143
.....	145
33: Pimpl.....	147
.....	147
Examples.....	147
Pimpl.....	147
34: C ++.....	149
.....	149
Examples.....	149
C ++.....	149
35:	151
Examples.....	151
.....	151
Rethrow ().....	152
Try Blocks	153
Try Block	153
Try Blocks	154
: , const.....	154
.....	156
:: uncaught_exceptions.....	158
.....	159
36: std :: unordered_map.....	162
.....	162
.....	162
Examples.....	162
.....	162
.....	162
37:	164
.....	164

.....	164
.....	164
Examples.....	164
.....	164
.....	164
.....	165
38:	167
Examples.....	167
C ().....	167
.....	167
.....	168
-	168
.....	168
.....	170
.....	170
.....	171
.....	171
.....	172
.....	173
.....	174
.....	174
.....	175
39:	177
Examples.....	177
.....	177
.....	177
.....	177
.....	178
.....	178
40:	179

Examples.....	179
.....	179
prvalue.....	180
xvalue.....	180
.....	180
glvalue.....	181
Rvalue.....	181
41: /	183
.....	183
.....	183
Examples.....	183
.....	183
.....	184
.....	185
.....	187
.....	189
.....	190
.....	191
:	192
.....	193
.....	193
/	194
.....	199
.....	203
.....	208
struct / class.....	210
42: const.....	212
.....	212
.....	212
Examples.....	212
.....	212
.....	213

-.....	213
const non-const getter.....	214
43: Friend.....	216
.....	216
Examples.....	216
.....	216
Friend.....	217
.....	217
44: mutable.....	219
Examples.....	219
.....	219
.....	219
45:	221
.....	221
.....	221
.....	221
Examples.....	223
.....	223
.....	224
noexcept.....	225
.....	226
.....	227
.....	227
46:	233
Examples.....	233
Const.....	233
decltype.....	233
.....	234
.....	234
.....	235
47:	236
.....	236

Examples.....	236
1. , , ...	236
2. , ...	237
3. ()......	238
48:	240
.....	240
.....	240
Examples.....	240
GCC.....	240
:	242
Visual C ++ ().....	242
Visual Studio () - Hello World.....	247
Clang.....	254
.....	255
C ++.....	256
:: ().....	259
49:	265
.....	265
Examples.....	265
-.....	265
50: C ++.....	267
.....	267
Examples.....	267
- C ++.....	267
51: Elision.....	270
Examples.....	270
.....	270
.....	271
elision.....	273
elision.....	273
elision.....	274

.....	274
52:	276
.....	276
.....	276
.....	276
Examples.....	276
.....	277
.....	277
Vs Assignment Constructor	278
53:	280
.....	280
Examples.....	280
.....	280
.....	280
nullptr.....	280
.....	281
.....	282
54: (CRTP).....	284
.....	284
Examples.....	284
(CRTP).....	284
CRTP, ...	286
55:	288
.....	288
.....	288
.....	289
Examples.....	289
-?.....	289
.....	292
.....	293
.....	294
.....	295

.....	296
.....	297
.....	298
.....	299
- C ++ 03	300
.....	302
std::function.....	302
:.....	302
Y-.....	302
lambdas	303
56:	306
.....	306
Examples.....	306
.....	306
.....	309
.....	309
.....	309
.....	309
.....	309
.....	310
57:	311
.....	311
Examples.....	311
: safe	311
.....	312
std :: vector.....	313
(. . 2D-).....	314
, std :: vector	315
.....	317
58:	319
.....	319
.....	319
Examples.....	319

.....	319
.....	322
std :: integer_sequence.....	323
.....	324
, ...	325
C ++ 11 ().....	326
T.....	327
--.....	328
Generic Min / Max count.....	329
59:	330
.....	330
Examples.....	330
.....	330
Goto Cleanup.....	333
60:	334
.....	334
.....	334
.....	334
Examples.....	334
.....	334
.....	335
std :: thread.....	335
.....	337
std :: async std :: thread.....	339
.....	339
.....	339
.....	340
.....	340
.....	341
.....	342
.....	343
.....	345

61: C ++ 11	347
.....	347
.....	347
.....	348
.....	348
-.....	349
-.....	349
.....	349
Examples.....	350
.....	350
.....	352
62:	353
.....	353
std :: shared_mutex, std :: shared_timed_mutex	353
MSVC14.1: std :: shared_mutex.....	353
MSVC14.1: std :: shared_timed_mutex.....	355
std :: shared_mutex / 2 , std :: shared_timed_.....	358
Examples.....	361
std :: unique_lock, std :: shared_lock, std :: lock_guard.....	361
: std :: try_to_lock, std :: _lock, std :: defer_l.....	362
::	363
std :: scoped_lock (C ++ 17).....	364
.....	364
::	364
63:	365
.....	365
.....	365
Examples.....	366
.....	366
.....	367
.....	367
.....	367

.....	368
.....	368
.....	369
().	370
.....	371
.....	372
.....	372
()	373
,	373
.....	374
`std` `posix`	374
.....	375
.....	376
.....	376
const.	376
.....	377
.....	377
.....	378
.....	379
[[noreturn]]	379
,	379
.....	380
64:	381
.....	381
Examples	381
globals TU	381
.....	382
void *	382
reinterpret_cast.	383
.....	383
,	384
.....	384

.....	386
65:	387
Examples.....	387
.....	387
.....	387
66:	389
.....	389
.....	389
Examples.....	389
.....	389
~	389
67: / (GCC)	391
Examples.....	391
: «***»	391
.....	391
.....	391
`*** '	392
: ***:	393
68: (ODR)	394
Examples.....	394
.....	394
.....	395
ODR	396
69:	397
.....	397
Examples.....	397
& -	397
-	398
^ - XOR ().....	398
~ - NOT ().....	400
<< -	401
>> -	402

70:	404
.....	404
Examples.....	404
/	404
.....	404
71: C ++	406
Examples.....	406
.....	406
.....	406
.....	407
.....	407
.....	407
/	408
.....	409
.....	409
.....	410
?	411
72:	412
Examples.....	412
.....	412
BOOL.....	412
.....	412
char16_t.....	412
char32_t.....	413
.....	413
.....	413
.....	413
.....	414
.....	414
wchar_t.....	415
73:	416
Examples.....	416

.....	416
.....	416
74: OpenMP	417
.....	417
.....	417
Examples.....	417
OpenMP:	417
OpenMP:	418
OpenMP:	419
OpenMP: /	419
75:	421
.....	421
.....	421
Examples.....	421
.....	421
.....	423
.....	424
.....	425
.....	426
.....	427
.....	428
NOT.....	429
/	430
.....	430
.....	434
76:	438
.....	438
.....	438
Examples.....	438
?.....	438
.....	440
cv-qualifier	440

77:	443
.....	443
Examples.....	443
?.....	443
78:	445
Examples.....	445
.....	445
.....	445
.....	447
std :: move $O(n^2)$ $O(n)$	448
.....	452
.....	452
79:	454
Examples.....	454
.....	454
switch.....	455
.....	455
.....	456
C ++ 11.....	457
80: ,	459
Examples.....	459
Char 	459
.....	459
char	459
.....	459
char16_t char32_t	461
bool	462
wchar_t	462
.....	462
.....	463
.....	463
.....	

.....	466
.....	466
(,)	467
81:	468
Examples	468
.....	468
82:	470
Examples	470
.....	470
.....	471
.....	473
83:	475
Examples	475
.....	475
.....	475
.....	476
.....	477
.....	477
84:	479
.....	479
.....	479
Examples	480
.....	480
85: C ++	482
.....	482
Examples	482
.....	482
.....	483
.....	483
.....	483
.....	

.....	484
.....	484
.....	485
iostream.....	485
.....	485
.....	486
.....	486
.....	487
.....	487
STL.....	487
.....	487
.....	488
86:	489
.....	489
.....	489
Examples.....	491
.....	491
.....	497
.....	498
87: , ...	501
Examples.....	501
.....	501
.....	502
.....	503
.....	505
88:	507
.....	507
.....	507
Examples.....	507
.....	507

-	508
.....	510
.....	514
.....	515
X-.....	517
#	519
.....	519
89:	521
Examples.....	521
, TCP-.....	521
, TCP-.....	524
90:	526
.....	526
Examples.....	527
.....	527
AND OR.....	527
&& :	527
.....	529
91:	530
.....	530
.....	530
.....	530
Examples.....	531
?.....	531
.....	532
.....	533
.....	533
.....	534
ADL	535
.....	535
/	537
.....	538
.....	

.....	539
.....	540
92:	541
Examples.....	541
gcc gprof.....	541
callgraph gperf2dot.....	542
gcc Google Perf Tools.....	543
93:	546
.....	546
Examples.....	546
.....	546
.....	547
.....	548
.....	549
.....	549
.....	551
.....	552
.....	553
94: C ++	556
.....	556
.....	556
Examples.....	556
.....	556
.....	559
.....	562
Builder Fluent API.....	563
.....	564
: Mutable object	565
95:	566
.....	566
.....	566

.....	566
Examples.....	567
regex_match regex_search.....	567
regex_replace.....	568
regex_token_iterator.....	568
regex_iterator.....	568
.....	569
.....	569
.....	571
96:	573
Examples.....	573
.....	573
C ++ -	573
97:	575
Examples.....	575
:: recursive_mutex.....	575
98: C ++.....	576
Examples.....	576
, ...	576
.....	576
99:	578
.....	578
Examples.....	578
.....	578
.....	579
.....	579
100:	581
.....	581
Examples.....	581
C ++ 11.....	581
.....	581
101:	583

.....	583
.....	583
Examples.....	583
CMake.....	583
GNU make.....	584
.....	584
.....	585
.....	586
.....	587
SCons.....	587
.....	588
.....	588
NMAKE (Microsoft).....	588
.....	588
Autotools (GNU).....	588
.....	588
102:	590
.....	590
Examples.....	590
.....	590
.....	590
.....	591
- (C ++ 11).....	592
.....	593
std :: map ().....	594
.....	596
103:	597
.....	597
Examples.....	597
.....	597
.....	597
.....	

104:	599
Examples	599
	599
	600
	600
	602
Destructor	604
105:	607
	607
	607
Examples	607
	607
	608
	608
	609
	610
106: C ++, C ++ vs	612
Examples	612
	612
107: C ++	614
	614
	614
Examples	614
C ++ segfault - valgrind	614
Segfault GDB	616
	617
	619
/	619
	619
	620
	620

.....	620
.....	621
.....	621
.....	621
()	621
?	622
?	622
?	622
.....	622
108: :: forward_list	623
.....	623
.....	623
Examples	623
.....	623
.....	624
109: :: integer_sequence	626
.....	626
Examples	626
std :: tuple	626
,	627
.....	627
110: :: iomanip	629
Examples	629
:: setw	629
:: setprecision	629
:: setfill	630
:: setiosflags	630
111: ::	633
.....	633
Examples	633
std :: variant	633
~	634

`std :: variant`	635
112: ::	637
.....	637
.....	637
Examples	637
std :: vector	637
.....	638
std :: vector	640
.....	640
.....	641
const	641
.....	642
.....	643
:	643
:	645
std :: vector C	646
/ Invalidation	647
.....	648
:	648
:	648
:	648
:	648
:	649
:	649
,	649
:	649
:	650
std :: vector	651
std :: vector	652
: , ...	653
.....	

.....656

.....657

.....657

, ...659

.....660

.....661

113: :: 663

.....663

Examples.....663

.....663

std :: map std :: multimap.....664

.....665

.....666

std :: map std :: multimap.....668

std :: map std :: multimap.....668

.....669

.....669

.....669

Multi-Map.....670

- ().....670

std :: map 671

.....671

114: :: 673

.....673

Examples.....673

.....673

115: :: 674

.....674

.....674

Examples.....674

std :: array.....	674
.....	675
.....	678
.....	678
.....	678
116: ::	679
Examples.....	679
.....	679
.....	679
.....	679
Sentinel.....	679
vs std::pair<bool, T>.....	679
.....	680
.....	680
.....	682
value_or.....	683
117: ::	684
Examples.....	684
.....	684
.....	684
118: ::	686
.....	686
.....	686
.....	687
Examples.....	687
.....	687
.....	688
.....	688
.....	689
.....	689
.....	690
⌈ 0.....	690

().....	690
().....	691
().....	691
.....	691
(const) char *.....	692
()	693
/	694
.....	695
std :: wstring.....	696
std :: string_view.....	698
.....	698
/	699
.....	700
,	701
std :: string.....	702
119: ISO C ++.....	704
.....	704
.....	704
Examples.....	705
.....	705
C ++ 11.....	705
.....	705
.....	705
.....	706
.....	706
.....	706
.....	706
.....	707
.....	707
.....	707
.....	707
.....	

C ++ 14.....	708
.....	708
.....	708
/	708
C ++ 17.....	709
.....	709
.....	709
C ++ 03.....	709
.....	709
++ 98.....	710
(C89 / C90).....	710
.....	710
C ++ 20.....	710
.....	710
.....	710
120:	712
Examples.....	712
:: for_each.....	712
:: next_permutation.....	712
::	713
::	715
::	716
:: count_if.....	717
:: find_if.....	719
:: min_element.....	720
std :: nth_element ().....	722
121: ::	724
Examples.....	724
.....	724
122: C ++.....	727

Examples.....	727
Linked List C ++.....	727
123:	730
.....	730
Examples.....	730
:: shared_lock.....	730
std :: call_once, std :: once_flag.....	730
.....	731
std :: condition_variable_any, std :: cv_status.....	732
124: C ++	733
.....	733
Examples.....	733
Google.....	733
.....	733
.....	734
125: Erasure	736
.....	736
Examples.....	736
.....	736
Regular vtable.....	737
`std :: function`.....	740
T.....	743
std :: any.....	744
126:	750
Examples.....	750
.....	750
.....	751
.....	751
.....	753
127:	754
.....	754
.....	754

Examples.....	754
typedef.....	754
typedef.....	755
typedef.....	755
"".....	755
128:	757
.....	757
Examples.....	757
.....	757
.....	757
.....	757
.....	758
std :: is_same.....	758
.....	759
.....	761
129:	763
.....	763
.....	763
.....	763
Examples.....	763
.....	763
.....	763
.....	764
.....	765
.....	766
.....	766
.....	767
/	767
/	767
.....	768
130:	769

.....	769
Examples.....	769
".....	769
".....	770
".....	770
".....	771
131:	773
.....	773
.....	773
Examples.....	773
(std :: shared_ptr).....	773
(std :: weak_ptr).....	776
(std :: unique_ptr).....	778
C.....	780
(auto_ptr).....	782
shared_ptr, ...	783
std :: shared_ptr	784
: value_ptr.....	785
132:	788
.....	788
.....	788
Examples.....	788
.....	789
(, ...).....	789
.....	791
133:	793
.....	793
Examples.....	793
.....	793
.....	793
.....	794
.....	794
.....	794

.....	795
.....	795
.....	796
.....	796
.....	797
: if, if..else.....	798
: break, continue, goto, exit.....	799
134:	803
.....	803
Examples.....	803
.....	803
.....	804
135: -	807
.....	807
Examples.....	807
.....	807
.....	808
.....	810
.....	811
.....	812
.....	813
ASCII std ::.....	813
.....	815
`struct`	816
.....	817
, ?.....	817
.....	818
136:	820
.....	820
Examples.....	820
.....	821
.....	

.....	822
.....	822
137:	824
.....	824
.....	824
Examples.....	824
.....	824
.....	825
.....	826
.....	828
Const Correctness.....	831
138: C ++ « » « »	833
.....	833
Examples.....	833
.....	833
139:	835
.....	835
Examples.....	835
std :: future std :: prom.....	835
.....	835
std :: packaged_task std :: future.....	836
std :: future_error std :: future_errc.....	836
std :: future std :: async.....	838
.....	839
140:	840
.....	840
.....	840
.....	840
Examples.....	840
Unix-	840
C, C ++.....	841

141:	842
.....	842
.....	842
Examples.....	842
.....	842
.....	843
142:	845
Examples.....	845
.....	845
143: Singleton	847
.....	847
Examples.....	847
.....	847
.....	848
Singeton.....	849
-	850
144:	851
.....	851
.....	851
.....	851
Examples.....	853
.....	853
.....	855
.....	855
.....	856
.....	857
.....	858
.....	859
.....	859
non-type	860
unique_ptr.....	860
.....	861

Variadic.....	862
.....	865
145:	867
Examples.....	867
.....	867
vec.hh: wrapper std :: vector, ...	869
expr.hh: (.....	871
main.cc: test src.....	874
, ...	877
146:	882
.....	882
Examples.....	882
.....	882
.....	885
.....	885
CV-Qualifiers.....	887
Ref-Qualifiers.....	890
147:	892
.....	892
.....	892
.....	892
Examples.....	893
.....	893
.....	894
punning.....	894
.....	895
.....	896
.....	896
Enum.....	897
.....	898
void * to T *.....	899
C.....	899
.....	

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [cplusplus](#)

It is an unofficial and free C++ ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official C++.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с C ++

замечания

Программа «Hello World» является распространенным примером, который можно просто использовать для проверки наличия компилятора и библиотеки. Он использует стандартную библиотеку C ++ с `std::cout` из `<iostream>` и имеет только один файл для компиляции, что минимизирует вероятность возможной ошибки пользователя во время компиляции.

Процесс компиляции C ++-программы по своей сути отличается между компиляторами и операционными системами. В разделе « [Компиляция и построение](#) » содержатся сведения о том, как скомпилировать код C ++ на разных платформах для различных компиляторов.

Версии

Версия	стандарт	Дата выхода
C ++ 98	ISO / IEC 14882: 1998	1998-09-01
C ++ 03	ISO / IEC 14882: 2003	2003-10-16
C ++ 11	ISO / IEC 14882: 2011	2011-09-01
C ++ 14	ISO / IEC 14882: 2014	2014-12-15
C ++ 17	TBD	2017-01-01
C ++ 20	TBD	2020-01-01

Examples

Привет, мир

Эта программа печатает `Hello World!` к стандартному потоку вывода:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
}
```

Смотрите на [живую в Колиру](#) .

Анализ

Давайте подробно рассмотрим каждую часть этого кода:

- `#include <iostream>` - это **препроцессорная директива**, которая включает в себя содержимое стандартного файла `iostream` заголовочного файла C ++.

`iostream` - это **стандартный файл заголовка библиотеки**, который содержит определения стандартных потоков ввода и вывода. Эти определения включены в пространство имен `std` , как описано ниже.

Стандартные потоки ввода / вывода (I / O) обеспечивают способы ввода и вывода программ во внешнюю систему - обычно терминал.

- `int main() { ... }` определяет новую **функцию** с именем `main` . По соглашению `main` функция называется исполнением программы. В программе на C ++ должна быть только одна `main` функция, и она должна всегда возвращать число типа `int` .

Здесь `int` - это то, что называется **возвращаемым типом функции** . Значение, возвращаемое `main` функцией, - это **код выхода**.

По соглашению код выхода программы `0` или `EXIT_SUCCESS` интерпретируется как успех системой, выполняющей программу. Любой другой код возврата связан с ошибкой.

Если оператор `return` не присутствует, `main` функция (и, следовательно, сама программа) возвращает `0` по умолчанию. В этом примере нам не нужно явно писать `return 0;` ,

Все остальные функции, кроме тех, которые возвращают тип `void` , должны явно возвращать значение в соответствии с их типом возвращаемого значения или иначе не должны возвращаться вообще.

- `std::cout << "Hello World!" << std::endl;` печатает «Hello World!» к стандартному потоку вывода:
 - `std` - **пространство имен** , а `::` - **оператор разрешения области**, который позволяет искать объекты по имени в пространстве имен.

Существует много пространств имен. Здесь мы используем `::` чтобы показать, что мы хотим использовать `cout` из пространства имен `std` . Для получения дополнительной информации см. [Оператор разрешения области действия - Документация Microsoft](#) .

- `std::cout` - это **стандартный объект потока вывода** , определенный в `iostream` , и он выводит на стандартный вывод (`stdout`).
- `<<` является в этом контексте **оператором вставки потока** , так называемым, потому что он *вставляет* объект в объект *потока* .

Стандартная библиотека определяет оператора `<<` для выполнения ввода данных для определенных типов данных в выходные потоки. `stream << content` вставляет `content` в поток и возвращает тот же, но обновленный поток. Это позволяет привязывать потоковые вставки: `std::cout << "Foo" << " Bar";` печатает «FooBar» на консоли.

- `"Hello World!"` является **символьным строковым литералом** или «текстовым литералом». Оператор вставки потока для символьных строковых литералов определяется в файле `iostream` .
- `std::endl` - специальный объект **манипулятора ввода-вывода** , также определенный в файле `iostream` . Вставка манипулятора в поток изменяет состояние потока.

Манипулятор потока `std::endl` выполняет две функции: сначала он вставляет символ конца строки, а затем очищает буфер потока, чтобы текст отображался на консоли. Это гарантирует, что данные, вставленные в поток, действительно отображаются на консоли. (Данные о потоке обычно хранятся в буфере, а затем «размываются» партиями, если вы немедленно не отключите флеш.)

Альтернативный метод, который позволяет избежать флеша, заключается в следующем:

```
std::cout << "Hello World!\n";
```

где `\n` - **последовательность escape символов** для символа новой строки.

- Точка с запятой (;) уведомляет компилятор о завершении оператора. Все инструкции C ++ и определения классов требуют конечной / конечной точки с запятой.

Комментарии

Комментарий - это способ помещать произвольный текст внутри исходного кода, не компилятор C ++ интерпретирует его с любым функциональным значением. Комментарии используются, чтобы дать представление о дизайне или методе программы.

В C ++ есть два типа комментариев:

Однострочные комментарии

Двойная последовательность косой черты `//` будет отмечать весь текст до появления новой строки в виде комментария:

```
int main()
{
    // This is a single-line comment.
    int a; // this also is a single-line comment
    int i; // this is another single-line comment
}
```

Комментарии к C-Style / Block

Последовательность `/*` используется для объявления начала блока комментариев, а последовательность `*/` используется для объявления конца комментария. Весь текст между начальной и конечной последовательностями интерпретируется как комментарий, даже если текст в остальном является синтаксисом C++. Они иногда называются комментариями «C-style», так как этот синтаксис комментария унаследован от языка предшественника C++, C:

```
int main()
{
    /*
     * This is a block comment.
     */
    int a;
}
```

В любом комментарии блока вы можете написать все, что хотите. Когда компилятор встречает символ `*/`, он завершает комментарий блока:

```
int main()
{
    /* A block comment with the symbol /*
       Note that the compiler is not affected by the second /*
       however, once the end-block-comment symbol is reached,
       the comment ends.
    */
    int a;
}
```

Вышеприведенный пример является допустимым кодом C++ (и C). Однако наличие дополнительного `/*` внутри комментария блока может привести к предупреждению некоторых компиляторов.

Блочные комментарии также могут начинаться и заканчиваться в одной строке. Например:

```
void SomeFunction(/* argument 1 */ int a, /* argument 2 */ int b);
```

Важность комментариев

Как и во всех языках программирования, комментарии предоставляют несколько преимуществ:

- Явная документация кода, упрощающая чтение / обслуживание
- Объяснение цели и функциональности кода
- Подробная информация об истории или аргументах кода
- Размещение авторских прав / лицензий, примечаний к проектам, особая благодарность, кредиты вкладчиков и т. Д. Непосредственно в исходном коде.

Однако комментарии также имеют свои недостатки:

- Они должны поддерживаться, чтобы отражать любые изменения в коде
- Чрезмерные комментарии, как правило, делают код *менее* удобочитаемым

Потребность в комментариях может быть уменьшена путем написания четкого, самодокументирующего кода. Простым примером является использование пояснительных имен для переменных, функций и типов. Факторизация логически связанных задач в дискретные функции идет рука об руку с этим.

Маркеры комментариев, используемые для отключения кода

Во время разработки комментарии также могут использоваться для быстрого отключения частей кода без его удаления. Это часто полезно для тестирования или отладки, но не является хорошим стилем для чего угодно, кроме временных изменений. Это часто называют «комментированием».

Точно так же сохранение старых версий части кода в комментарии для справочных целей не одобряется, поскольку оно загромождает файлы, при этом мало ценное по сравнению с исследованием истории кода через систему управления версиями.

функция

Функция - это единица кода, представляющая последовательность операторов.

Функции могут принимать **аргументы** или значения и **возвращать** одно значение (или нет). Для использования функции для значений аргументов используется **вызов** функции, а использование самого вызова функции заменяется его возвращаемым значением.

Каждая функция имеет **сигнатуру типа** - типы ее аргументов и тип возвращаемого типа.

Функции вдохновлены понятиями процедуры и математической функции.

- **Примечание.** Функции C ++ являются по существу процедурами и не соответствуют точному определению или правилам математических функций.

Функции часто предназначены для выполнения конкретной задачи. и может быть вызван из других частей программы. Функция должна быть объявлена и определена до ее вызова в другом месте программы.

- **Примечание:** популярные определения функций могут быть скрыты в других включенных файлах (часто для удобства и повторного использования во многих файлах). Это обычное использование файлов заголовков.

Объявление функции

Объявление **функции** объявляет о существовании функции с ее именем и сигнатурой типа компилятору. Синтаксис следующий:

```
int add2(int i); // The function is of the type (int) -> (int)
```

В приведенном выше примере `int add2(int i)` объявляет компилятору следующее:

- Тип **возврата** - `int` .
- **Имя** функции - `add2` .
- **Число аргументов** функции равно 1:
 - Первый аргумент имеет тип `int` .
 - Первый аргумент будет обозначен в содержимом функции именем `i` .

Имя аргумента необязательно; декларация для функции также может быть следующей:

```
int add2(int); // Omitting the function arguments' name is also permitted.
```

В соответствии с **одним правилом** определения функция с определенной сигнатурой типа может быть объявлена или определена только один раз во всей кодовой базе C ++, видимой компилятору C ++. Другими словами, функции с определенной сигнатурой типа не могут быть переопределены - их нужно определять только один раз. Таким образом, недопустимо следующее: C ++:

```
int add2(int i); // The compiler will note that add2 is a function (int) -> int
int add2(int j); // As add2 already has a definition of (int) -> int, the compiler
                // will regard this as an error.
```

Если функция ничего не возвращает, ее возвращаемый тип записывается как `void` . Если он

не принимает никаких параметров, список параметров должен быть пустым.

```
void do_something(); // The function takes no parameters, and does not return anything.
                    // Note that it can still affect variables it has access to.
```

Вызов функции

Функция может быть вызвана после ее объявления. Например, следующая программа вызывает `add2` со значением 2 в функции `main`:

```
#include <iostream>

int add2(int i);    // Declaration of add2

// Note: add2 is still missing a DEFINITION.
// Even though it doesn't appear directly in code,
// add2's definition may be LINKED in from another object file.

int main()
{
    std::cout << add2(2) << "\n"; // add2(2) will be evaluated at this point,
                                   // and the result is printed.

    return 0;
}
```

Здесь `add2(2)` является синтаксисом вызова функции.

Определение функции

Определение *функции* * похоже на объявление, за исключением того, что оно также содержит код, который выполняется, когда функция вызывается внутри ее тела.

Примером определения функции для `add2` может быть:

```
int add2(int i)      // Data that is passed into (int i) will be referred to by the name i
{                    // while in the function's curly brackets or "scope."

    int j = i + 2;    // Definition of a variable j as the value of i+2.
    return j;         // Returning or, in essence, substitution of j for a function call to
                      // add2.
}
```

Перегрузка функции

Вы можете создавать несколько функций с тем же именем, но с разными параметрами.

```
int add2(int i)      // Code contained in this definition will be evaluated
```

```

{
    int j = i + 2;
    return j;
}

// when add2() is called with one parameter.

int add2(int i, int j)    // However, when add2() is called with two parameters, the
{
    // code from the initial declaration will be overloaded,
    int k = i + j + 2 ;   // and the code in this declaration will be evaluated
    return k;             // instead.
}

```

Обе функции вызываются одним и тем же именем `add2` , но действительная функция, которая вызывается, напрямую зависит от количества и типа параметров в вызове. В большинстве случаев компилятор C ++ может вычислить, какую функцию вызывать. В некоторых случаях тип должен быть явно указан.

Параметры по умолчанию

Значения по умолчанию для параметров функции могут быть указаны только в объявлениях функций.

```

int multiply(int a, int b = 7); // b has default value of 7.
int multiply(int a, int b)
{
    return a * b;
}
// If multiply() is called with one parameter, the
// value will be multiplied by the default, 7.

```

В этом примере `multiply()` можно вызвать с одним или двумя параметрами. Если задан только один параметр, `b` будет иметь значение по умолчанию 7. По умолчанию аргументы должны быть помещены в последние аргументы функции. Например:

```

int multiply(int a = 10, int b = 20); // This is legal
int multiply(int a = 10, int b);      // This is illegal since int a is in the former

```

Специальные вызовы функций - Операторы

Существуют специальные вызовы функций в C ++, которые имеют различный синтаксис, чем `name_of_function(value1, value2, value3)` . Наиболее распространенным примером является оператор.

Некоторые специальные последовательности символов, которые будут сведены к вызову функции компилятором, например `!` , `+` , `-` , `*` , `%` , и `<<` и многое другое. Эти специальные символы обычно связаны с использованием не-программирования или используются для эстетики (например, символ `+` обычно распознается как символ добавления как в

программировании на C ++, так и в элементарной математике).

C ++ обрабатывает эти последовательности символов специальным синтаксисом; но, по существу, каждое возникновение оператора сводится к вызову функции. Например, следующее выражение C ++:

```
3+3
```

эквивалентно следующему вызову функции:

```
operator+(3, 3)
```

Все имена функций оператора начинаются с `operator`.

В то время как в непосредственном предшественнике C ++ имена операторов C не могут присваиваться различным значениям, предоставляя дополнительные определения с сигнатурами разных типов, в C ++ это действительно. «Скрытие» дополнительных определений функций под одним уникальным именем функции называется **перегрузкой оператора** на C ++ и является относительно распространенным, но не универсальным соглашением на C ++.

Видимость прототипов функций и деклараций

В C ++ код должен быть объявлен или определен перед использованием. Например, следующее создает ошибку времени компиляции:

```
int main()
{
    foo(2); // error: foo is called, but has not yet been declared
}

void foo(int x) // this later definition is not known in main
{
}
```

Это можно решить двумя способами: поставить определение или декларацию `foo()` перед его использованием в `main()`. Вот один пример:

```
void foo(int x) {} //Declare the foo function and body first

int main()
{
    foo(2); // OK: foo is completely defined beforehand, so it can be called here.
}
```

Однако также возможно «форвард-объявить» функцию, помещая только объявление «прототипа» перед ее использованием и затем определяя тело функции позже:

```
void foo(int); // Prototype declaration of foo, seen by main
               // Must specify return type, name, and argument list types
int main()
{
    foo(2); // OK: foo is known, called even though its body is not yet defined
}

void foo(int x) //Must match the prototype
{
    // Define body of foo here
}
```

Прототип должен указывать тип возвращаемого значения (`void`), имя функции (`foo`) и типы переменных списка аргументов (`int`), но **имена аргументов НЕ требуются** .

Одним из распространенных способов интегрирования этого в организацию исходных файлов является создание файла заголовка, содержащего все объявления прототипов:

```
// foo.h
void foo(int); // prototype declaration
```

а затем предоставить полное определение в другом месте:

```
// foo.cpp --> foo.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
void foo(int x) { } // foo's body definition
```

а затем, после компиляции, связать соответствующий файл объекта `foo.o` в скомпилированном объектном файле, где он используется в фазе связывания, `main.o` :

```
// main.cpp --> main.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
int main() { foo(2); } // foo is valid to call because its prototype declaration was
beforehand.
// the prototype and body definitions of foo are linked through the object files
```

Ошибка «неразрешенного внешнего символа» возникает, когда *прототип* функции и *вызов* существуют, но *тело* функции не определено. Это может быть сложнее решить, поскольку компилятор не сообщит об ошибке до этапа окончательной компоновки, и он не знает, в какую строку перейти в код, чтобы показать ошибку.

Стандартный процесс компиляции C ++

Исполняемый программный код C ++ обычно создается компилятором.

Компилятор - это программа, которая переводит код с языка программирования в другую форму, которая (более) выполняется непосредственно для компьютера. Использование компилятора для перевода кода называется **компиляцией**.

C ++ наследует форму своего процесса компиляции с его «родительского» языка, C. Ниже

приведен список четырех основных шагов компиляции в C ++:

1. Препроцессор C ++ копирует содержимое любых включенных заголовочных файлов в файл исходного кода, генерирует макрокod и заменяет символические константы, определенные с помощью `#define`, с их значениями.
 2. Расширенный файл исходного кода, созданный препроцессором C ++, скомпилирован в ассемблере, подходящем для платформы.
 3. Код ассемблера, сгенерированный компилятором, собран в соответствующий объектный код для платформы.
 4. Файл объектного кода, сгенерированный ассемблером, связан вместе с файлами объектных кодов для любых библиотечных функций, используемых для создания исполняемого файла.
- Примечание: некоторые скомпилированные коды связаны друг с другом, но не для создания окончательной программы. Обычно этот «связанный» код также может быть упакован в формат, который может использоваться другими программами. Этот «пакет упакованного, полезного кода» - это то, что программисты на C ++ называют **библиотекой**.

Многие компиляторы C ++ могут также объединять или объединять определенные части процесса компиляции для удобства или для дополнительного анализа. Многие программисты на C ++ будут использовать разные инструменты, но все инструменты, как правило, следуют этому обобщенному процессу, когда они участвуют в создании программы.

Ссылка ниже расширяет эту дискуссию и обеспечивает приятную графику. [1]:
<http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html>

препроцессор

Препроцессор является важной частью [компилятора](#).

Он редактирует исходный код, вырезает несколько битов, меняет другие и добавляет другие вещи.

В исходных файлах мы можем включать директивы препроцессора. Эти директивы говорят препроцессору выполнять определенные действия. Директива начинается с `#` на новой строке. Пример:

```
#define ZERO 0
```

Первая препроцессорная директива, которую вы встретите, вероятно, является

```
#include <something>
```


директивы. Что он делает, так это все `something` и вставляет в файл, где была указатель.

Всемирная программа приветствия начинается с линии

```
#include <iostream>
```

Эта строка добавляет функции и объекты, которые позволяют использовать стандартный ввод и вывод.

Язык C, который также использует препроцессор, не имеет столько заголовочных файлов, как язык C ++, но на C ++ вы можете использовать все файлы заголовков C.

Следующей важной директивой, вероятно, является

```
#define something something_else
```

директивы. Это говорит препроцессору, что, когда он идет по файлу, он должен заменить каждое появление `something something_else` . Это также может сделать вещи похожими на функции, но это, вероятно, считается продвинутым C ++.

`something_else` не требуется, но если вы определите `something` как ничто, то внешние директивы препроцессора, все вхождения `something` то исчезнут.

Это действительно полезно из-за `#if` , `#else` и `#ifdef` . Формат для них будет следующим:

```
#if something==true
//code
#else
//more code
#endif

#ifdef thing_that_you_want_to_know_if_is_defined
//code
#endif
```

Эти директивы вставляют код, который находится в истинном бите, и удаляют ложные биты. это может быть использовано для того, чтобы иметь биты кода, которые включены только в определенные операционные системы, без необходимости переписывать весь код.

Прочитайте Начало работы с C ++ онлайн: <https://riptutorial.com/ru/cplusplus/topic/206/начало-работы-с-с-plusplus>

глава 2: C несовместимости

Вступление

Это описывает, какой код C будет разбит на компилятор C ++.

Examples

Зарезервированные ключевые слова

Первый пример - это ключевые слова, которые имеют особую цель в C ++: в C есть законный, но не C ++.

```
int class = 5
```

Эти ошибки легко исправить: просто переименуйте переменную.

Слабо напечатанные указатели

В C указатели могут быть переведены в `void*`, для которых требуется явное преобразование в C ++. В C ++ существует недопустимое, но законное в C:

```
void* ptr;  
int* intptr = ptr;
```

Добавление явного приведения делает эту работу, но может вызвать дополнительные проблемы.

перейти или переключиться

В C ++ вы не можете пропустить инициализацию с помощью `goto` или `switch`. В C действует, но не C ++:

```
goto foo;  
int skipped = 1;  
foo;
```

Эти ошибки могут потребовать редизайна.

Прочитайте C несовместимости онлайн: <https://riptutorial.com/ru/cplusplus/topic/9645/c-несовместимости>

глава 3: Const Correctness

Синтаксис

- `class ClassOne {public: bool non_modifying_member_function () const {/ * ... * /};`
- `int ClassTwo :: non_modifying_member_function () const {/ * ... * /}`
- `void ClassTwo :: modifying_member_function () {/ * ... * /}`
- `char non_param_modding_func (const ClassOne & one, const ClassTwo * two) {/ * ... * /}`
- `float parameter_modification_function (ClassTwo & one, ClassOne * two) {/ * ... * /}`
- `short ClassThree :: non_modding_non_param_modding_f (const ClassOne &) const {/ * ... * /}`

замечания

`const` корректность - очень полезный инструмент устранения неполадок, так как он позволяет программисту быстро определить, какие функции могут непреднамеренно модифицировать код. Он также предотвращает непреднамеренные ошибки, такие как показанные в `Const Correct Function Parameters`, правильные и незаметные компиляции.

Гораздо проще создать класс для корректности `const`, чем позже добавить `const` корректность к существующему классу. Если возможно, спроектируйте любой класс, который *может* быть `const` правильным, чтобы он *был* `const` правильным, чтобы спасти себя, а другие - из-за того, что позже его модифицировали.

Обратите внимание, что это также может быть применено к `volatile` правильности, если необходимо, с теми же правилами, что и для `const` корректности, но это используется гораздо реже.

Рефракции:

[ISO_CPP](#)

[Продайте меня на const const](#)

[Учебник C ++](#)

Examples

Основы

`const` *корректность* - это практика разработки кода, так что только код, который *должен* изменить экземпляр, *может* модифицировать экземпляр (т. е. имеет доступ на запись), и, наоборот, что любой код, который не нуждается в изменении экземпляра, не может

сделать поэтому (т.е. имеет доступ только для чтения). Это предотвращает непреднамеренное изменение экземпляра, делая код менее ошибочным и документирует, предназначен ли код для изменения состояния экземпляра или нет. Он также позволяет экземплярам обрабатываться как `const` когда они не нуждаются в модификации или определяются как `const` если их не нужно изменять после инициализации, не теряя при этом никакой функциональности.

Это делается путем предоставления функций-членов `const` **CV-квалификаторов** и путем создания указателей / ссылочных параметров `const` , за исключением случаев, когда им нужен доступ на запись.

```
class ConstCorrectClass {
    int x;

public:
    int getX() const { return x; } // Function is const: Doesn't modify instance.
    void setX(int i) { x = i; }    // Not const: Modifies instance.
};

// Parameter is const: Doesn't modify parameter.
int const_correct_reader(const ConstCorrectClass& c) {
    return c.getX();
}

// Parameter isn't const: Modifies parameter.
void const_correct_writer(ConstCorrectClass& c) {
    c.setX(42);
}

const ConstCorrectClass invariant; // Instance is const: Can't be modified.
ConstCorrectClass variant; // Instance isn't const: Can be modified.

// ...

const_correct_reader(invariant); // Good.    Calling non-modifying function on const instance.
const_correct_reader(variant);    // Good.    Calling non-modifying function on modifiable
instance.

const_correct_writer(variant);    // Good.    Calling modifying function on modifiable instance.
const_correct_writer(invariant); // Error.    Calling modifying function on const instance.
```

Из-за характера корректности `const` это начинается с функций-членов класса и выходит наружу; если вы попытаетесь вызвать функцию non-`const` члена из экземпляра `const` или из не-`const` экземпляра, который рассматривается как `const` , компилятор даст вам ошибку об этом, теряя cv-квалификаторы.

Const Correct Class Design

В `const` -correct класса, все функции - члены , которые не изменяются логическое состояния есть `this` CV-квалифицируются как `const` , указывая , что они не изменяют объект (за исключением любых `mutable` полей, которые свободно могут быть изменены даже в `const` случаях); если функция `const` cv-qual возвращает ссылку, эта ссылка также должна быть

`const` . Это позволяет им вызываться как с константами, так и с не-св-критериями, поскольку `const T*` может связываться с `T*` или `const T*` . Это, в свою очередь, позволяет функциям объявлять свои передаваемые по параметрам параметры как `const` когда их не нужно изменять, не теряя при этом никакой функциональности.

Кроме того, в `const` правильном классе все параметры функции `pass-by-reference` будут `const` , как описано в `Const Correct Function Parameters` , так что они могут быть изменены только тогда, когда функция явно *должна* их модифицировать.

Во-первых, давайте посмотрим на `this` св-квалификаторы:

```
// Assume class Field, with member function "void insert_value(int);".

class ConstIncorrect {
    Field fld;

public:
    ConstIncorrect(Field& f); // Modifies.

    Field& getField();        // Might modify. Also exposes member as non-const reference,
                               // allowing indirect modification.
    void setField(Field& f);  // Modifies.

    void doSomething(int i);  // Might modify.
    void doNothing();         // Might modify.
};

ConstIncorrect::ConstIncorrect(Field& f) : fld(f) {} // Modifies.
Field& ConstIncorrect::getField() { return fld; }    // Doesn't modify.
void ConstIncorrect::setField(Field& f) { fld = f; } // Modifies.
void ConstIncorrect::doSomething(int i) {            // Modifies.
    fld.insert_value(i);
}
void ConstIncorrect::doNothing() {}                  // Doesn't modify.

class ConstCorrectCVQ {
    Field fld;

public:
    ConstCorrectCVQ(Field& f); // Modifies.

    const Field& getField() const; // Doesn't modify. Exposes member as const reference,
                                    // preventing indirect modification.
    void setField(Field& f);      // Modifies.

    void doSomething(int i);      // Modifies.
    void doNothing() const;      // Doesn't modify.
};

ConstCorrectCVQ::ConstCorrectCVQ(Field& f) : fld(f) {}
Field& ConstCorrectCVQ::getField() const { return fld; }
void ConstCorrectCVQ::setField(Field& f) { fld = f; }
void ConstCorrectCVQ::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrectCVQ::doNothing() const {}
```

```

// This won't work.
// No member functions can be called on const ConstIncorrect instances.
void const_correct_func(const ConstIncorrect& c) {
    Field f = c.getField();
    c.do_nothing();
}

// But this will.
// getField() and doNothing() can be called on const ConstCorrectCVQ instances.
void const_correct_func(const ConstCorrectCVQ& c) {
    Field f = c.getField();
    c.do_nothing();
}

```

Затем мы можем объединить это с Const Correct Function Parameters , в результате чего класс будет полностью `const -correct`.

```

class ConstCorrect {
    Field fld;

public:
    ConstCorrect(const Field& f); // Modifies instance. Doesn't modify parameter.

    const Field& getField() const; // Doesn't modify. Exposes member as const reference,
                                   // preventing indirect modification.
    void setField(const Field& f); // Modifies instance. Doesn't modify parameter.

    void doSomething(int i); // Modifies. Doesn't modify parameter (passed by value).
    void doNothing() const; // Doesn't modify.
};

ConstCorrect::ConstCorrect(const Field& f) : fld(f) {}
Field& ConstCorrect::getField() const { return fld; }
void ConstCorrect::setField(const Field& f) { fld = f; }
void ConstCorrect::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrect::doNothing() const {}

```

Это также может быть объединено с перегрузки на основе `const` Несс, в том случае, если мы хотим одно поведение , если экземпляр является `const` , и другое поведение , если оно не; Обычное использование для этого - это константы, предоставляющие аксессуры, которые допускают модификацию, если сам контейнер не `const` .

```

class ConstCorrectContainer {
    int arr[5];

public:
    // Subscript operator provides read access if instance is const, or read/write access
    // otherwise.
    int& operator[](size_t index) { return arr[index]; }
    const int& operator[](size_t index) const { return arr[index]; }

    // ...
};

```

Это обычно используется в стандартной библиотеке, причем большинство контейнеров обеспечения перегрузки принять `const` ность во внимание.

Const Правильные функциональные параметры

В `const` правильной функции все параметры, передаваемые по ссылке, помечены как `const` если функция прямо или косвенно не изменяет их, не позволяя программисту непреднамеренно изменять то, что они не хотели изменять. Это позволяет функции принимать как `const` и неконвертированные экземпляры, и, в свою очередь, приводит к тому, что экземпляр `this` типа имеет тип `const T*` при вызове функции-члена, где `T` является типом класса.

```
struct Example {
    void func()          { std::cout << 3 << std::endl; }
    void func() const { std::cout << 5 << std::endl; }
};

void const_incorrect_function(Example& one, Example* two) {
    one.func();
    two->func();
}

void const_correct_function(const Example& one, const Example* two) {
    one.func();
    two->func();
}

int main() {
    Example a, b;
    const_incorrect_function(a, &b);
    const_correct_function(a, &b);
}

// Output:
3
3
5
5
```

Хотя последствия этого менее сразу видно, чем `const` правильное проектирование класса (в этом `const` -correct функций и `const` -incorrect классов вызовут ошибку компиляции, в то время как `const` классы -correct и `const` -incorrect функция компилируют правильно), `const` правильно функции поймут много ошибок, которые будут некорректными функциями `const`, например, ниже. [Обратите внимание, однако, что функция `const` -incorrect *приведет* к ошибкам компиляции, если передана экземпляр `const` если ожидается `const` .]

```
// Read value from vector, then compute & return a value.
// Caches return values for speed.
template<typename T>
const T& bad_func(std::vector<T>& v, Helper<T>& h) {
    // Cache values, for future use.
    // Once a return value has been calculated, it's cached & its index is registered.
    static std::vector<T> vals = {};
```

```

int v_ind = h.get_index(); // Current working index for v.
int vals_ind = h.get_cache_index(v_ind); // Will be -1 if cache index isn't registered.

if (vals.size() && (vals_ind != -1) && (vals_ind < vals.size()) && !(h.needs_recalc())) {
    return vals[h.get_cache_index(v_ind)];
}

T temp = v[v_ind];

temp -= h.poll_device();
temp *= h.obtain_random();
temp += h.do_tedious_calculation(temp, v[h.get_last_handled_index()]);

// We're feeling tired all of a sudden, and this happens.
if (vals_ind != -1) {
    vals[vals_ind] = temp;
} else {
    v.push_back(temp); // Oops. Should've been accessing vals.
    vals_ind = vals.size() - 1;
    h.register_index(v_ind, vals_ind);
}

return vals[vals_ind];
}

// Const correct version. Is identical to above version, so most of it shall be skipped.
template<typename T>
const T& good_func(const std::vector<T>& v, Helper<T>& h) {
    // ...

    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp); // Error: discards qualifiers.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }

    return vals[vals_ind];
}

```

Const Correctness как документация

Одна из наиболее полезных вещей о корректности `const` заключается в том, что она служит способом документирования кода, предоставляя определенные гарантии программисту и другим пользователям. Эти гарантии исполняются компилятором благодаря `const` Несс, с отсутствием `const` Несс, в свою очередь, указывающей, что код не обеспечивает их.

`const` CV-Qualified Member Functions:

- Можно предположить, что любая функция-член, которая является `const` имеет намерение прочитать экземпляр и:
 - Не изменять логическое состояние экземпляра, на который они вызваны.

Поэтому они не должны изменять какие-либо переменные-члены экземпляра, на который они вызывают, кроме `mutable` переменных.

- Не следует вызывать любые *другие* функции, которые изменяли бы любые переменные-члены экземпляра, кроме `mutable` переменных.
- И наоборот, любая функция-член, которая не является `const` может предполагать, что она имеет намерение изменить экземпляр и:
 - Может или не может изменять логическое состояние.
 - Может или не может вызывать другие функции, которые изменяют логическое состояние.

Это можно использовать, чтобы сделать предположения о состоянии объекта после вызова любой из функций-членов, даже не видя определения этой функции:

```
// ConstMemberFunctions.h

class ConstMemberFunctions {
    int val;
    mutable int cache;
    mutable bool state_changed;

public:
    // Constructor clearly changes logical state.  No assumptions necessary.
    ConstMemberFunctions(int v = 0);

    // We can assume this function doesn't change logical state, and doesn't call
    // set_val().  It may or may not call squared_calc() or bad_func().
    int calc() const;

    // We can assume this function doesn't change logical state, and doesn't call
    // set_val().  It may or may not call calc() or bad_func().
    int squared_calc() const;

    // We can assume this function doesn't change logical state, and doesn't call
    // set_val().  It may or may not call calc() or squared_calc().
    void bad_func() const;

    // We can assume this function changes logical state, and may or may not call
    // calc(), squared_calc(), or bad_func().
    void set_val(int v);
};
```

Из-за правил `const` эти предположения будут фактически выполняться компилятором.

```
// ConstMemberFunctions.cpp

ConstMemberFunctions::ConstMemberFunctions(int v /* = 0 */)
    : cache(0), val(v), state_changed(true) {}

// Our assumption was correct.
int ConstMemberFunctions::calc() const {
    if (state_changed) {
        cache = 3 * val;
        state_changed = false;
    }
}
```

```

    return cache;
}

// Our assumption was correct.
int ConstMemberFunctions::squared_calc() const {
    return calc() * calc();
}

// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers.
void ConstMemberFunctions::bad_func() const {
    set_val(863);
}

// Our assumption was correct.
void ConstMemberFunctions::set_val(int v) {
    if (v != val) {
        val = v;
        state_changed = true;
    }
}

```

`const` Параметры функции:

- Можно предположить, что любая функция с одним или несколькими параметрами, которые являются `const` имеет намерение читать эти параметры и:
 - Не изменять эти параметры или вызывать какие-либо функции-члены, которые бы их модифицировали.
 - Не передавать эти параметры в любую *другую* функцию, которая бы их модифицировала и / или вызывала любые функции-члены, которые могли бы их модифицировать.
- И наоборот, любая функция с одним или несколькими параметрами, которые не являются `const` может иметь намерение изменить эти параметры и:
 - Может или не может изменять эти параметры или вызывать какие-либо функции-члены, которые должны их модифицировать.
 - Может или не может передать эти параметры другим функциям, которые будут изменять их и / или вызывать любые функции-члены, которые будут их изменять.

Это можно использовать, чтобы сделать предположения о состоянии параметров после передачи какой-либо данной функции, даже не видя определения этой функции.

```

// function_parameter.h

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
// to non_qualified_function_parameter(). If passed to one_const_one_not(), it is the first
// parameter.
void const_function_parameter(const ConstMemberFunctions& c);

// We can assume that c is modified and/or c.set_val() is called, and may or may not be passed
// to any of these functions. If passed to one_const_one_not, it may be either parameter.
void non_qualified_function_parameter(ConstMemberFunctions& c);

```

```

// We can assume that:
// l is not modified, and l.set_val() won't be called.
// l may or may not be passed to const_function_parameter().
// r is modified, and/or r.set_val() may be called.
// r may or may not be passed to either of the preceding functions.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r);

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
// to non_qualified_function_parameter(). If passed to one_const_one_not(), it is the first
// parameter.
void bad_parameter(const ConstMemberFunctions& c);

```

Из-за правил `const` эти предположения будут фактически выполняться компилятором.

```

// function_parameter.cpp

// Our assumption was correct.
void const_function_parameter(const ConstMemberFunctions& c) {
    std::cout << "With the current value, the output is: " << c.calc() << '\n'
               << "If squared, it's: " << c.squared_calc()
               << std::endl;
}

// Our assumption was correct.
void non_qualified_function_parameter(ConstMemberFunctions& c) {
    c.set_val(42);
    std::cout << "For the value 42, the output is: " << c.calc() << '\n'
               << "If squared, it's: " << c.squared_calc()
               << std::endl;
}

// Our assumption was correct, in the ugliest possible way.
// Note that const correctness doesn't prevent encapsulation from intentionally being broken,
// it merely prevents code from having write access when it doesn't need it.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r) {
    // Let's just punch access modifiers and common sense in the face here.
    struct Machiavelli {
        int val;
        int unimportant;
        bool state_changed;
    };
    reinterpret_cast<Machiavelli&>(r).val = l.calc();
    reinterpret_cast<Machiavelli&>(r).state_changed = true;

    const_function_parameter(l);
    const_function_parameter(r);
}

// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers in c.set_val().
void bad_parameter(const ConstMemberFunctions& c) {
    c.set_val(18);
}

```

В то время как можно **обойти** `const` **правильность**, и расширением разрывать эти гарантии, то это должно быть сделано намеренно программистом (так же, как нарушение

инкапсуляции с `Machiavelli` , выше), и, вероятно, вызовут неопределенное поведение.

```
class DealBreaker : public ConstMemberFunctions {
public:
    DealBreaker(int v = 0);

    // A foreboding name, but it's const...
    void no_guarantees() const;
}

DealBreaker::DealBreaker(int v /* = 0 */) : ConstMemberFunctions(v) {}

// Our assumption was incorrect.
// const_cast removes const-ness, making the compiler think we know what we're doing.
void DealBreaker::no_guarantees() const {
    const_cast<DealBreaker*>(this)->set_val(823);
}

// ...

const DealBreaker d(50);
d.no_guarantees(); // Undefined behaviour: d really IS const, it may or may not be modified.
```

Однако, в связи с этим требует от программиста очень конкретно *сказать* компилятору , что они намерены игнорировать `const` ность, и быть согласованы между составителями, как правило , можно предположить , что `const` правильный код будет воздерживаться от этого , если не указано иное.

Прочитайте Const Correctness онлайн: <https://riptutorial.com/ru/cplusplus/topic/7217/const-correctness>

глава 4: constexpr

Вступление

`constexpr` - это **ключевое слово**, которое может использоваться для обозначения значения переменной как постоянного выражения, функции, потенциально пригодной для использования в постоянных выражениях, или (поскольку C ++ 17) оператор `if` имеет только одну из своих ветвей, выбранных для компиляции.

замечания

`constexpr` слово `constexpr` было добавлено в C ++ 11, но в течение нескольких лет после публикации стандарта C ++ 11 не все основные компиляторы поддерживали его. в то время, когда был опубликован стандарт C ++ 11. На момент публикации C ++ 14 все основные компиляторы поддерживают `constexpr`.

Examples

переменные constexpr

Переменная, объявленная `constexpr`, неявно `const` и ее значение может использоваться как постоянное выражение.

Сравнение с `#define`

`constexpr` - это безопасная замена для выражений времени компиляции на основе `#define`. С `constexpr` выражение с `constexpr` времени компиляции заменяется результатом. Например:

C ++ 11

```
int main()
{
    constexpr int N = 10 + 2;
    cout << N;
}
```

выдает следующий код:

```
cout << 12;
```

Макрос компиляции, основанный на предварительном процессоре, будет отличаться. Рассматривать:

```
#define N 10 + 2
```

```
int main()
{
    cout << N;
}
```

будет производить:

```
cout << 10 + 2;
```

который, очевидно, будет преобразован в `cout << 10 + 2;`, Однако компилятору придется больше работать. Кроме того, это создает проблему, если не используется правильно.

Например (с `#define`):

```
cout << N * 2;
```

формы:

```
cout << 10 + 2 * 2; // 14
```

Но предварительно оцененный `constexpr` правильно дал бы 24 .

Сравнение с `const`

`const` переменная - это **переменная**, для которой требуется память для хранения. `constexpr` не делает. `constexpr` создает `constexpr` времени компиляции, которая не может быть изменена. Вы можете утверждать, что `const` также не может быть изменен. Но учтите:

```
int main()
{
    const int size1 = 10;
    const int size2 = abs(10);

    int arr_one[size1];
    int arr_two[size2];
}
```

С большинством компиляторов второй оператор не работает (может работать с GCC, например). Размер любого массива, как вы знаете, должен быть постоянным выражением (т. Е. Результатом значения времени компиляции). Второй переменной `size2` присваивается определенное значение, которое определяется во время выполнения (даже если вы знаете, что это 10 , для компилятора это не время компиляции).

Это означает , что `const` может или не может быть истинной константой времени компиляции. Вы не можете гарантировать или обеспечить, чтобы конкретное значение `const` абсолютно компилируемым. Вы можете использовать `#define` но у него есть свои подводные камни.

Поэтому просто используйте:

C ++ 11

```
int main()
{
    constexpr int size = 10;

    int arr[size];
}
```

Выражение `constexpr` должно оцениваться до значения времени компиляции. Таким образом, вы не можете использовать:

C ++ 11

```
constexpr int size = abs(10);
```

Если функция (`abs`) сама возвращает `constexpr` .

Все базовые типы могут быть инициализированы с помощью `constexpr` .

C ++ 11

```
constexpr bool FailFatal = true;
constexpr float PI = 3.14f;
constexpr char* site= "StackOverflow";
```

Интересно, что и удобно, вы также можете использовать `auto` :

C ++ 11

```
constexpr auto domain = ".COM"; // const char * const domain = ".COM"
constexpr auto PI = 3.14;       // constexpr double
```

функции `constexpr`

Функция, объявленная как `constexpr` является неявно встроенной, и `constexpr` такой функции потенциально дает постоянные выражения. Например, следующая функция, если вызывается с постоянными аргументами выражения, также дает константное выражение:

C ++ 11

```
constexpr int Sum(int a, int b)
{
    return a + b;
}
```

Таким образом, результат вызова функции может использоваться как `constexpr` массива или аргумент шаблона или инициализировать переменную `constexpr` :

C ++ 11

```
int main()
{
    constexpr int S = Sum(10,20);

    int Array[S];
    int Array2[Sum(20,30)]; // 50 array size, compile time
}
```

Обратите внимание: если вы удаляете `constexpr` из спецификации типа возвращаемого значения, присваивание `s` не будет работать, так как `s` является `constexpr` и ей должна быть назначена `constexpr` времени компиляции. Аналогично, размер массива также не будет константным выражением, если функция `Sum` не `constexpr`.

Интересная вещь о `constexpr` заключается в том, что вы можете использовать ее как обычные функции:

C ++ 11

```
int a = 20;
auto sum = Sum(a, abs(-20));
```

`Sum` теперь не будет функцией `constexpr`, она будет скомпилирована как обычная функция, принимая переменные (непостоянные) аргументы и возвращая непостоянное значение. Вам не нужно писать две функции.

Это также означает, что если вы попытаетесь назначить такой вызов неконстантной переменной, он не будет компилироваться:

C ++ 11

```
int a = 20;
constexpr auto sum = Sum(a, abs(-20));
```

Причина проста: `constexpr` должна `constexpr` только константа времени компиляции. Однако вышеупомянутый вызов функции делает `Sum` не `constexpr` (значение `R` не является `constexpr`, но `L`-значение объявляет себя `constexpr`).

Функция `constexpr` также **должна** возвращать константу времени компиляции. Следующее не будет компилироваться:

C ++ 11

```
constexpr int Sum(int a, int b)
{
    int a1 = a;    // ERROR
    return a + b;
}
```


Поскольку `a1` является *переменной*, отличной от `constexpr`, и запрещает функции быть истинной функцией `constexpr`. Что делает его `constexpr` и присвоение ему также не будет работать - так как значение `a` (входящего параметра) до сих пор еще не известно:

C ++ 11

```
constexpr int Sum(int a, int b)
{
    constexpr int a1 = a;    // ERROR
    ..
```

Кроме того, следующее не будет компилироваться:

C ++ 11

```
constexpr int Sum(int a, int b)
{
    return abs(a) + b; // or abs(a) + abs(b)
}
```

Поскольку `abs(a)` не является постоянным выражением (даже `abs(10)` не будет работать, так как `abs` не возвращает `constexpr int` !

Как насчет этого?

C ++ 11

```
constexpr int Abs(int v)
{
    return v >= 0 ? v : -v;
}

constexpr int Sum(int a, int b)
{
    return Abs(a) + b;
}
```

Мы создали собственную функцию `Abs` которая является `constexpr`, а тело `Abs` также не нарушает никакого правила. Кроме того, на сайте вызова (внутри `Sum`) выражение вычисляется как `constexpr`. Следовательно, вызов `Sum(-10, 20)` будет выражением постоянной времени компиляции, результатом которого будет `30`.

Статический оператор if

C ++ 17

Оператор `if constexpr` может использоваться для условной компиляции кода. Условие должно быть постоянным выражением. Отключенная ветвь не *отбрасывается*. Отбрасываемый оператор внутри шаблона не создается. Например:

```
template<class T, class ... Rest>
void g(T &&p, Rest &&...rs)
{
    // ... handle p
    if constexpr (sizeof...(rs) > 0)
        g(rs...); // never instantiated with an empty argument list
}
```

Кроме того, переменные и функции, которые используются odr только внутри отброшенных операторов, не требуются для определения, а отбрасываемые операторы `return` не используются для вычитания типа функции.

`if constexpr` отличается от `#ifdef`. `#ifdef` условно компилирует код, но только на основе условий, которые могут быть оценены во время предварительной обработки. Например, `#ifdef` не может использоваться для условной компиляции кода в зависимости от значения параметра шаблона. С другой стороны, `if constexpr` не может использоваться для отбрасывания синтаксически недействительного кода, в то время как `#ifdef` может.

```
if constexpr(false) {
    foobar; // error; foobar has not been declared
    std::vector<int> v("hello, world"); // error; no matching constructor
}
```

Прочитайте `constexpr` онлайн: <https://riptutorial.com/ru/cplusplus/topic/3899/constexpr>

глава 5: decltype

Вступление

Ключевое слово `decltype` может использоваться для получения типа переменной, функции или выражения.

Examples

Основной пример

Этот пример просто иллюстрирует, как можно использовать это ключевое слово.

```
int a = 10;

// Assume that type of variable 'a' is not known here, or it may
// be changed by programmer (from int to long long, for example).
// Hence we declare another variable, 'b' of the same type using
// decltype keyword.
decltype(a) b; // 'decltype(a)' evaluates to 'int'
```

Если, например, кто-то изменяет тип 'a':

```
float a=99.0f;
```

Тогда тип переменной `b` теперь автоматически становится `float`.

Другой пример

Скажем, у нас есть вектор:

```
std::vector<int> intVector;
```

И мы хотим объявить итератор для этого вектора. Очевидная идея - использовать `auto`. Однако может потребоваться просто объявить переменную итератора (и не назначать ее чему-либо). Мы бы сделали:

```
vector<int>::iterator iter;
```

Тем не менее, с помощью `decltype` становится легче и меньше подвержено ошибкам (если изменяется тип `intVector`).

```
decltype(intVector)::iterator iter;
```

В качестве альтернативы:

```
decltype(intVector.begin()) iter;
```

Во втором примере возвращаемый тип `begin` используется для определения фактического типа, который является `vector<int>::iterator`.

Если нам нужен `const_iterator`, нам просто нужно использовать `cbegin`:

```
decltype(intVector.cbegin()) iter; // vector<int>::const_iterator
```

Прочитайте `decltype` онлайн: <https://riptutorial.com/ru/cplusplus/topic/9930/decltype>

глава 6: Loops

Вступление

Оператор цикла выполняет группу операторов несколько раз, пока не будет выполнено условие. В C++ существует три типа примитивных циклов: `for`, `while` и `do ... while`.

Синтаксис

- в то время как (*условие*) оператор;
- `do statement while ( выражение );`
- `for ( for-init-statement ; condition ; expression ) statement ;`
- `for ( for-range-declaration : for-range-initializer ) оператор ;`
- перерыв ;
- Продолжить ;

замечания

`algorithm` вызова обычно предпочтительнее для ручных петель.

Если вы хотите что-то, что уже делает алгоритм (или что-то очень похожее), вызов алгоритма более ясный, часто более эффективный и менее подверженный ошибкам.

Если вам нужен цикл, который делает что-то довольно простое (но для использования алгоритма требуется путаница путаницы связующих и адаптеров), просто напишите цикл.

Examples

Диапазон для

C++ 11

`for` циклов можно использовать для итерации по элементам диапазона, основанного на итераторе, без использования числового индекса или прямого доступа к итераторам:

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(auto val: v)
{
    std::cout << val << " ";
}

std::cout << std::endl;
```

Это будет выполнять итерацию по каждому элементу из `v`, причем `val` получает значение текущего элемента. Следующее утверждение:

```
for (for-range-declaration : for-range-initializer ) statement
```

ЭКВИВАЛЕНТНО:

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr, __end = end-expr;
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

C ++ 17

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr;
    auto __end = end-expr; // end is allowed to be a different type than begin in C++17
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

Это изменение было введено для запланированной поддержки Ranges TS в C ++ 20.

В этом случае наш цикл эквивалентен:

```
{
    auto&& __range = v;
    auto __begin = v.begin(), __end = v.end();
    for (; __begin != __end; ++__begin) {
        auto val = *__begin;
        std::cout << val << " ";
    }
}
```

Обратите внимание, что `auto val` объявляет тип значения, который будет копией значения, хранящегося в диапазоне (мы инициализируем его инициализацию с помощью итератора по мере продвижения). Если значения, хранящиеся в диапазоне, дороги для копирования, вы можете использовать `const auto &val`. Вы также не обязаны использовать `auto`; вы можете использовать соответствующее имя типа, если оно неявно конвертируется из типа значения диапазона.

Если вам нужен доступ к итератору, диапазон на основе не может помочь вам (не без каких-либо усилий, по крайней мере).

Если вы хотите сослаться на него, вы можете сделать это:

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(float &val: v)
{
    std::cout << val << " ";
}
```

Вы можете итерации по ссылке `const` если у вас есть контейнер `const` :

```
const vector<float> v = {0.4f, 12.5f, 16.234f};

for(const float &val: v)
{
    std::cout << val << " ";
}
```

Можно использовать пересылку ссылок, когда итератор последовательности возвращает объект-прокси, и вам нужно работать с этим объектом `const` способом. Примечание: это, скорее всего, путает читателей вашего кода.

```
vector<bool> v(10);

for(auto&& val: v)
{
    val = true;
}
```

Типа «Диапазон» предоставляется на основе диапазона `for` может быть одним из следующих:

- Языковые массивы:

```
float arr[] = {0.4f, 12.5f, 16.234f};

for(auto val: arr)
{
    std::cout << val << " ";
}
```

Обратите внимание, что выделение динамического массива не учитывается:

```
float *arr = new float[3]{0.4f, 12.5f, 16.234f};

for(auto val: arr) //Compile error.
{
    std::cout << val << " ";
}
```

- Любой тип, который имеет функции-члены `begin()` и `end()` , возвращает итераторы к элементам типа. Стандартные библиотечные контейнеры подходят, но могут использоваться и пользовательские типы:

```

struct Rng
{
    float arr[3];

    // pointers are iterators
    const float* begin() const {return &arr[0];}
    const float* end() const   {return &arr[3];}
    float* begin() {return &arr[0];}
    float* end()   {return &arr[3];}
};

int main()
{
    Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}

```

- Любой тип, который имеет функции non-member `begin(type)` и `end(type)` которые могут быть найдены через зависимый от аргументов поиск по `type`. Это полезно для создания типа диапазона без необходимости изменять тип класса:

```

namespace Mine
{
    struct Rng {float arr[3];};

    // pointers are iterators
    const float* begin(const Rng &rng) {return &rng.arr[0];}
    const float* end(const Rng &rng) {return &rng.arr[3];}
    float* begin(Rng &rng) {return &rng.arr[0];}
    float* end(Rng &rng) {return &rng.arr[3];}
}

int main()
{
    Mine::Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}

```

Для цикла

Цикл `for` выполняет инструкции в `loop body`, а `condition` цикла - `true`. Прежде, чем `initialization statement` цикла выполняется ровно один раз. После каждого цикла выполняется исполняющая часть `iteration execution`.

Цикл `for` определяется следующим образом:

```

for (/*initialization statement*/; /*condition*/; /*iteration execution*/)

```



```
{
    // body of the loop
}
```

Объяснение заявлений заполнителя:

- `initialization statement` : этот оператор запускается только один раз, в начале цикла `for` . Вы можете ввести объявление нескольких переменных одного типа, например `int i = 0, a = 2, b = 3` . Эти переменные действительны только в области цикла. Переменные, определенные перед циклом с тем же именем, скрываются во время выполнения цикла.
- `condition` : Этот оператор оценивается перед выполнением каждого *цикла цикла* и прерывает цикл, если он оценивает значение `false` .
- `iteration execution` : этот оператор запускается после *тела* цикла, перед оценкой следующего *условия* , если цикл `for` не прерван в *теле* (по `break` , `goto` , `return` или исключаемому исключению). Вы можете ввести несколько операторов в части `iteration execution` , например, `a++` , `b+=10` , `c=b+a` .

Грубый эквивалент `for` цикла, переписан как в `while` петли:

```
/*initialization*/
while (/*condition*/)
{
    // body of the loop; using 'continue' will skip to increment part below
    /*iteration execution*/
}
```

Наиболее распространенным случаем использования цикла `for` является выполнение операторов определенное количество раз. Например, рассмотрим следующее:

```
for(int i = 0; i < 10; i++) {
    std::cout << i << std::endl;
}
```

Правильный цикл также:

```
for(int a = 0, b = 10, c = 20; (a+b+c < 100); c--, b++, a+=c) {
    std::cout << a << " " << b << " " << c << std::endl;
}
```

Пример скрытия объявленных переменных перед циклом:

```
int i = 99; //i = 99
for(int i = 0; i < 10; i++) { //we declare a new variable i
    //some operations, the value of i ranges from 0 to 9 during loop execution
}
//after the loop is executed, we can access i with value of 99
```

Но если вы хотите использовать уже объявленную переменную и не скрывать ее, то

опустите часть объявления:

```
int i = 99; //i = 99
for(i = 0; i < 10; i++) { //we are using already declared variable i
    //some operations, the value of i ranges from 0 to 9 during loop execution
}
//after the loop is executed, we can access i with value of 10
```

Заметки:

- Операторы инициализации и приращения могут выполнять операции, не связанные с оператором условия, или вообще ничего - если вы этого хотите. Но по соображениям удобочитаемости лучше всего выполнять операции, непосредственно относящиеся к циклу.
- Переменная, объявленная в инструкции инициализации, видна только внутри области цикла `for` и освобождается после завершения цикла.
- Не забывайте, что переменная, которая была объявлена в `initialization statement` может быть изменена во время цикла, а также переменная, отмеченная в `condition`.

Пример цикла, который рассчитывается от 0 до 10:

```
for (int counter = 0; counter <= 10; ++counter)
{
    std::cout << counter << '\n';
}
// counter is not accessible here (had value 11 at the end)
```

Объяснение фрагментов кода:

- `int counter = 0` инициализирует `counter` переменных равным 0. (Эта переменная может использоваться только внутри цикла `for`.)
- `counter <= 10` - это логическое условие, которое проверяет, равен ли `counter` меньше или равен 10. Если это `true`, цикл выполняется. Если он `false`, цикл завершается.
- `++counter` - это операция приращения, которая увеличивает значение `counter` на 1 перед следующей проверкой состояния.

Если оставить все утверждения пустыми, вы можете создать бесконечный цикл:

```
// infinite loop
for (;;)
    std::cout << "Never ending!\n";
```

В `while` петля эквивалент выше:

```
// infinite loop
while (true)
    std::cout << "Never ending!\n";
```

Тем не менее, бесконечный цикл все равно может быть оставлен при использовании операторов `break` , `goto` или `return` или путем исключения исключения.

Следующий общий пример итерации по всем элементам из коллекции STL (например, `vector`) без использования заголовка `<algorithm>` :

```
std::vector<std::string> names = {"Albert Einstein", "Stephen Hawking", "Michael Ellis"};
for(std::vector<std::string>::iterator it = names.begin(); it != names.end(); ++it) {
    std::cout << *it << std::endl;
}
```

Пока цикл

В `while` цикл выполняет операторы , пока данное условие не станет `false` . Этот оператор управления используется, когда заранее неизвестно, сколько раз должен выполняться блок кода.

Например, чтобы напечатать все числа от 0 до 9, можно использовать следующий код:

```
int i = 0;
while (i < 10)
{
    std::cout << i << " ";
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; "0 1 2 3 4 5 6 7 8 9" is printed to the console
```

C ++ 17

Обратите внимание, что поскольку C ++ 17, первые 2 оператора могут быть объединены

```
while (int i = 0; i < 10)
//... The rest is the same
```

Чтобы создать бесконечный цикл, можно использовать следующую конструкцию:

```
while (true)
{
    // Do something forever (however, you can exit the loop by calling 'break'
}
```

Существует еще один вариант циклов `while` , а именно `do...while` construct. Дополнительную информацию см. В [примере цикла while-while](#) .

Объявление переменных в условиях

В состоянии циклов `for` и `while` также разрешено объявлять объект. Этот объект будет рассматриваться как область действия до конца цикла и будет сохраняться через каждую итерацию цикла:

```

for (int i = 0; i < 5; ++i) {
    do_something(i);
}
// i is no longer in scope.

for (auto& a : some_container) {
    a.do_something();
}
// a is no longer in scope.

while(std::shared_ptr<Object> p = get_object()) {
    p->do_something();
}
// p is no longer in scope.

```

Однако не разрешается делать то же самое с `do...while` loop; вместо этого объявляйте переменную перед циклом и (необязательно) вставляйте как переменную, так и цикл в локальную область, если вы хотите, чтобы переменная выходила из области действия после завершения цикла:

```

//This doesn't compile
do {
    s = do_something();
} while (short s > 0);

// Good
short s;
do {
    s = do_something();
} while (s > 0);

```

Это связано с тем, что часть *оператора* цикла `do...while` `while` (тело цикла) оценивается до того, как часть *выражения* (`while`) достигнута, и, таким образом, любое объявление в *выражении* не будет видимым во время первой итерации петли.

Цикл Do-while

Цикл *Do-to время* очень похоже на *время* цикла, за исключением того, что условие проверяется в конце каждого цикла, а не в начале. Поэтому цикл гарантированно выполняется хотя бы один раз.

Следующий код будет печатать `0` , поскольку условие будет оцениваться как `false` в конце первой итерации:

```

int i =0;
do
{
    std::cout << i;
    ++i; // Increment counter
}
while (i < 0);
std::cout << std::endl; // End of line; 0 is printed to the console

```

Примечание: не забывайте точку с запятой в конце `while(condition);` , который необходим в конструкции `do-while` .

В отличие от цикла `do- while`, следующее не будет печатать ничего, потому что условие принимает значение `false` в начале первой итерации:

```
int i =0;
while (i < 0)
{
    std::cout << i;
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; nothing is printed to the console
```

Примечание: А в то время как цикл может быть прерван без условия становится ложным, используя `break` , `goto` , или `return` заявление.

```
int i = 0;
do
{
    std::cout << i;
    ++i; // Increment counter
    if (i > 5)
    {
        break;
    }
}
while (true);
std::cout << std::endl; // End of line; 0 1 2 3 4 5 is printed to the console
```

Тривиальный цикл `do- while` также иногда используется для записи макросов, для которых требуется их собственная область (в этом случае конечная точка с запятой опускается из определения макроса и должна быть предоставлена пользователем):

```
#define BAD_MACRO(x) f1(x); f2(x); f3(x);

// Only the call to f1 is protected by the condition here
if (cond) BAD_MACRO(var);

#define GOOD_MACRO(x) do { f1(x); f2(x); f3(x); } while(0)

// All calls are protected here
if (cond) GOOD_MACRO(var);
```

Записи цикла Loop: Break and Continue

Операторы управления циклом используются для изменения потока выполнения из его обычной последовательности. Когда выполнение оставляет область, все автоматические объекты, созданные в этой области, уничтожаются. `break` и `continue` - это операторы управления контуром.

Инструкция `break` прекращает цикл без какого-либо дальнейшего рассмотрения.

```
for (int i = 0; i < 10; i++)
{
    if (i == 4)
        break; // this will immediately exit our loop
    std::cout << i << '\n';
}
```

Вышеприведенный код будет распечатан:

```
1
2
3
```

Оператор `continue` не сразу выходит из цикла, а пропускает остальную часть тела цикла и переходит в начало цикла (включая проверку состояния).

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 == 0) // evaluates to true if i is even
        continue; // this will immediately go back to the start of the loop
    /* the next line will only be reached if the above "continue" statement
       does not execute */
    std::cout << i << " is an odd number\n";
}
```

Вышеприведенный код будет распечатан:

```
1 is an odd number
3 is an odd number
5 is an odd number
```

Поскольку такие изменения потока управления иногда трудны для людей, которые легко понимают, `break` и `continue` использоваться экономно. Более простую реализацию, как правило, легче читать и понимать. Например, первый цикл `for` с `break` выше может быть переписан как:

```
for (int i = 0; i < 4; i++)
{
    std::cout << i << '\n';
}
```

Второй пример с `continue` может быть переписан как:

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 != 0) {
        std::cout << i << " is an odd number\n";
    }
}
```

Диапазон - для суб-диапазона

Используя циклы базы значений, вы можете перебрать часть под конкретного контейнера или другого диапазона, создав прокси-объект, который подходит для циклов, основанных на диапазонах.

```
template<class Iterator, class Sentinel=Iterator>
struct range_t {
    Iterator b;
    Sentinel e;
    Iterator begin() const { return b; }
    Sentinel end() const { return e; }
    bool empty() const { return begin()==end(); }
    range_t without_front( std::size_t count=1 ) const {
        if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
            count = (std::min)(std::size_t(std::distance(b,e)), count);
        }
        return {std::next(b, count), e};
    }
    range_t without_back( std::size_t count=1 ) const {
        if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
            count = (std::min)(std::size_t(std::distance(b,e)), count);
        }
        return {b, std::prev(e, count)};
    }
};

template<class Iterator, class Sentinel>
range_t<Iterator, Sentinel> range( Iterator b, Sentinel e ) {
    return {b,e};
}

template<class Iterable>
auto range( Iterable& r ) {
    using std::begin; using std::end;
    return range(begin(r),end(r));
}

template<class C>
auto except_first( C& c ) {
    auto r = range(c);
    if (r.empty()) return r;
    return r.without_front();
}
```

теперь мы можем сделать:

```
std::vector<int> v = {1,2,3,4};

for (auto i : except_first(v))
    std::cout << i << '\n';
```

и распечатать

```
2
3
4
```

Имейте в виду, что промежуточные объекты, сгенерированные в части `for (:range_expression)` цикла `for` будут истекли к моменту начала цикла `for` .

Прочитайте Loops онлайн: <https://riptutorial.com/ru/cplusplus/topic/589/loops>

глава 7: Perfect Forwarding

замечания

Для идеальной пересылки требуется *пересылка ссылок*, чтобы сохранить ref-квалификаторы аргументов. Такие ссылки появляются только в *выведенном контексте*. То есть:

```
template<class T>
void f(T&& x) // x is a forwarding reference, because T is deduced from a call to f()
{
    g(std::forward<T>(x)); // g() will receive an lvalue or an rvalue, depending on x
}
```

Следующее не связано с совершенной пересылкой, потому что `T` не выводится из вызова конструктора:

```
template<class T>
struct a
{
    a(T&& x); // x is a rvalue reference, not a forwarding reference
};
```

C ++ 17

C ++ 17 позволит вычесть аргументы шаблона класса. Конструктор «а» в приведенном выше примере станет пользователем ссылки пересылки

```
a example1(1);
// same as a<int> example1(1);

int x = 1;
a example2(x);
// same as a<int&> example2(x);
```

Examples

Заводские функции

Предположим, мы хотим написать фабричную функцию, которая принимает произвольный список аргументов и передает эти аргументы без изменений в другую функцию. Примером такой функции является `make_unique`, которая используется для безопасного создания нового экземпляра `T` и возврата `unique_ptr<T>` которому принадлежит экземпляр.

Языковые правила, касающиеся вариационных шаблонов и ссылок на `rvalue`, позволяют нам написать такую функцию.

```
template<class T, class... A>
unique_ptr<T> make_unique(A&&... args)
{
    return unique_ptr<T>(new T(std::forward<A>(args)...));
}
```

Использование эллипсов ... указывает пакет параметров, который представляет собой произвольное количество типов. Компилятор расширит этот пакет параметров до правильного количества аргументов на сайте вызова. Эти аргументы затем передаются в конструктор `T`, используя `std::forward`. Эта функция необходима для сохранения ref-квалификаторов аргументов.

```
struct foo
{
    foo() {}
    foo(const foo&) {} // copy constructor
    foo(foo&&) {} // copy constructor
    foo(int, int, int) {}
};

foo f;
auto p1 = make_unique<foo>(f); // calls foo::foo(const foo&)
auto p2 = make_unique<foo>(std::move(f)); // calls foo::foo(foo&&)
auto p3 = make_unique<foo>(1, 2, 3);
```

Прочитайте Perfect Forwarding онлайн: <https://riptutorial.com/ru/cplusplus/topic/1750/perfect-forwarding>

глава 8: RAII: Приобретение ресурсов является инициализацией

замечания

RAII означает **R**esource **A**cquisition **I**s **I**nitialization. Также иногда упоминается как SBRM (управление ресурсами на основе Scope) или RRID (Release Release Is Destruction), RAII - это идиома, используемая для привязки ресурсов к времени жизни объекта. В C++ деструктор для объекта всегда выполняется, когда объект выходит из области видимости - мы можем воспользоваться этим, чтобы связать очистку ресурсов с уничтожением объектов.

Каждый раз, когда вам нужно приобрести некоторый ресурс (например, блокировку, дескриптор файла, выделенный буфер), который вам в конечном итоге нужно будет выпускать, вам следует рассмотреть возможность использования объекта для управления этим управлением ресурсами. Развертывание стека будет происходить независимо от исключения или выхода на ранней стадии, поэтому объект обработчика ресурсов очистит ресурс для вас, не задумываясь о всех возможных текущих и будущих кодах.

Стоит отметить, что RAII не полностью освобождает разработчика от мысли о ресурсе жизни. Один случай - это, очевидно, вызов `crash` или `exit()`, который предотвратит вызов деструкторов. Поскольку ОС будет очищать локальные ресурсы процесса, такие как память, после завершения процесса, это не проблема в большинстве случаев. Однако с системными ресурсами (например, именованными каналами, файлами блокировки, разделяемой памятью) вам по-прежнему нужны средства для решения ситуации, когда процесс не очищался после себя, то есть при запуске теста, если файл блокировки существует, если это так, проверьте, действительно ли процесс с `pid` существует, а затем действуйте соответствующим образом.

Другая ситуация заключается в том, что процесс `unix` вызывает функцию из семейства `exec`, то есть после `fork-exec` для создания нового процесса. Здесь дочерний процесс будет иметь полную копию родительской памяти (включая объекты RAII), но после вызова `exec` ни один из деструкторов не будет вызван в этом процессе. С другой стороны, если процесс разветвляется и ни один из процессов не вызывает `exec`, все ресурсы очищаются в обоих процессах. Это правильно только для всех ресурсов, которые были фактически дублированы в `fork`, но с системными ресурсами оба процесса будут иметь только ссылку на ресурс (т. Е. Путь к файлу блокировки), и оба попытаются выпустить его отдельно, что потенциально может вызвать другой процесс терпит неудачу.

Examples

Блокировка

Плохая блокировка:

```
std::mutex mtx;

void bad_lock_example() {
    mtx.lock();
    try
    {
        foo();
        bar();
        if (baz()) {
            mtx.unlock();    // Have to unlock on each exit point.
            return;
        }
        quux();
        mtx.unlock();        // Normal unlock happens here.
    }
    catch(...) {
        mtx.unlock();        // Must also force unlock in the presence of
        throw;               // exceptions and allow the exception to continue.
    }
}
```

Это неправильный способ реализовать блокировку и разблокировку мьютекса. Для обеспечения правильной разблокировки мьютекса с помощью `unlock()` требуется, чтобы программатор удостоверился, что все потоки, приводящие к выходу из функции, приводят к вызову `unlock()`. Как показано выше, это хрупкие процессы, так как требуется, чтобы все сопровождающие продолжали следовать за рисунком вручную.

Используя подходящий класс для реализации RAII, проблема тривиальна:

```
std::mutex mtx;

void good_lock_example() {
    std::lock_guard<std::mutex> lk(mtx);    // constructor locks.
                                           // destructor unlocks. destructor call
                                           // guaranteed by language.

    foo();
    bar();
    if (baz()) {
        return;
    }
    quux();
}
```

`lock_guard` - чрезвычайно простой шаблон класса, который просто вызывает `lock()` в своем аргументе в своем конструкторе, сохраняет ссылку на аргумент и вызывает `unlock()` в аргументе в своем деструкторе. То есть, когда `lock_guard` выходит из области видимости, `mutex` гарантированно разблокируется. Не имеет значения, является ли причина, по которой он вышел из сферы действия, исключение или раннее возвращение - все дела обрабатываются; независимо от потока управления, мы гарантировали, что мы

разблокируемся правильно.

Наконец / ScopeExit

В случаях, когда мы не хотим писать специальные классы для обработки некоторого ресурса, мы можем написать общий класс:

```
template<typename Function>
class Finally final
{
public:
    explicit Finally(Function f) : f(std::move(f)) {}
    ~Finally() { f(); } // (1) See below

    Finally(const Finally&) = delete;
    Finally(Finally&&) = default;
    Finally& operator =(const Finally&) = delete;
    Finally& operator =(Finally&&) = delete;
private:
    Function f;
};
// Execute the function f when the returned object goes out of scope.
template<typename Function>
auto onExit(Function &&f) { return Finally<std::decay_t<Function>>{std::forward<Function>(f)};
}
```

И пример использования

```
void foo(std::vector<int>& v, int i)
{
    // ...

    v[i] += 42;
    auto autoRollBackChange = onExit([&]() { v[i] -= 42; });

    // ... code as recursive call `foo(v, i + 1)`
}
```

Примечание (1): Некоторое обсуждение определения деструктора должно рассматриваться как обращение к исключению:

- `~Finally() noexcept { f(); } : std::terminate` **вызывается в случае исключения**
- `~Finally() noexcept(noexcept(f())) { f(); } : terminate ()` **вызывается только в случае исключения во время разматывания пакетов.**
- `~Finally() noexcept { try { f(); } catch (...) { /* ignore exception (might log it) */ } }`
Нет `std::terminate`, но мы не можем обрабатывать ошибку (даже для разворачивания без стека).

ScopeSuccess (с ++ 17)

C ++ 17

Благодаря `int std::uncaught_exceptions()` мы можем реализовать действие, которое выполняется только при успешном завершении (исключение исключено из области). Ранее `bool std::uncaught_exception()` просто позволяет определить, запущена ли **какая-либо** переедресация стека.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeSuccess
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeSuccess(const F& f) : f(f) {}
    ScopeSuccess(const ScopeSuccess&) = delete;
    ScopeSuccess& operator =(const ScopeSuccess&) = delete;

    // f() might throw, as it can be caught normally.
    ~ScopeSuccess() noexcept(noexcept(f())) {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeSuccess logSuccess{[]() {std::cout << "Success 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeSuccess logSuccess{[]() {std::cout << "Success 2\n";}};

            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }
};

int main()
{
    try {
        Foo foo;

        throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

Выход:

ScopeFail (с ++ 17)

С ++ 17

Благодаря `int std::uncaught_exceptions()` мы можем реализовать действие, которое выполняется только при сбое (выбрано исключение в области). Ранее `bool std::uncaught_exception()` просто позволяет определить, запущена ли **какая-либо** переадресация стека.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeFail
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeFail(const F& f) : f(f) {}
    ScopeFail(const ScopeFail&) = delete;
    ScopeFail& operator =(const ScopeFail&) = delete;

    // f() should not throw, else std::terminate is called.
    ~ScopeFail() {
        if (uncaughtExceptionCount != std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeFail logFailure{[](){std::cout << "Fail 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeFail logFailure{[](){std::cout << "Failure 2\n";}};

            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }
};

int main()
{
    try {
```

```
    Foo foo;

    throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
} catch (...) { // std::uncaught_exceptions() == 0
}
}
```

Выход:

```
Failure 2
```

Прочитайте RAII: Приобретение ресурсов является инициализацией онлайн:
<https://riptutorial.com/ru/cplusplus/topic/1320/raii--приобретение-ресурсов-является-инициализацией>

глава 9: RTTI: информация о времени выполнения

Examples

Название типа

Вы можете получить определенное имя реализации типа во время выполнения, используя функцию `.name()` члена объекта `std::type_info` возвращаемого `typeid`.

```
#include <iostream>
#include <typeinfo>

int main()
{
    int speed = 110;

    std::cout << typeid(speed).name() << '\n';
}
```

Выход (определенный реализацией):

```
int
```

dynamic_cast

Используйте функцию `dynamic_cast<>()` как функцию, которая поможет вам сбрасывать иерархию наследования ([основное описание](#)).

Если вы должны выполнить некоторую непалиморфную работу над некоторыми производными классами `B` и `C`, но получили базовый `class A`, напишите так:

```
class A { public: virtual ~A(){} };

class B: public A
{ public: void work4B(){} };

class C: public A
{ public: void work4C(){} };

void non_polymorphic_work(A* ap)
{
    if (B* bp =dynamic_cast<B*>(ap))
        bp->work4B();
    if (C* cp =dynamic_cast<C*>(ap))
        cp->work4C();
}
```

Ключевое слово typeid

`typeid` **слово** `typeid` является унарным оператором, который дает информацию типа времени выполнения о ее операнде, если тип операнда является полиморфным типом класса. Он возвращает lvalue типа `const std::type_info`. Квалификация верхнего уровня игнорируется.

```
struct Base {
    virtual ~Base() = default;
};
struct Derived : Base {};
Base* b = new Derived;
assert(typeid(*b) == typeid(Derived{})); // OK
```

`typeid` также может применяться к типу напрямую. В этом случае первые ссылки верхнего уровня удаляются, тогда верхняя степень cv-квалификации игнорируется. Таким образом, приведенный выше пример мог быть написан с помощью `typeid(Derived)` вместо `typeid(Derived{})`:

```
assert(typeid(*b) == typeid(Derived{})); // OK
```

Если `typeid` применяется к любому выражению, которое *не* относится к типу полиморфного класса, операнд не оценивается, а возвращаемая информация типа - для статического типа.

```
struct Base {
    // note: no virtual destructor
};
struct Derived : Base {};
Derived d;
Base& b = d;
assert(typeid(b) == typeid(Base)); // not Derived
assert(typeid(std::declval<Base>()) == typeid(Base)); // OK because unevaluated
```

Когда использовать, какой листинг в c ++

Используйте **`dynamic_cast`** для преобразования указателей / ссылок в иерархию наследования.

Используйте **`static_cast`** для конверсий обычного типа.

Используйте **`reinterpret_cast`** для низкоуровневого переинтерпретации битовых шаблонов. Используйте с особой осторожностью.

Используйте **`const_cast`** для отбрасывания `const` / `volatile`. Избегайте этого, если вы не застряли с использованием некорректного API.

Прочитайте RTTI: информация о времени выполнения онлайн:

<https://riptutorial.com/ru/cplusplus/topic/3129/rtti--информация-о-времени-выполнения>

глава 10: SFINAE (ошибка замены не является ошибкой)

Examples

enable_if

`std::enable_if` - удобная утилита для использования булевых условий для запуска SFINAE. Он определяется как:

```
template <bool Cond, typename Result=void>
struct enable_if { };

template <typename Result>
struct enable_if<true, Result> {
    using type = Result;
};
```

То есть, `enable_if<true, R>::type` является псевдонимом для `R`, тогда как `enable_if<true, R>::type` `enable_if<false, T>::type` неформатирован, так как эта специализация `enable_if` не имеет `type` члена типа.

`std::enable_if` может использоваться для ограничения шаблонов:

```
int negate(int i) { return -i; }

template <class F>
auto negate(F f) { return -f(); }
```

Здесь вызов `negate(1)` потерпел неудачу из-за двусмысленности. Но вторая перегрузка не предназначена для использования в интегральных типах, поэтому мы можем добавить:

```
int negate(int i) { return -i; }

template <class F, class = typename std::enable_if<!std::is_arithmetic<F>::value>::type>
auto negate(F f) { return -f(); }
```

Теперь создание экземпляра `negate<int>` приведет к `!std::is_arithmetic<int>::value` замены, так как `!std::is_arithmetic<int>::value` `false`. Из-за SFINAE это не является жесткой ошибкой, этот кандидат просто удаляется из набора перегрузки. В результате `negate(1)` имеет только одного жизнеспособного кандидата, который затем называется.

Когда его использовать

Следует иметь в виду, что `std::enable_if` является помощником *поверх* SFINAE, но это не то, что делает работу SFINAE в первую очередь. Давайте рассмотрим эти две альтернативы для реализации функциональности, аналогичной `std::size`, то есть `size(arg)` набора перегрузки `size(arg)` который создает размер контейнера или массива:

```
// for containers
template<typename Cont>
auto size1(Cont const& cont) -> decltype( cont.size() );

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size1(Elt const(&arr)[Size]);

// implementation omitted
template<typename Cont>
struct is_sizeable;

// for containers
template<typename Cont, std::enable_if_t<std::is_sizeable<Cont>::value, int> = 0>
auto size2(Cont const& cont);

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size2(Elt const(&arr)[Size]);
```

Предполагая, что `is_sizeable` написано надлежащим образом, эти два объявления должны быть в точности эквивалентными по отношению к SFINAE. Что проще всего написать, и что проще всего просмотреть и понять сразу?

Теперь давайте рассмотрим, как мы можем реализовать арифметические помощники, которые избегают целочисленного переполнения со знаком в сторону обтекания или модульного поведения. То есть, например, `incr(i, 3)` будет таким же, как `i += 3` за исключением того, что результат всегда будет определен, даже если `i` является `int` со значением `INT_MAX`. Это две возможные альтернативы:

```
// handle signed types
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(-1) < static_cast<Int>(0)]>;

// handle unsigned types by just doing target += amount
// since unsigned arithmetic already behaves as intended
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(0) < static_cast<Int>(-1)]>;

template<typename Int, std::enable_if_t<std::is_signed<Int>::value, int> = 0>
void incr2(Int& target, Int amount);

template<typename Int, std::enable_if_t<std::is_unsigned<Int>::value, int> = 0>
void incr2(Int& target, Int amount);
```

Еще раз, что проще всего написать, и который проще всего просматривать и понимать с первого взгляда?

Сила `std::enable_if` заключается в том, как она работает с рефакторингом и дизайном API. Если `is_sizeable<Cont>::value` означает, что `cont.size()` является допустимым, то просто выражение, как оно появляется для `size1` может быть более кратким, хотя это может зависеть от того, будет ли `is_sizeable` использоваться в нескольких местах или нет, Сравните это с `std::is_signed` который отражает его намерение гораздо более четко, чем когда его реализация просачивается в объявление `incr1`.

void_t

C++ 11

`void_t` - это мета-функция, которая отображает любое (число) типов для ввода `void`. Основной целью `void_t` является упрощение написания признаков типа.

`std::void_t` будет частью C++ 17, но до `std::void_t` пор это было чрезвычайно просто реализовать:

```
template <class...> using void_t = void;
```

Некоторые компиляторы [требуют](#) несколько иной реализации:

```
template <class...>
struct make_void { using type = void; };

template <typename... T>
using void_t = typename make_void<T...>::type;
```

Первичным применением `void_t` является запись типов типов, которые проверяют правильность утверждения. Например, давайте проверим, имеет ли тип функция-член `foo()` которая не принимает аргументов:

```
template <class T, class=void>
struct has_foo : std::false_type {};

template <class T>
struct has_foo<T, void_t<decltype(std::declval<T>().foo())>> : std::true_type {};
```

Как это работает? Когда я пытаюсь создать экземпляр `has_foo<T>::value`, это заставит компилятор попытаться найти лучшую специализацию для `has_foo<T, void>`. У нас есть два варианта: первичный и вторичный, который включает в себя создание экземпляра этого основного выражения:

- Если `T` имеет функцию член `foo()`, то любой тип, который возвращает конвертируется в `void`, и специализация предпочтительно первичный на основе шаблона правил частичного упорядочения. Таким образом `has_foo<T>::value` будет `true`
- Если `T` не имеет такой функции-члена (или для этого требуется более одного аргумента), то подстановка не выполняется для специализации, и у нас есть только

основной шаблон для возврата. Следовательно, `has_foo<T>::value false` .

Простейший случай:

```
template<class T, class=void>
struct can_reference : std::false_type {};

template<class T>
struct can_reference<T, std::void_t<T&>> : std::true_type {};
```

это не использует `std::declval` или `decltype` .

Вы можете заметить общую схему аргумента `void`. Мы можем это исключить:

```
struct details {
    template<template<class...>class Z, class=void, class...Ts>
    struct can_apply:
        std::false_type
    {};
    template<template<class...>class Z, class...Ts>
    struct can_apply<Z, std::void_t<Z<Ts...>>, Ts...>:
        std::true_type
    {};
};

template<template<class...>class Z, class...Ts>
using can_apply = details::can_apply<Z, void, Ts...>;
```

который скрывает использование `std::void_t` и делает `can_apply` действующим как индикатор того, является ли тип, предоставленный в качестве первого аргумента шаблона, хорошо сформированным после подстановки в него других типов. Предыдущие примеры теперь можно переписать с помощью `can_apply` как:

```
template<class T>
using ref_t = T&;

template<class T>
using can_reference = can_apply<ref_t, T>;    // Is T& well formed for T?
```

а также:

```
template<class T>
using dot_foo_r = decltype(std::declval<T&>().foo());

template<class T>
using can_dot_foo = can_apply<dot_foo_r, T>;    // Is T.foo() well formed for T?
```

который кажется более простым, чем исходные версии.

Есть предложения post-C ++ 17 для `std` характеристик, похожих на `can_apply` .

Утилита `void_t` была обнаружена Уолтером Брауном. Он дал замечательную [презентацию](#)

на нем в CppCon 2016.

trailing decltype в шаблонах функций

C ++ 11

Одной из ограничивающих функций является использование trailing `decltype` для указания типа возврата:

```
namespace details {
    using std::to_string;

    // this one is constrained on being able to call to_string(T)
    template <class T>
    auto convert_to_string(T const& val, int )
        -> decltype(to_string(val))
    {
        return to_string(val);
    }

    // this one is unconstrained, but less preferred due to the ellipsis argument
    template <class T>
    std::string convert_to_string(T const& val, ... )
    {
        std::ostringstream oss;
        oss << val;
        return oss.str();
    }
}

template <class T>
std::string convert_to_string(T const& val)
{
    return details::convert_to_string(val, 0);
}
```

Если я вызову `convert_to_string()` с аргументом, с которым я могу вызвать `to_string()` , то у меня есть две жизнеспособные функции для `details::convert_to_string()` . Первый предпочтительнее, поскольку преобразование от `0` до `int` является более лучшей неявной последовательностью преобразования, чем преобразование от `0` до `...` .

Если я вызываю `convert_to_string()` с аргументом, из которого я не могу вызывать `to_string()` , то создание первого экземпляра шаблона функции приводит к `decltype(to_string(val))` подстановки (отсутствует `decltype(to_string(val))`). В результате этот кандидат удаляется из набора перегрузки. Второй шаблон функции не ограничен, поэтому он выбирается, и мы вместо этого выполняем `operator<<(std::ostream&, T)` . Если этот параметр не определен, тогда у нас есть жесткая ошибка компиляции с стеком шаблонов на строке `oss << val` .

Что такое SFINAE

SFINAE означает **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror. Некорректный код, который

получается из подстановочных типов (или значений) для создания экземпляра шаблона функции или шаблона класса, **не** является сложной компиляционной ошибкой, а рассматривается только как отказ от вычета.

Ошибки дедукции при создании экземпляров шаблонов функций или специализации шаблонов классов удаляют этого кандидата из набора соображений - как будто этого неудачного кандидата не существует для начала.

```
template <class T>
auto begin(T& c) -> decltype(c.begin()) { return c.begin(); }

template <class T, size_t N>
T* begin(T (&arr)[N]) { return arr; }

int vals[10];
begin(vals); // OK. The first function template substitution fails because
              // vals.begin() is ill-formed. This is not an error! That function
              // is just removed from consideration as a viable overload candidate,
              // leaving us with the array overload.
```

Только неудачи замещения в **непосредственном контексте** считаются ошибками дедукции, все остальные считаются трудными ошибками.

```
template <class T>
void add_one(T& val) { val += 1; }

int i = 4;
add_one(i); // ok

std::string msg = "Hello";
add_one(msg); // error. msg += 1 is ill-formed for std::string, but this
              // failure is NOT in the immediate context of substituting T
```

enable_if_all / enable_if_any

C ++ 11

Мотивационный пример

Если в списке параметров шаблона имеется пакет вариативных шаблонов, как в следующем фрагменте кода:

```
template<typename ...Args> void func(Args &&...args) { //... };
```

Стандартная библиотека (до C ++ 17) не дает прямого способа написать **enable_if** для ограничения SFINAE для **всех параметров** в Args или **любых параметров** в Args . C ++ 17 предлагает `std::conjunction` и `std::disjunction` которые решают эту проблему. Например:

```
/// C++17: SFINAE constraints on all of the parameters in Args.
```



```
template<typename ...Args,
        std::enable_if_t<std::conjunction_v<custom_conditions_v<Args>...>>* = nullptr>
void func(Args &&...args) { //... };

/// C++17: SFINAE constraints on any of the parameters in Args.
template<typename ...Args,
        std::enable_if_t<std::disjunction_v<custom_conditions_v<Args>...>>* = nullptr>
void func(Args &&...args) { //... };
```

Если у вас нет доступных C++ 17, для их достижения есть несколько решений. Одним из них является использование класса базового класса и **частичных специализаций**, как показано в ответах на этот [вопрос](#).

В качестве альтернативы, можно также вручную реализовать поведение `std::conjunction` и `std::disjunction` в довольно прямом пути. В следующем примере я продемонстрирую реализации и объединим их с `std::enable_if` для создания двух псевдонимов: `enable_if_all` и `enable_if_any`, которые делают именно то, что они предполагают семантически. Это может обеспечить более масштабируемое решение.

Реализация `enable_if_all` и `enable_if_any`

Сначала давайте подражать `std::conjunction seq_and` и `std::disjunction seq_or` используя индивидуальные `seq_and` и `seq_or` соответственно:

```
/// Helper for prior to C++14.
template<bool B, class T, class F >
using conditional_t = typename std::conditional<B,T,F>::type;

/// Emulate C++17 std::conjunction.
template<bool...> struct seq_or: std::false_type {};
template<bool...> struct seq_and: std::true_type {};

template<bool B1, bool... Bs>
struct seq_or<B1,Bs...>:
    conditional_t<B1,std::true_type,seq_or<Bs...>> {};

template<bool B1, bool... Bs>
struct seq_and<B1,Bs...>:
    conditional_t<B1,seq_and<Bs...>,std::false_type> {};
```

Тогда реализация довольно прямолинейна:

```
template<bool... Bs>
using enable_if_any = std::enable_if<seq_or<Bs...>::value>;

template<bool... Bs>
using enable_if_all = std::enable_if<seq_and<Bs...>::value>;
```

В конце концов некоторые помощники:

```
template<bool... Bs>
using enable_if_any_t = typename enable_if_any<Bs...>::type;

template<bool... Bs>
using enable_if_all_t = typename enable_if_all<Bs...>::type;
```

использование

Использование также прямолинейно:

```
/// SFINAE constraints on all of the parameters in Args.
template<typename ...Args,
        enable_if_all_t<custom_conditions_v<Args>...>* = nullptr>
void func(Args &&...args) { //... };

/// SFINAE constraints on any of the parameters in Args.
template<typename ...Args,
        enable_if_any_t<custom_conditions_v<Args>...>* = nullptr>
void func(Args &&...args) { //... };
```

is_detected

Чтобы обобщить создание `type_trait`: на основе SFINAE обнаружены экспериментальные признаки `detected_or`, `detected_t`, `is_detected`.

С параметрами шаблона `typename Default`, `template <typename...> Op` и `typename ... Args`:

- `is_detected`: псевдоним `std::true_type` или `std::false_type` зависимости от действительности `Op<Args...>`
- `detected_t`: псевдоним `Op<Args...>` или `nonesuch` зависимости от действительности `Op<Args...>`.
- `detected_or`: псевдоним структуры с `value_t` который `is_detected`, и `type` который является `Op<Args...>` или `Default` зависимости от действительности `Op<Args...>`

который может быть реализован с использованием `std::void_t` для SFINAE следующим образом:

C ++ 17

```
namespace detail {
    template <class Default, class AlwaysVoid,
              template<class...> class Op, class... Args>
    struct detector
    {
        using value_t = std::false_type;
        using type = Default;
    };

    template <class Default, template<class...> class Op, class... Args>
    struct detector<Default, std::void_t<Op<Args...>>, Op, Args...>
    {
```

```

        using value_t = std::true_type;
        using type = Op<Args...>;
    };

} // namespace detail

// special type to indicate detection failure
struct nonesuch {
    nonesuch() = delete;
    ~nonesuch() = delete;
    nonesuch(nonesuch const&) = delete;
    void operator=(nonesuch const&) = delete;
};

template <template<class...> class Op, class... Args>
using is_detected =
    typename detail::detector<nonesuch, void, Op, Args...>::value_t;

template <template<class...> class Op, class... Args>
using detected_t = typename detail::detector<nonesuch, void, Op, Args...>::type;

template <class Default, template<class...> class Op, class... Args>
using detected_or = detail::detector<Default, void, Op, Args...>;

```

Затем можно просто реализовать черты для обнаружения присутствия метода:

```

typename <typename T, typename ...Ts>
using foo_type = decltype(std::declval<T>().foo(std::declval<Ts>()...));

struct C1 {};

struct C2 {
    int foo(char) const;
};

template <typename T>
using has_foo_char = is_detected<foo_type, T, char>;

static_assert(!has_foo_char<C1>::value, "Unexpected");
static_assert(has_foo_char<C2>::value, "Unexpected");

static_assert(std::is_same<int, detected_t<foo_type, C2, char>>::value,
    "Unexpected");

static_assert(std::is_same<void, // Default
    detected_or<void, foo_type, C1, char>>::value,
    "Unexpected");
static_assert(std::is_same<int, detected_or<void, foo_type, C2, char>>::value,
    "Unexpected");

```

Разрешение перегрузки с большим количеством опций

Если вам нужно выбрать один из нескольких вариантов, включение только одного с помощью `enable_if<>` может быть довольно громоздким, так как некоторые условия также должны быть сведены на нет.

Порядок между перегрузками может быть выбран с помощью наследования, т.е. отправки

меток.

Вместо того, чтобы тестировать вещи, которые должны быть хорошо сформированы, а также проверять отрицание всех других версий, мы вместо этого проверяем только то, что нам нужно, предпочтительно в `decltype` в трейлинг-возврате.

Это может привести к тому, что несколько вариантов будут хорошо сформированы, мы будем различать те, которые используют «теги», похожие на теги-метки-итераторы (`random_access_tag` и др.). Это работает, потому что прямое совпадение лучше, чем базовый класс, который лучше, чем базовый класс базового класса и т. Д.

```
#include <algorithm>
#include <iterator>

namespace detail
{
    // this gives us infinite types, that inherit from each other
    template<std::size_t N>
    struct pick : pick<N-1> {};
    template<>
    struct pick<0> {};

    // the overload we want to be preferred have a higher N in pick<N>
    // this is the first helper template function
    template<typename T>
    auto stable_sort(T& t, pick<2>)
        -> decltype( t.stable_sort(), void() )
    {
        // if the container have a member stable_sort, use that
        t.stable_sort();
    }

    // this helper will be second best match
    template<typename T>
    auto stable_sort(T& t, pick<1>)
        -> decltype( t.sort(), void() )
    {
        // if the container have a member sort, but no member stable_sort
        // it's customary that the sort member is stable
        t.sort();
    }

    // this helper will be picked last
    template<typename T>
    auto stable_sort(T& t, pick<0>)
        -> decltype( std::stable_sort(std::begin(t), std::end(t)), void() )
    {
        // the container have neither a member sort, nor member stable_sort
        std::stable_sort(std::begin(t), std::end(t));
    }
}

// this is the function the user calls. it will dispatch the call
// to the correct implementation with the help of 'tags'.
template<typename T>
void stable_sort(T& t)
{
    // use an N that is higher than any used above.
```

```
// this will pick the highest overload that is well formed.  
detail::stable_sort(t, detail::pick<10>{});  
}
```

Существуют и другие методы, которые обычно используются для различения перегрузок, таких как точное совпадение, лучше, чем преобразование, лучше, чем многоточие.

Тем не менее, отправка тегов может распространяться на любое количество вариантов, и это немного более ясно в намерениях.

Прочитайте SFINAE (ошибка замены не является ошибкой) онлайн:

<https://riptutorial.com/ru/cplusplus/topic/1169/sfinae--ошибка-замены-не-является-ошибкой->

глава 11: static_assert

Синтаксис

- `static_assert ( bool_constexpr , message )`
- `static_assert ( bool_constexpr ) / * Поскольку C ++ 17 * /`

параметры

параметр	подробности
<i>bool_constexpr</i>	Выражение для проверки
<i>сообщение</i>	Сообщение для печати, когда <i>bool_constexpr</i> является <i>ложным</i>

замечания

В отличие от [утверждений времени исполнения](#), статические утверждения проверяются во время компиляции и также применяются при компиляции оптимизированных сборок.

Examples

static_assert

Утверждения означают, что условие должно быть проверено, и если оно ложно, это ошибка. Для `static_assert()` это делается компиляцией.

```
template<typename T>
T mul10(const T t)
{
    static_assert( std::is_integral<T>::value, "mul10() only works for integral types" );
    return (t << 3) + (t << 1);
}
```

У `static_assert()` есть обязательный первый параметр, условие - это `bool constexpr`. Он *может* иметь второй параметр, это сообщение, которое является строковым литералом. Из C ++ 17 второй параметр является необязательным; до этого он является обязательным.

C ++ 17

```
template<typename T>
T mul10(const T t)
{
    static_assert( std::is_integral<T>::value );
}
```

```
return (t << 3) + (t << 1);  
}
```

Он используется, когда:

- В общем случае проверка во время компиляции требуется для некоторого типа по значению constexpr
- Функция шаблона должна проверять определенные свойства передаваемого им типа
- Один хочет написать тестовые примеры:
 - шаблонные метафоры
 - функции constexpr
 - макропрограммирование макросов
- Требуются определенные определения (например, версия на C++)
- Портинг устаревшего кода, утверждений о `sizeof(T)` (например, 32-битный int)
- Для работы программы требуются определенные функции компилятора (упаковка, пустая оптимизация базового класса и т. Д.).

Обратите внимание, что `static_assert()` не участвует в **SFINAE**: таким образом, когда возможны дополнительные перегрузки / специализации, его нельзя использовать вместо методов метапрограммирования шаблонов (например, `std::enable_if<>`). Он может использоваться в коде шаблона, когда ожидаемая перегрузка / специализация уже найдена, но необходимы дальнейшие проверки. В таких случаях он может предоставить более конкретные сообщения об ошибках, чем полагаться на SFINAE для этого.

Прочитайте `static_assert` онлайн: <https://riptutorial.com/ru/cplusplus/topic/3822/static-assert>

глава 12: std :: function: для обертывания любого элемента, вызываемого

Examples

Простое использование

```
#include <iostream>
#include <functional>
std::function<void(int , const std::string&)> myFuncObj;
void theFunc(int i, const std::string& s)
{
    std::cout << s << ": " << i << std::endl;
}
int main(int argc, char *argv[])
{
    myFuncObj = theFunc;
    myFuncObj(10, "hello world");
}
```

std :: function, используемая с std :: bind

Подумайте о ситуации, когда нам нужно вызвать функцию с аргументами. `std::function` используемая с `std::bind` дает очень мощную конструкцию проекта, как показано ниже.

```
class A
{
public:
    std::function<void(int, const std::string&)> m_CbFunc = nullptr;
    void foo()
    {
        if (m_CbFunc)
        {
            m_CbFunc(100, "event fired");
        }
    }
};

class B
{
public:
    B()
    {
        auto aFunc = std::bind(&B::eventHandler, this, std::placeholders::_1,
std::placeholders::_2);
        anObjA.m_CbFunc = aFunc;
    }
    void eventHandler(int i, const std::string& s)
    {
        std::cout << s << ": " << i << std::endl;
    }
};
```



```

    }

    void DoSomethingOnA()
    {
        anObjA.foo();
    }

    A anObjA;
};

int main(int argc, char *argv[])
{
    B anObjB;
    anObjB.DoSomethingOnA();
}

```

std :: function c lambda и std :: bind

```

#include <iostream>
#include <functional>

using std::placeholders::_1; // to be used in std::bind example

int stdf_foobar (int x, std::function<int(int)> moo)
{
    return x + moo(x); // std::function moo called
}

int foo (int x) { return 2+x; }

int foo_2 (int x, int y) { return 9*x + y; }

int main()
{
    int a = 2;

    /* Function pointers */
    std::cout << stdf_foobar(a, &foo) << std::endl; // 6 ( 2 + (2+2) )
    // can also be: stdf_foobar(2, foo)

    /* Lambda expressions */
    /* An unnamed closure from a lambda expression can be
     * stored in a std::function object:
     */
    int capture_value = 3;
    std::cout << stdf_foobar(a,
                            [capture_value](int param) -> int { return 7 + capture_value *
param; })
                << std::endl;
    // result: 15 == value + (7 * capture_value * value) == 2 + (7 + 3 * 2)

    /* std::bind expressions */
    /* The result of a std::bind expression can be passed.
     * For example by binding parameters to a function pointer call:
     */
    int b = stdf_foobar(a, std::bind(foo_2, _1, 3));
    std::cout << b << std::endl;
    // b == 23 == 2 + ( 9*2 + 3 )
    int c = stdf_foobar(a, std::bind(foo_2, 5, _1));
}

```

```
std::cout << c << std::endl;
// c == 49 == 2 + ( 9*5 + 2 )

return 0;
}
```

«служебные» функции

`std::function` может вызвать значительные накладные расходы. Поскольку `std::function` имеет [значение семантики] [1], она должна скопировать или переместить данный вызываемый в себя. Но поскольку он может принимать вызовы произвольного типа, ему часто приходится выделять память динамически для этого.

Некоторые реализации `function` имеют так называемую «оптимизацию небольших объектов», где небольшие объекты (например, указатели на функции, указатели элементов или функторы с очень маленьким состоянием) будут храниться непосредственно в объекте `function`. Но даже это работает только в том случае, если тип `noexcept`. Кроме того, стандарт C++ не требует, чтобы все реализации обеспечивали один.

Рассмотрим следующее:

```
//Header file
using MyPredicate = std::function<bool(const MyValue &, const MyValue &)>;

void SortMyContainer(MyContainer &C, const MyPredicate &pred);

//Source file
void SortMyContainer(MyContainer &C, const MyPredicate &pred)
{
    std::sort(C.begin(), C.end(), pred);
}
```

Параметр шаблона будет предпочтительным решением для `SortMyContainer`, но предположим, что это невозможно или желательно по любой причине. `SortMyContainer` не нужно хранить `pred` за пределами своего собственного вызова. И все же, `pred` может хорошо распределять память, если предоставленный ему функтор имеет некоторый нетривиальный размер.

`function` выделяет память, потому что ей нужно что-то скопировать / переместить; `function` переходит в собственность вызываемого. Но `SortMyContainer` не должен *владеть* вызываемым; это просто ссылка на него. Поэтому использование `function` здесь является излишним; он может быть эффективным, но может и не быть.

Нет стандартного типа библиотеки, который просто ссылается на вызываемый. Поэтому потребуется найти альтернативное решение, или вы можете жить с накладными расходами.

Кроме того, `function` не имеет эффективных средств для управления тем, откуда

происходят выделения памяти для объекта. Да, у него есть конструкторы, которые берут `allocator`, но [многие реализации не реализуют их правильно ... или даже *вообще*] [2].

C++ 17

Конструкторы `function` которые принимают `allocator` больше не являются частью типа. Поэтому нет способа управлять распределением.

Вызов `function` также медленнее, чем вызов содержимого напрямую. Поскольку любой экземпляр `function` может содержать любые вызываемые вызовы, вызов через `function` должен быть косвенным. Накладные расходы `function` вызова находятся в порядке вызова виртуальной функции.

Связывание `std::function` с различными вызываемыми типами

```
/*
 * This example show some ways of using std::function to call
 * a) C-like function
 * b) class-member function
 * c) operator()
 * d) lambda function
 *
 * Function call can be made:
 * a) with right arguments
 * b) argumens with different order, types and count
 */
#include <iostream>
#include <functional>
#include <iostream>
#include <vector>

using std::cout;
using std::endl;
using namespace std::placeholders;

// simple function to be called
double foo_fn(int x, float y, double z)
{
    double res = x + y + z;
    std::cout << "foo_fn called with arguments: "
              << x << ", " << y << ", " << z
              << " result is : " << res
              << std::endl;
    return res;
}

// structure with member function to call
struct foo_struct
{
    // member function to call
    double foo_fn(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::foo_fn called with arguments: "
```

```

        << x << ", " << y << ", " << z
        << " result is : " << res
        << std::endl;
    return res;
}
// this member function has different signature - but it can be used too
// please not that argument order is changed too
double foo_fn_4(int x, double z, float y, long xx)
{
    double res = x + y + z + xx;
    std::cout << "foo_struct::foo_fn_4 called with arguments: "
        << x << ", " << z << ", " << y << ", " << xx
        << " result is : " << res
        << std::endl;
    return res;
}
// overloaded operator() makes whole object to be callable
double operator()(int x, float y, double z)
{
    double res = x + y + z;
    std::cout << "foo_struct::operator() called with arguments: "
        << x << ", " << y << ", " << z
        << " result is : " << res
        << std::endl;
    return res;
}
};

int main(void)
{
    // typedefs
    using function_type = std::function<double(int, float, double)>;

    // foo_struct instance
    foo_struct fs;

    // here we will store all binded functions
    std::vector<function_type> bindings;

    // var #1 - you can use simple function
    function_type var1 = foo_fn;
    bindings.push_back(var1);

    // var #2 - you can use member function
    function_type var2 = std::bind(&foo_struct::foo_fn, fs, _1, _2, _3);
    bindings.push_back(var2);

    // var #3 - you can use member function with different signature
    // foo_fn_4 has different count of arguments and types
    function_type var3 = std::bind(&foo_struct::foo_fn_4, fs, _1, _3, _2, 0l);
    bindings.push_back(var3);

    // var #4 - you can use object with overloaded operator()
    function_type var4 = fs;
    bindings.push_back(var4);

    // var #5 - you can use lambda function
    function_type var5 = [](int x, float y, double z)
    {
        double res = x + y + z;

```

```

        std::cout << "lambda  called with arguments: "
            << x << ", " << y << ", " << z
            << " result is : " << res
            << std::endl;
        return res;
    };
    bindings.push_back(var5);

    std::cout << "Test stored functions with arguments: x = 1, y = 2, z = 3"
        << std::endl;

    for (auto f : bindings)
        f(1, 2, 3);
}

```

Жить

Выход:

```

Test stored functions with arguments: x = 1, y = 2, z = 3
foo_fn called with arguments: 1, 2, 3 result is : 6
foo_struct::foo_fn called with arguments: 1, 2, 3 result is : 6
foo_struct::foo_fn_4 called with arguments: 1, 3, 2, 0 result is : 6
foo_struct::operator() called with arguments: 1, 2, 3 result is : 6
lambda  called with arguments: 1, 2, 3 result is : 6

```

Сохранение аргументов функции в std :: tuple

Некоторым программам необходимо хранить аргументы для будущего вызова некоторой функции.

В этом примере показано, как вызвать любую функцию с аргументами, хранящимися в std :: tuple

```

#include <iostream>
#include <functional>
#include <tuple>
#include <iostream>

// simple function to be called
double foo_fn(int x, float y, double z)
{
    double res = x + y + z;
    std::cout << "foo_fn called. x = " << x << " y = " << y << " z = " << z
        << " res=" << res;
    return res;
}

// helpers for tuple unrolling
template<int ...> struct seq {};
template<int N, int ...S> struct gens : gens<N-1, N-1, S...> {};
template<int ...S> struct gens<0, S...>{ typedef seq<S...> type; };

// invocation helper
template<typename FN, typename P, int ...S>

```

```
double call_fn_internal(const FN& fn, const P& params, const seq<S...>)
{
    return fn(std::get<S>(params) ...);
}
// call function with arguments stored in std::tuple
template<typename Ret, typename ...Args>
Ret call_fn(const std::function<Ret(Args...)>& fn,
            const std::tuple<Args...>& params)
{
    return call_fn_internal(fn, params, typename gens<sizeof...(Args)>::type());
}

int main(void)
{
    // arguments
    std::tuple<int, float, double> t = std::make_tuple(1, 5, 10);
    // function to call
    std::function<double(int, float, double)> fn = foo_fn;

    // invoke a function with stored arguments
    call_fn(fn, t);
}
```

Жить

Выход:

```
foo_fn called. x = 1 y = 5 z = 10 res=16
```

Прочитайте `std::function`: для обертывания любого элемента, вызываемого онлайн:
<https://riptutorial.com/ru/cplusplus/topic/2294/std---function--для-обертывания-любого-элемента--вызываемого>

глава 13: std :: set и std :: multiset

Вступление

`set` - это тип контейнера, элементы которого отсортированы и уникальны. `multiset` аналогичен, но в случае `multiset` множественные элементы могут иметь одинаковое значение.

замечания

В этих примерах использовались разные стили C++. Будьте осторожны, если вы используете компилятор C++ 98; некоторые из этого кода могут не использоваться.

Examples

Вставка значений в набор

Три разных метода вставки могут использоваться с наборами.

- Во-первых, простая вставка значения. Этот метод возвращает пару, позволяющую вызывающему абоненту проверить, действительно ли вставка произошла.
- Во-вторых, вставка, давая подсказку о том, где значение будет вставлено. Цель состоит в том, чтобы оптимизировать время вставки в таком случае, но знание того, где должно быть вставлено значение, не является обычным случаем. **Будьте осторожны в этом случае; способ дать подсказку отличается от версий компилятора.**
- Наконец, вы можете вставить диапазон значений, указав начальный и конечный указатель. Исходный из них будет включен во вставку, конечный - исключен.

```
#include <iostream>
#include <set>

int main ()
{
    std::set<int> sut;
    std::set<int>::iterator it;
    std::pair<std::set<int>::iterator,bool> ret;

    // Basic insert
    sut.insert(7);
    sut.insert(5);
    sut.insert(12);

    ret = sut.insert(23);
    if (ret.second==true)
        std::cout << "# 23 has been inserted!" << std::endl;
```

```

    ret = sut.insert(23); // since it's a set and 23 is already present in it, this insert
    should fail
    if (ret.second==false)
        std::cout << "# 23 already present in set!" << std::endl;

    // Insert with hint for optimization
    it = sut.end();
    // This case is optimized for C++11 and above
    // For earlier version, point to the element preceding your insertion
    sut.insert(it, 30);

    // inserting a range of values
    std::set<int> sut2;
    sut2.insert(20);
    sut2.insert(30);
    sut2.insert(45);
    std::set<int>::iterator itStart = sut2.begin();
    std::set<int>::iterator itEnd = sut2.end();

    sut.insert (itStart, itEnd); // second iterator is excluded from insertion

    std::cout << std::endl << "Set under test contains:" << std::endl;
    for (it = sut.begin(); it != sut.end(); ++it)
    {
        std::cout << *it << std::endl;
    }

    return 0;
}

```

Выход будет:

```

# 23 has been inserted!

# 23 already present in set!


Set under test contains:

5

7

12

20

23

30

45

```

Вставка значений в мультимножество

Все методы вставки из наборов также применяются к мультимножествам. Тем не менее,

существует еще одна возможность, предоставляющая `initializer_list`:

```
auto il = { 7, 5, 12 };
std::multiset<int> msut;
msut.insert(il);
```

Изменение типа сортировки по умолчанию

`set` и `multiset` имеют методы сравнения по умолчанию, но в некоторых случаях вам может потребоваться перегрузить их.

Представим себе, что мы сохраняем строковые значения в наборе, но мы знаем, что эти строки содержат только числовые значения. По умолчанию сортировка будет представлять собой сравнение лексикографической строки, поэтому порядок не будет соответствовать численному типу. Если вы хотите применить сортировку, эквивалентную тому, что у вас было бы с значениями `int`, вам нужен функтор для перегрузки метода сравнения:

```
#include <iostream>
#include <set>
#include <stdlib.h>

struct custom_compare final
{
    bool operator() (const std::string& left, const std::string& right) const
    {
        int nLeft = atoi(left.c_str());
        int nRight = atoi(right.c_str());
        return nLeft < nRight;
    }
};

int main ()
{
    std::set<std::string> sut({"1", "2", "5", "23", "6", "290"});

    std::cout << "### Default sort on std::set<std::string> :" << std::endl;
    for (auto &&data: sut)
        std::cout << data << std::endl;

    std::set<std::string, custom_compare> sut_custom({"1", "2", "5", "23", "6", "290"},
                                                    custom_compare{}); //< Compare object
    optional as its default constructible.

    std::cout << std::endl << "### Custom sort on set :" << std::endl;
    for (auto &&data : sut_custom)
        std::cout << data << std::endl;

    auto compare_via_lambda = [](auto &&lhs, auto &&rhs){ return lhs > rhs; };
    using set_via_lambda = std::set<std::string, decltype(compare_via_lambda)>;
    set_via_lambda sut_reverse_via_lambda({"1", "2", "5", "23", "6", "290"},
                                          compare_via_lambda);

    std::cout << std::endl << "### Lambda sort on set :" << std::endl;
    for (auto &&data : sut_reverse_via_lambda)
```

```

        std::cout << data << std::endl;

    return 0;
}

```

Выход будет:

```

### Default sort on std::set<std::string> :
1
2
23
290
5
6
### Custom sort on set :
1
2
5
6
23
290

### Lambda sort on set :
6
5
290
23
2
1

```

В приведенном выше примере можно найти 3 разных способа добавления операций сравнения в `std::set`, каждый из которых полезен в своем собственном контексте.

Сорт по умолчанию

Это будет использовать оператор сравнения ключа (первый аргумент шаблона). Часто ключ уже обеспечивает хорошее значение по умолчанию для функции `std::less<T>`. Если эта функция не является специализированной, она использует `operator<` объекта. Это особенно полезно, когда другой код также пытается использовать некоторый порядок, поскольку это позволяет обеспечить согласованность всей базы кода.

Написание кода таким образом уменьшит усилия по обновлению вашего кода, когда ключевыми изменениями будут API, например: класс, содержащий 2 члена, который изменяется на класс, содержащий 3 члена. Обновляя `operator<` в классе, все вхождения будут обновляться.

Как и следовало ожидать, использование сортировки по умолчанию является разумным по умолчанию.

Пользовательский вид

Добавление пользовательской сортировки через объект с оператором сравнения часто используется, когда сравнение по умолчанию не выполняется. В приведенном выше примере это связано с тем, что строки ссылаются на целые числа. В других случаях это часто используется, когда вы хотите сравнивать (умные) указатели на основе объекта, на который они ссылаются, или потому, что вам нужны разные ограничения для сравнения (например: сравнение `std::pair` по значению `first`).

При создании оператора сравнения это должна быть стабильная сортировка. Если результат оператора сравнения изменяется после вставки, вы будете иметь неопределенное поведение. Как хорошая практика, ваш оператор сравнения должен использовать только постоянные данные (константные члены, функции `const` ...).

Как и в примере выше, вы часто сталкиваетесь с классами без членов в качестве операторов сравнения. Это приводит к конструкторам по умолчанию и конструкторам копирования. Конструктор по умолчанию позволяет опустить экземпляр во время построения, и требуется конструктор копирования, так как набор принимает копию оператора сравнения.

Лямбда-сортировка

Lambdas - это более короткий способ писать объекты функции. Это позволяет записывать оператор сравнения на меньшее количество строк, делая общий код более удобочитаемым.

Недостатком использования `lambdas` является то, что каждый `lambda` получает определенный тип во время компиляции, поэтому `decltype(lambda)` будет отличаться для каждой компиляции одной и той же единицы компиляции (файл `cpp`) как множество множественных единиц компиляции (если они включены в заголовочный файл). По этой причине рекомендуется использовать функциональные объекты в качестве оператора сравнения при использовании в файлах заголовков.

Эта конструкция часто встречается, когда `std::set` используется в локальной области действия функции, тогда как функциональный объект является предпочтительным при использовании в качестве аргументов функции или членов класса.

Другие варианты сортировки

Поскольку оператор сравнения `std::set` является аргументом шаблона, все **вызываемые объекты** могут использоваться как оператор сравнения, а приведенные выше примеры - это только конкретные случаи. Единственными ограничениями, которые имеют эти вызываемые объекты, являются:

- Они должны быть готовы к копированию
- Они должны быть вызваны двумя аргументами типа ключа. (подразумеваемые преобразования разрешены, но не рекомендуется, так как это может повредить

производительность)

Поиск значений в наборе и мультимножестве

Существует несколько способов поиска заданного значения в `std::set` или в `std::multiset`:

Чтобы получить итератор первого вхождения ключа, можно использовать функцию `find()`. Он возвращает `end()` если ключ не существует.

```
std::set<int> sut;
sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3); // contains 3, 10, 15, 22

auto itS = sut.find(10); // the value is found, so *itS == 10
itS = sut.find(555); // the value is not found, so itS == sut.end()

std::multiset<int> msut;
sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(15);
sut.insert(3); // contains 3, 10, 15, 15, 22

auto itMS = msut.find(10);
```

Другой способ - использовать функцию `count()`, которая подсчитывает, сколько соответствующих значений найдено в `set` / `multiset` (в случае `set`, возвращаемое значение может быть только 0 или 1). Используя те же значения, что и выше, мы будем иметь:

```
int result = sut.count(10); // result == 1
result = sut.count(555); // result == 0

result = msut.count(10); // result == 1
result = msut.count(15); // result == 2
```

В случае `std::multiset` может быть несколько элементов, имеющих одинаковое значение. Чтобы получить этот диапазон, можно использовать функцию `equal_range()`. Он возвращает `std::pair` имеющую нижнюю границу итератора (включительно) и верхнюю границу (исключая) соответственно. Если ключ не существует, оба итератора будут указывать на ближайшее превосходящее значение (на основе метода сравнения, используемого для сортировки заданного `multiset`).

```
auto eqr = msut.equal_range(15);
auto st = eqr.first; // point to first element '15'
auto en = eqr.second; // point to element '22'

eqr = msut.equal_range(9); // both eqr.first and eqr.second point to element '10'
```

Удаление значений из набора

Самый очевидный метод, если вы просто хотите сбросить свой набор / мультимножество на пустой, - это использовать `clear` :

```
std::set<int> sut;
sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3);
sut.clear(); //size of sut is 0
```

Затем можно использовать метод `erase` . Он предлагает несколько возможностей, несколько похожих на вставку:

```
std::set<int> sut;
std::set<int>::iterator it;

sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3);
sut.insert(30);
sut.insert(33);
sut.insert(45);

// Basic deletion
sut.erase(3);

// Using iterator
it = sut.find(22);
sut.erase(it);

// Deleting a range of values
it = sut.find(33);
sut.erase(it, sut.end());

std::cout << std::endl << "Set under test contains:" << std::endl;
for (it = sut.begin(); it != sut.end(); ++it)
{
    std::cout << *it << std::endl;
}
```

Выход будет:

```
Set under test contains:

10

15

30
```

Все эти методы также применяются к `multiset` . Обратите внимание, что если вы попросите удалить элемент из `multiset` , и он присутствует несколько раз, **все эквивалентные значения будут удалены** .

Прочитайте `std::set` и `std::multiset` онлайн: <https://riptutorial.com/ru/cplusplus/topic/9005/std---set-и-std---multiset>

глава 14: авто

замечания

Ключевое слово `auto` - это имя-тип, представляющий автоматически выведенный тип.

Это уже зарезервированное ключевое слово в C++ 98, унаследованное от C. В старых версиях C++ его можно было использовать для явного указания, что переменная имеет автоматическую продолжительность хранения:

```
int main()
{
    auto int i = 5; // removing auto has no effect
}
```

Этот старый смысл теперь удален.

Examples

Основной автоматический образец

Ключевое слово `auto` предоставляет автоматический вывод типа переменной.

Это особенно удобно при работе с длинными именами типов:

```
std::map< std::string, std::shared_ptr< Widget > > table;
// C++98
std::map< std::string, std::shared_ptr< Widget > >::iterator i = table.find( "42" );
// C++11/14/17
auto j = table.find( "42" );
```

с диапазоном для циклов :

```
vector<int> v = {0, 1, 2, 3, 4, 5};
for(auto n: v)
    std::cout << n << ' ';
```

с лямбдами :

```
auto f = [](){ std::cout << "lambda\n"; };
f();
```

чтобы избежать повторения типа:

```
auto w = std::make_shared< Widget >();
```

чтобы избежать неожиданных и ненужных копий:

```
auto myMap = std::map<int, float>();
myMap.emplace(1, 3.14);

std::pair<int, float> const& firstPair2 = *myMap.begin(); // copy!
auto const& firstPair = *myMap.begin(); // no copy!
```

Причиной для копирования является то, что возвращаемый тип фактически является `std::pair<const int, float> !`

авто и экспрессии

`auto` также может вызвать проблемы, в которые входят шаблоны выражения:

```
auto mult(int c) {
    return c * std::valarray<int>{1};
}

auto v = mult(3);
std::cout << v[0]; // some value that could be, but almost certainly is not, 3.
```

Причина в том, что `operator*` на `valarray` дает вам прокси-объект, который ссылается на `valarray` как средство ленивой оценки. Используя `auto`, вы создаете оборванную ссылку. Вместо `mult` вернул `std::valarray<int>`, тогда код определенно напечатает 3.

auto, const и ссылки

Ключевое слово `auto` представляет собой тип значения, аналогичный `int` или `char`. Он может быть изменен с помощью ключевого слова `const` и символа `&` для представления типа `const` или ссылочного типа, соответственно. Эти модификаторы можно комбинировать.

В этом примере `s` является типом значения (его тип будет выведен как `std::string`), поэтому каждая итерация цикла `for` копирует строку из вектора в `s`.

```
std::vector<std::string> strings = { "stuff", "things", "misc" };
for(auto s : strings) {
    std::cout << s << std::endl;
}
```

Если тело цикла изменяет `s` (например, вызывая `s.append(" and stuff")`), будет изменена только эта копия, а не исходный член `strings`.

С другой стороны, если `s` объявлено с `auto&` он будет ссылочным типом (предполагается, что он является `std::string&`), поэтому на каждой итерации цикла ему будет назначена ссылка на строку в векторе:

```
for(auto& s : strings) {
    std::cout << s << std::endl;
}
```



```
}
```

В теле этого цикла изменения `s` будут непосредственно влиять на элемент `strings` который он ссылается.

Наконец, если `s` объявлен `const auto&`, это будет тип ссылки `const`, что означает, что на каждой итерации цикла ему будет назначена *ссылка const* на строку в векторе:

```
for(const auto& s : strings) {  
    std::cout << s << std::endl;  
}
```

Внутри тела этого цикла `s` не может быть изменен (т.е. на нем не могут быть вызваны неконстантные методы).

При использовании `auto` с использованием циклов, основанных `for` диапазонах, обычно рекомендуется использовать `const auto&` если тело цикла не будет изменять структуру, закливаемую, поскольку это позволяет избежать ненужных копий.

Обратный тип возврата

`auto` используется в синтаксисе для возвращаемого возвращаемого типа:

```
auto main() -> int {}
```

что эквивалентно

```
int main() {}
```

В основном полезно в сочетании с `decltype` использовать параметры вместо `std::declval<T>`:

```
template <typename T1, typename T2>  
auto Add(const T1& lhs, const T2& rhs) -> decltype(lhs + rhs) { return lhs + rhs; }
```

Общий лямбда (C ++ 14)

C ++ 14

C ++ 14 позволяет использовать `auto` в аргументе лямбда

```
auto print = [](const auto& arg) { std::cout << arg << std::endl; };  
  
print(42);  
print("hello world");
```

Эта лямбда в основном эквивалентна

```
struct lambda {
    template <typename T>
    auto operator () (const T& arg) const {
        std::cout << arg << std::endl;
    }
};
```

а потом

```
lambda print;

print(42);
print("hello world");
```

авто и прокси-объекты

Иногда `auto` может вести себя не так, как ожидал программист. Этот тип выводит выражение, даже если вывод типа не подходит.

Например, когда в коде используются прокси-объекты:

```
std::vector<bool> flags{true, true, false};
auto flag = flags[0];
flags.push_back(true);
```

Здесь `flag` будет не `bool`, а `std::vector<bool>::reference`, так как для `bool` специализации `vector` шаблона `operator []` возвращает прокси-объект с оператором `operator bool` преобразования `operator bool`.

Когда `flags.push_back(true)` изменяет контейнер, эта псевдо-ссылка может оказаться свисающей, ссылаясь на элемент, который больше не существует.

Это также делает возможной следующую ситуацию:

```
void foo(bool b);

std::vector<bool> getFlags();

auto flag = getFlags()[5];
foo(flag);
```

`vector` немедленно отбрасывается, поэтому `flag` является псевдо-ссылкой на элемент, который был отброшен. Вызов `foo` вызывает неопределенное поведение.

В таких случаях вы можете объявить переменную с `auto` и инициализировать ее, выбрав тип, который вы хотите вывести:

```
auto flag = static_cast<bool>(getFlags()[5]);
```

но в этот момент просто замена `auto` на `bool` имеет больше смысла.

Другим случаем, когда объекты-прокси могут вызывать проблемы, являются [шаблоны выражений](#). В этом случае шаблоны иногда не рассчитаны на то, чтобы оставаться выше текущего полного выражения для эффективности, а использование прокси-объекта в следующем случае приводит к неопределенному поведению.

Прочитайте авто онлайн: <https://riptutorial.com/ru/cplusplus/topic/2421/авто>

глава 15: Арифметика с плавающей точкой

Examples

Числа с плавающей запятой - странные

Первая ошибка, которую делает почти каждый программист, предполагает, что этот код будет работать по назначению:

```
float total = 0;
for(float a = 0; a != 2; a += 0.01f) {
    total += a;
}
```

Программист-новичок предполагает, что это суммирует каждое отдельное число в диапазоне 0, 0.01, 0.02, 0.03, ..., 1.97, 1.98, 1.99, чтобы получить результат 199 - математически правильный ответ.

Происходят две вещи, которые делают это неверным:

1. Программа, как написано, никогда не завершается. a никогда не становится равным 2, и цикл никогда не заканчивается.
2. Если мы перепишем логику цикла для проверки $a < 2$ вместо этого, цикл завершится, но итоговое значение окажется чем-то отличным от 199. На совместимых с IEEE754 машинах он часто будет содержать примерно до 201.

Причина, по которой это происходит, заключается в том, что числа с плавающей запятой представляют собой приближенные значения их назначенных значений.

Классическим примером является следующее вычисление:

```
double a = 0.1;
double b = 0.2;
double c = 0.3;
if(a + b == c)
    //This never prints on IEEE754-compliant machines
    std::cout << "This Computer is Magic!" << std::endl;
else
    std::cout << "This Computer is pretty normal, all things considered." << std::endl;
```

Хотя то, что мы, программист видим, это три числа, написанные в base10, то, что видят компилятор (и базовое оборудование), являются двоичными числами. Поскольку 0.1, 0.2 и 0.3 требуют идеального деления на 10 что довольно легко в системе base-10, но невозможно в системе base-2 - эти цифры должны храниться в неточных форматах, подобно тому, как число $1/3$ должен храниться в неточной форме 0.333333333333333... в базе-10.

```
//64-bit floats have 53 digits of precision, including the whole-number-part.  
double a =      001111111011100110011001100110011001100110011010; //imperfect  
representation of 0.1  
double b =      001111111100100110011001100110011001100110011010; //imperfect  
representation of 0.2  
double c =      001111111101001100110011001100110011001100110011; //imperfect  
representation of 0.3  
double a + b = 001111111101001100110011001100110011001100110100; //Note that  
this is not quite equal to the "canonical" 0.3!
```

Прочитайте Арифметика с плавающей точкой онлайн:

<https://riptutorial.com/ru/cplusplus/topic/5115/арифметика-с-плавающей-точкой>

глава 16: Арифметическое метапрограммирование

Вступление

Это пример использования метапрограммирования шаблонов C++ при обработке арифметических операций во время компиляции.

Examples

Вычисление мощности в $O(\log n)$

В этом примере показан эффективный способ вычисления мощности с использованием метапрограммирования шаблонов.

```
template <int base, unsigned int exponent>
struct power
{
    static const int halfvalue = power<base, exponent / 2>::value;
    static const int value = halfvalue * halfvalue * power<base, exponent % 2>::value;
};

template <int base>
struct power<base, 0>
{
    static const int value = 1;
    static_assert(base != 0, "power<0, 0> is not allowed");
};

template <int base>
struct power<base, 1>
{
    static const int value = base;
};
```

Пример использования:

```
std::cout << power<2, 9>::value;
```

C++ 14

Он также обрабатывает отрицательные показатели:

```
template <int base, int exponent>
struct powerDouble
{
    static const int exponentAbs = exponent < 0 ? (-exponent) : exponent;
```

```

        static const int halfvalue = powerDouble<base, exponentAbs / 2>::intermediateValue;
        static const int intermediateValue = halfvalue * halfvalue * powerDouble<base, exponentAbs
% 2>::intermediateValue;

        constexpr static double value = exponent < 0 ? (1.0 / intermediateValue) :
intermediateValue;

};

template <int base>
struct powerDouble<base, 0>
{
    static const int intermediateValue = 1;
    constexpr static double value = 1;
    static_assert(base != 0, "powerDouble<0, 0> is not allowed");
};

template <int base>
struct powerDouble<base, 1>
{
    static const int intermediateValue = base;
    constexpr static double value = base;
};

int main()
{
    std::cout << powerDouble<2, -3>::value;
}

```

Прочитайте Арифметическое метапрограммирование онлайн:

<https://riptutorial.com/ru/cplusplus/topic/10907/арифметическое-метапрограммирование>

глава 17: Атомные типы

Синтаксис

- `std::atomic<T>`
- `std::atomic_flag`

замечания

`std::atomic` позволяет атомарному доступу к типу `TriviallyCopyable`, он зависит от реализации, если это выполняется посредством атомных операций или с помощью блокировок. Единственный гарантированный атомный тип без блокировки - `std::atomic_flag`.

Examples

Многопоточный доступ

Атомный тип может использоваться для безопасного чтения и записи в ячейку памяти, разделяемую между двумя потоками.

Плохой пример, который может привести к гонке данных:

```
#include <thread>
#include <iostream>

//function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, int * result) {
    for (int i = a; i <= b; i++) {
        *result += i;
    }
}

int main() {
    //a primitive data type has no thread safety
    int shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with paramters a = 1, b = 100,
    result = &shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 100, &shared);

    //attempt to print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //this may cause undefined behavior or print a corrupted value
        //if the addingThread tries to write to 'shared' while the main thread is reading it
    }
```



```

        std::cout << shared << std::endl;
    }

    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}

```

Вышеприведенный пример может привести к повреждению чтения и может привести к неопределенному поведению.

Пример с безопасностью потока:

```

#include <atomic>
#include <thread>
#include <iostream>

//function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, std::atomic<int> * result) {
    for (int i = a; i <= b; i++) {
        //atomically add 'i' to result
        result->fetch_add(i);
    }
}

int main() {
    //atomic template used to store non-atomic objects
    std::atomic<int> shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with paramters a = 1, b = 100,
    result = &shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 10000, &shared);

    //print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //safe way to read the value of shared atomically for thread safe read
        std::cout << shared.load() << std::endl;
    }

    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}

```

Вышеприведенный пример является безопасным, поскольку все операции `store()` и `load()` типа `atomic` данных защищают инкапсулированный `int` от одновременного доступа.

Прочитайте Атомные типы онлайн: <https://riptutorial.com/ru/cplusplus/topic/3804/атомные-типы>

глава 18: Атрибуты

Синтаксис

- `[[details]]`: простой атрибут без аргумента
- `[[details (аргументы)]]`: Атрибут с аргументами
- `__attribute` (подробности): нестандартная спецификация GCC / Clang / IBM
- `__declspec` (подробности): нестандартная спецификация MSVC

Examples

[[без возврата]]

C ++ 11

В C ++ 11 был введен атрибут `[[noreturn]]`. Он может использоваться для функции, указывающей на то, что функция не возвращается к вызывающей стороне, либо выполняя оператор `return`, либо достигая конца, если это тело (важно отметить, что это не относится к функциям `void`, поскольку они возвращаются к вызывающему, они просто не возвращают никакого значения). Такая функция может завершиться вызовом `std::terminate` или `std::exit` или путем исключения исключения. Также стоит отметить, что такая функция может вернуться, выполнив `longjmp`.

Например, функция, приведенная ниже, всегда будет либо выдавать исключение, либо вызывать `std::terminate`, поэтому он является хорошим кандидатом для `[[noreturn]]`:

```
[[noreturn]] void ownAssertFailureHandler(std::string message) {
    std::cerr << message << std::endl;
    if (THROW_EXCEPTION_ON_ASSERT)
        throw AssertException(std::move(message));
    std::terminate();
}
```

Такая функциональность позволяет компилятору завершить работу без оператора `return`, если он знает, что код никогда не будет выполнен. Здесь, поскольку вызов `ownAssertFailureHandler` (определенный выше) в приведенном ниже коде никогда не вернется, компилятору не нужно добавлять код ниже этого вызова:

```
std::vector<int> createSequence(int end) {
    if (end > 0) {
        std::vector<int> sequence;
        sequence.reserve(end+1);
        for (int i = 0; i <= end; ++i)
```

```

        sequence.push_back(i);
        return sequence;
    }
    ownAssertFailureHandler("Negative number passed to createSequence()"s);
    // return std::vector<int>{}; //< Not needed because of [[noreturn]]
}

```

Это неопределенное поведение, если функция действительно вернется, поэтому следующее недопустимо:

```

[[noreturn]] void assertPositive(int number) {
    if (number >= 0)
        return;
    else
        ownAssertFailureHandler("Positive number expected"s); //< [[noreturn]]
}

```

Обратите внимание, что `[[noreturn]]` в основном используется в `void`-функциях. Однако это не является обязательным требованием, позволяющим использовать функции в общем программировании:

```

template<class InconsistencyHandler>
double fortyTwoDivideBy(int i) {
    if (i == 0)
        i = InconsistencyHandler::correct(i);
    return 42. / i;
}

struct InconsistencyThrower {
    static [[noreturn]] int correct(int i) { ownAssertFailureHandler("Unknown inconsistency"s); }
}

struct InconsistencyChangeToOne {
    static int correct(int i) { return 1; }
}

double fortyTwo = fortyTwoDivideBy<InconsistencyChangeToOne>(0);
double unreachable = fortyTwoDivideBy<InconsistencyThrower>(0);

```

Следующие стандартные функции библиотеки имеют этот атрибут:

- `std::prerr`
- `std::Exit`
- `std::quick_exit`
- `std::unexpected`
- `std::terminate`
- `std::rethrow_exception`
- `std::throw_with_nested`
- `std::nested_exception::rethrow_nested`

[[провалиться]]

C ++ 17

Всякий раз, когда `case` заканчивается в `switch`, код следующего случая будет выполнен. Последнюю можно предотвратить, используя оператор `'break'`. Так как это так называемое пошаговое поведение может вводить ошибки, когда они не предназначены, некоторые компиляторы и статические анализаторы дают предупреждение об этом.

Начиная с C ++ 17, был введен стандартный атрибут, указывающий на то, что предупреждение не требуется, когда код должен провалиться. Компиляторы могут безопасно давать предупреждения, когда дело заканчивается без `break` или `[[fallthrough]]` и имеет хотя бы одно заявление.

```
switch(input) {
    case 2011:
    case 2014:
    case 2017:
        std::cout << "Using modern C++" << std::endl;
        [[fallthrough]]; // > No warning
    case 1998:
    case 2003:
        standard = input;
}
```

См. [Предложение](#) для более подробных примеров того, как можно использовать `[[fallthrough]]`.

[[устаревшие]] и [[устаревшие («причина»)]]

C ++ 14

C ++ 14 представил стандартный способ устаревания функций через атрибуты.

`[[deprecated]]` может использоваться, чтобы указать, что функция устарела.

`[[deprecated("reason")]]` позволяет добавить определенную причину, которая может быть показана компилятором.

```
void function(std::unique_ptr<A> &&a);

// Provides specific message which helps other programmers fixing there code
[[deprecated("Use the variant with unique_ptr instead, this function will be removed in the
next release")]]
void function(std::auto_ptr<A> a);

// No message, will result in generic warning if called.
[[deprecated]]
void function(A *a);
```

Этот атрибут может применяться к:

- объявление класса
- typedef-name
- Переменная

- нестатический элемент данных
- функция
- перечисление
- типовая специализация

(см. [C ++ 14 стандартная черновик](#) : 7.6.5 Устаревший атрибут)

[[Nodiscard]]

C ++ 17

`[[nodiscard]]` может использоваться для указания того, что возвращаемое значение функции не должно игнорироваться при вызове функции. Если возвращаемое значение игнорируется, компилятор должен дать предупреждение об этом. Атрибут может быть добавлен в:

- Определение функции
- Тип

Добавление атрибута к типу имеет такое же поведение, как добавление атрибута к каждой отдельной функции, которая возвращает этот тип.

```
template<typename Function>
[[nodiscard]] Finally<std::decay_t<Function>> onExit(Function &&f);

void f(int &i) {
    assert(i == 0);                // Just to make comments clear!
    ++i;                          // i == 1
    auto exit1 = onExit([&i]{ --i; }); // Reduce by 1 on exiting f()
    ++i;                          // i == 2
    onExit([&i]{ --i; });          // BUG: Reducing by 1 directly
                                //      Compiler warning expected
    std::cout << i << std::end;    // Expected: 2, Real: 1
}
```

См. [Предложение](#) для более подробных примеров того, как можно использовать

`[[nodiscard]]` .

Примечание: Детали реализации `Finally` / `onExit` опущены в этом примере, см [Finally / ScopeExit](#) .

[[Maybe_unused]]

`[[maybe_unused]]` создан для указания в коде, что некоторая логика может не использоваться. Это часто связано с условиями препроцессора, где это может быть использовано или может не использоваться. Поскольку компиляторы могут давать предупреждения о неиспользуемых переменных, это способ подавить их, указав намерение.

Типичным примером переменных, которые необходимы в отладочных сборках, в то время как ненужные в производстве, являются возвращаемыми значениями, указывающими успех. В отладочных сборках условие должно быть подтверждено, хотя в производстве эти утверждения были удалены.

```
[[maybe_unused]] auto mapInsertResult = configuration.emplace("LicenseInfo",
stringifiedLicenseInfo);
assert(mapInsertResult.second); // We only get called during startup, so we can't be in the
map
```

Более сложным примером являются различные вспомогательные функции, которые находятся в неназванном пространстве имен. Если эти функции не используются во время компиляции, компилятор может дать предупреждение о них. В идеале вы хотели бы охранять их с теми же тегами препроцессора, что и вызывающий, хотя, поскольку это может стать сложным, атрибут `[[maybe_unused]]` является более `[[maybe_unused]]` альтернативой.

```
namespace {
    [[maybe_unused]] std::string createWindowsConfigFilePath(const std::string &relativePath);
    // TODO: Reuse this on BSD, MAC ...
    [[maybe_unused]] std::string createLinuxConfigFilePath(const std::string &relativePath);
}

std::string createConfigFilePath(const std::string &relativePath) {
    #if OS == "WINDOWS"
        return createWindowsConfigFilePath(relativePath);
    #elif OS == "LINUX"
        return createLinuxConfigFilePath(relativePath);
    #else
        #error "OS is not yet supported"
    #endif
}
```

См. [Предложение](#) для более подробных примеров того, как `[[maybe_unused]]` можно использовать.

Прочитайте Атрибуты онлайн: <https://riptutorial.com/ru/cplusplus/topic/5251/атрибуты>

глава 19: Базовый ввод / вывод в C++

замечания

Стандартная библиотека `<iostream>` определяет несколько потоков для ввода и вывода:

stream	description
cin	standard input stream
cout	standard output stream
cerr	standard error (output) stream
clog	standard logging (output) stream

Из четырех потоков, упомянутых выше, `cin` в основном используется для ввода пользователем, а для вывода данных используются три других. Обычно или в большинстве сред кодирования `cin` (*консольный ввод* или стандартный ввод) - это клавиатура и `cout` (*консольный вывод* или стандартный вывод).

```
cin >> value

cin      - input stream
'>>'    - extraction operator
value    - variable (destination)
```

`cin` здесь извлекает введенный пользователем ввод и передает значение переменной. Значение извлекается только после нажатия пользователем клавиши ENTER.

```
cout << "Enter a value: "

cout      - output stream
'<<'     - insertion operator
"Enter a value: " - string to be displayed
```

`cout` здесь отображает строку и вставляет ее в стандартный вывод или монитор

Все четыре потока расположены в стандартном пространстве имен `std` поэтому нам нужно напечатать `std::stream` для потока `stream` чтобы использовать его.

Существует также манипулятор `std::endl` в коде. Он может использоваться только с выходными потоками. Он вставляет конец строки `'\n'` в поток и сбрасывает его. Это приводит к немедленному производству продукции.

Examples

пользовательский ввод и стандартный вывод

```
#include <iostream>

int main()
{
    int value;
    std::cout << "Enter a value: " << std::endl;
    std::cin >> value;
    std::cout << "The square of entered value is: " << value * value << std::endl;
    return 0;
}
```

Прочитайте Базовый ввод / вывод в C++ онлайн:

<https://riptutorial.com/ru/cplusplus/topic/10683/базовый-ввод---вывод-в-c-plusplus>

глава 20: Без названия

Examples

Беззнаковые классы

В отличие от именованного класса или структуры, неназванные классы и структуры должны создаваться в тех случаях, когда они определены, и не могут иметь конструкторов или деструкторов.

```
struct {
    int foo;
    double bar;
} foobar;

foobar.foo = 5;
foobar.bar = 4.0;

class {
    int baz;
public:
    int buzz;

    void setBaz(int v) {
        baz = v;
    }
} barbar;

barbar.setBaz(15);
barbar.buzz = 2;
```

Анонимные участники

Как нестандартное расширение для C ++, обычные компиляторы позволяют использовать классы как анонимные члены.

```
struct Example {
    struct {
        int inner_b;
    };

    int outer_b;

    //The anonymous struct's members are accessed as if members of the parent struct
    Example() : inner_b(2), outer_b(4) {
        inner_b = outer_b + 2;
    }
};

Example ex;

//The same holds true for external code referencing the struct
```

```
ex.inner_b -= ex.outer_b;
```

Как псевдоним типа

Типы классов-типа могут также использоваться при создании псевдонимов типов, то есть посредством `typedef` и `using`:

C++ 11

```
using vec2d = struct {  
    float x;  
    float y;  
};
```

```
typedef struct {  
    float x;  
    float y;  
} vec2d;
```

```
vec2d pt;  
pt.x = 4.f;  
pt.y = 3.f;
```

Анонимный союз

Имена членов анонимного объединения относятся к сфере действия декларации союза, которая должна быть различной для всех других имен этой области. Пример здесь имеет ту же конструкцию, что и пример [анонимных пользователей](#) с использованием «struct», но является стандартным.

```
struct Sample {  
    union {  
        int a;  
        int b;  
    };  
    int c;  
};  
int main()  
{  
    Sample sa;  
    sa.a = 3;  
    sa.b = 4;  
    sa.c = 5;  
}
```

Прочитайте Без названия онлайн: <https://riptutorial.com/ru/cplusplus/topic/2704/без-названия>

глава 21: Бит-манипуляция

замечания

Чтобы использовать `std::bitset` вам нужно будет включить **заголовок** `<bitset>` .

```
#include <bitset>
```

`std::bitset` перегружает все функции оператора, чтобы обеспечить такое же использование, как обработка битов в стиле `c`.

Рекомендации

- [Бит Twiddling Hacks](#)

Examples

Установка бит

C-стиль бит манипуляции

Бит может быть установлен с помощью побитового оператора OR (`|`).

```
// Bit x will be set  
number |= 1LL << x;
```

Использование `std::bitset`

`set(x)` или `set(x,true)` - устанавливает бит в положение `x` в `1` .

```
std::bitset<5> num(std::string("01100"));  
num.set(0);      // num is now 01101  
num.set(2);      // num is still 01101  
num.set(4,true); // num is now 11110
```

Очистка бит

C-стиль бит-манипуляции

Бит можно очистить с помощью побитового оператора И (&).

```
// Bit x will be cleared
number &= ~(1LL << x);
```

Использование std :: bitset

reset (x) или set (x, false) - очищает бит в позиции x .

```
std::bitset<5> num(std::string("01100"));
num.reset(2);      // num is now 01000
num.reset(0);      // num is still 01000
num.set(3, false); // num is now 00000
```

Переключение немного

С-стиль бит-манипуляции

Бит можно переключить с помощью оператора XOR (^).

```
// Bit x will be the opposite value of what it is currently
number ^= 1LL << x;
```

Использование std :: bitset

```
std::bitset<4> num(std::string("0100"));
num.flip(2); // num is now 0000
num.flip(0); // num is now 0001
num.flip();  // num is now 1110 (flips all bits)
```

Проверка бит

С-стиль бит-манипуляции

Значение бит может быть получено путем смещения числа вправо x раз, а затем выполнения побитового AND (&) на нем:

```
(number >> x) & 1LL; // 1 if the 'x'th bit of 'number' is set, 0 otherwise
```

Операция смены вправо может быть реализована как арифметический (подписанный) сдвиг, так и логический (беззнаковый) сдвиг. Если number в number выражения number >> x

имеет тип подписи и отрицательное значение, результирующее значение определяется реализацией.

Если нам нужно значение этого бита непосредственно на месте, мы могли бы вместо этого сдвинуть маску:

```
(number & (1LL << x)); // (1 << x) if the 'x'th bit of 'number' is set, 0 otherwise
```

Либо можно использовать как условие, так как все ненулевые значения считаются истинными.

Использование std :: bitset

```
std::bitset<4> num(std::string("0010"));  
bool bit_val = num.test(1); // bit_val value is set to true;
```

Изменение n-го бита на x

С-стиль бит-манипуляции

```
// Bit n will be set if x is 1 and cleared if x is 0.  
number ^= (-x ^ number) & (1LL << n);
```

Использование std :: bitset

set(n, val) - устанавливает бит n в значение val .

```
std::bitset<5> num(std::string("00100"));  
num.set(0,true); // num is now 00101  
num.set(2,false); // num is now 00001
```

Установить все биты

С-стиль бит-манипуляции

```
x = -1; // -1 == 1111 1111 ... 1111b
```

(См. [Здесь](#), чтобы объяснить, почему это работает и на самом деле лучший подход.)

Использование std :: bitset

```
std::bitset<10> x;  
x.set(); // Sets all bits to '1'
```

Снять правый бит

С-стиль бит-манипуляции

```
template <typename T>  
T rightmostSetBitRemoved(T n)  
{  
    // static_assert(std::is_integral<T>::value && !std::is_signed<T>::value, "type should be  
    unsigned"); // For c++11 and later  
    return n & (n - 1);  
}
```

объяснение

- если n равно нулю, то $0 \& 0xFF..FF$ которое равно нулю
- else n может быть записано $0bxxxxxx10..00$ а $n - 1 = 0bxxxxxx011..11$, поэтому $n \& (n - 1)$ равно $0bxxxxxx000..00$.

Набор счетных битов

В криптографии и других приложениях часто требуется подсчет численности битовой строки, и проблема широко изучена.

Наивный способ требует одной итерации на бит:

```
unsigned value = 1234;  
unsigned bits = 0; // accumulates the total number of bits set in `n`  
  
for (bits = 0; value; value >>= 1)  
    bits += value & 1;
```

Хороший трюк (основанный на « [Удалить самый правый бит](#) »):

```
unsigned bits = 0; // accumulates the total number of bits set in `n`  
  
for (; value; ++bits)  
    value &= value - 1;
```

Он проходит через столько итераций, сколько существует набор битов, так что это хорошо, когда `value`, как ожидается, несколько отличных от нуля бит.

Этот метод был впервые предложен Петром Вегнером (в [CACM 3/322 - 1960](#)), и он хорошо известен, так как он появляется на *языке программирования C* Брайаном Керниганом и Деннисом М. Ричи.

Для этого требуется 12 арифметических операций, один из которых представляет собой мультипликацию:

```
unsigned popcount(std::uint64_t x)
{
    const std::uint64_t m1 = 0x5555555555555555; // binary: 0101...
    const std::uint64_t m2 = 0x3333333333333333; // binary: 00110011..
    const std::uint64_t m4 = 0x0f0f0f0f0f0f0f0f; // binary: 0000111100001111

    x -= (x >> 1) & m1;           // put count of each 2 bits into those 2 bits
    x = (x & m2) + ((x >> 2) & m2); // put count of each 4 bits into those 4 bits
    x = (x + (x >> 4)) & m4;       // put count of each 8 bits into those 8 bits
    return (x * h01) >> 56; // left 8 bits of x + (x<<8) + (x<<16) + (x<<24) + ...
}
```

Такая реализация имеет лучшее худшее поведение (см. [Вес Хэмминга](#) для получения дополнительной информации).

Многие процессоры имеют определенную инструкцию (например, x86's `popcnt`), и компилятор может предложить определенную (**не стандартную**) встроенную функцию. Например, с `g++` есть:

```
int __builtin_popcount (unsigned x);
```

Проверьте, является ли целое число мощностью 2

`n & (n - 1)` трюк (см. « [Удалить самый правый бит](#) ») также полезен для определения того, является ли целое число мощностью 2:

```
bool power_of_2 = n && !(n & (n - 1));
```

Обратите внимание, что без первой части проверки (`n &&`) 0 неверно считается степенью 2.

Бит-манипулирование Приложение: от малого до крупного письма

Одним из нескольких применений манипуляции бит является преобразование буквы от малого к капиталу или наоборот, выбирая **маску** и правильную **операцию бит**. Например, буква имеет это бинарное представление `01(1)00001`, а его капитал аналог имеет `01(0)00001`. Они отличаются только в бит в скобках. В этом случае преобразование письма от маленьких до столицы, в основном установка бита в скобках к одному. Для этого мы делаем следующее:

```

/*****
convert small letter to captial letter.
=====
      a: 01100001
      mask: 11011111  <-- (0xDF)  11(0)11111
      :-----
a&mask: 01000001  <-- A letter
*****/

```

Код для преобразования буквы в букву А

```

#include <cstdio>

int main()
{
    char op1 = 'a'; // "a" letter (i.e. small case)
    int mask = 0xDF; // choosing a proper mask

    printf("a (AND) mask = A\n");
    printf("%c    &    0xDF = %c\n", op1, op1 & mask);

    return 0;
}

```

В результате

```

$ g++ main.cpp -o test1
$ ./test1
a (AND) mask = A
a    &    0xDF = A

```

Прочитайте Бит-манипуляция онлайн: <https://riptutorial.com/ru/cplusplus/topic/3016/бит-манипуляция>

глава 22: Более неопределенное поведение в C ++

Вступление

Дополнительные примеры того, как C ++ может пойти не так.

Продолжение от [неопределенного поведения](#)

Examples

Ссылаясь на нестатические элементы в списках инициализаторов

Ссылаясь на нестатические элементы в списках инициализаторов до того, как конструктор начал выполнение, может привести к неопределенному поведению. Это связано с тем, что в это время не все члены построены. Из стандартного проекта:

§12.7.1. Для объекта с нетривиальным конструктором, ссылаясь на любой нестатический член или базовый класс объекта перед тем, как конструктор начинает выполнение, приводит к неопределенному поведению.

пример

```
struct W { int j; };
struct X : public virtual W { };
struct Y {
    int *p;
    X x;
    Y() : p(&x.j) { // undefined, x is not yet constructed
    }
};
```

Прочитайте [Более неопределенное поведение в C ++ онлайн](#):

<https://riptutorial.com/ru/cplusplus/topic/9885/более-неопределенное-поведение-в-c-plusplus>

глава 23: Виртуальные функции участника

Синтаксис

- `virtual void f ();`
- `virtual void g () = 0;`
- // C ++ 11 или новее:
 - `virtual void h () override;`
 - `void i ()` переопределить;
 - `virtual void j () final;`
 - `void k () final;`

замечания

- Только нестатические, не-шаблонные функции-члены могут быть `virtual`.
- Если вы используете C ++ 11 или более позднюю версию, рекомендуется `override` при переопределении виртуальной функции-члена из базового класса.
- **Полиморфные базовые классы часто имеют виртуальные деструкторы, позволяющие удалить производный объект с помощью указателя на базовый класс**. Если деструктор не был виртуальным, такая операция приводит к **неопределенному поведению** [expr.delete] §5.3.5 / 3.

Examples

Использование переопределения с виртуальными в C ++ 11 и более поздних версиях

`override` спецификатора имеет особое значение в C ++ 11, если оно добавлено в конце сигнатуры функции. Это означает, что функция

- Переопределение функции, присутствующей в базовом классе &
- Функция базового класса является `virtual`

Не существует значимости `run time` выполнения этого спецификатора, как это в основном означает указание для компиляторов

Пример ниже демонстрирует изменение в поведении с нашим без использования переопределения.

Без `override` :

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // Y::f() will not override X::f() because it has a different signature,
    // but the compiler will accept the code (and silently ignore Y::f()).
    virtual void f(int a) { std::cout << a << "\n"; }
};
```

C override :

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // The compiler will alert you to the fact that Y::f() does not
    // actually override anything.
    virtual void f(int a) override { std::cout << a << "\n"; }
};
```

Обратите внимание, что `override` - это не ключевое слово, а специальный идентификатор, который может появляться только в сигнатурах функций. Во всех остальных случаях `override` прежнему может использоваться как идентификатор:

```
void foo() {
    int override = 1; // OK.
    int virtual = 2; // Compilation error: keywords can't be used as identifiers.
}
```

Виртуальные виртуальные функции-члены

C виртуальными функциями-членами:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // Specifying virtual again here is optional
    // because it can be inferred from X::f().
    virtual void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}
```

```
int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "Y::f()"
}
```

Без виртуальных функций-членов:

```
#include <iostream>

struct X {
    void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "X::f()"
}
```

Конечные виртуальные функции

С ++ 11 представил `final` спецификатор, который запрещает переопределение метода, если он появился в сигнатуре метода:

```
class Base {
public:
    virtual void foo() {
        std::cout << "Base::Foo\n";
    }
};

class Derived1 : public Base {
public:
    // Overriding Base::foo
    void foo() final {
        std::cout << "Derived1::Foo\n";
    }
};

class Derived2 : public Derived1 {
public:
    // Compilation error: cannot override final method
    virtual void foo() {
        std::cout << "Derived2::Foo\n";
    }
};
```

```
    }  
};
```

Спецификатор `final` может быть использован только с «виртуального» функции члена и не могут быть применены к функциям неvirtуальных членов

Как и в случае с `final`, существует также опция «переопределить» спецификатора, которая предотвращает переопределение `virtual` функций в производном классе.

`override` спецификаторов и `final` могут объединяться вместе, чтобы иметь желаемый эффект:

```
class Derived1 : public Base {  
public:  
    void foo() final override {  
        std::cout << "Derived1::Foo\n";  
    }  
};
```

Поведение виртуальных функций в конструкторах и деструкторах

Поведение виртуальных функций в конструкторах и деструкторах часто сбивает с толку при первом обнаружении.

```
#include <iostream>  
using namespace std;  
  
class base {  
public:  
    base() { f("base constructor"); }  
    ~base() { f("base destructor"); }  
  
    virtual const char* v() { return "base::v()"; }  
  
    void f(const char* caller) {  
        cout << "When called from " << caller << ", " << v() << " gets called.\n";  
    }  
};  
  
class derived : public base {  
public:  
    derived() { f("derived constructor"); }  
    ~derived() { f("derived destructor"); }  
  
    const char* v() override { return "derived::v()"; }  
};  
  
int main() {  
    derived d;  
}
```

Выход:

Когда вызывается из базового конструктора, вызывается `base :: v ()`.

Когда вызывается из производного конструктора, вызывается метод `output :: v ()`.

Когда вызывается из производного деструктора, вызывается производный `:: v ()`.

При вызове из базового деструктора вызывается `base :: v ()`.

Причиной этого является то, что производный класс может определять дополнительные элементы, которые еще не инициализированы (в случае с конструктором) или уже уничтожены (в случае деструктора), а вызов его функций-членов будет небезопасным. Поэтому при построении и уничтожении объектов C++ *динамический* тип `*this` считается классом конструктора или деструктора, а не более производным классом.

Пример:

```
#include <iostream>
#include <memory>

using namespace std;
class base {
public:
    base()
    {
        std::cout << "foo is " << foo() << std::endl;
    }
    virtual int foo() { return 42; }
};

class derived : public base {
    unique_ptr<int> ptr_;
public:
    derived(int i) : ptr_(new int(i*i)) { }
    // The following cannot be called before derived::derived due to how C++ behaves,
    // if it was possible... Kaboom!
    int foo() override { return *ptr_; }
};

int main() {
    derived d(4);
}
```

Чистые виртуальные функции

Мы также можем указать, что `virtual` функция является *чистой виртуальной* (абстрактной), добавляя `= 0` к объявлению. Классы с одной или несколькими чистыми виртуальными функциями считаются абстрактными и не могут быть созданы; могут быть созданы только производные классы, которые определяют или наследуют определения для всех чистых виртуальных функций.

```
struct Abstract {
    virtual void f() = 0;
};
```

```

struct Concrete {
    void f() override {}
};

Abstract a; // Error.
Concrete c; // Good.

```

Даже если функция указана как чистый виртуальный, ей может быть предоставлена реализация по умолчанию. Несмотря на это, функция все равно будет считаться абстрактной, и производные классы должны будут определить ее до того, как они будут созданы. В этом случае версия функции производного класса даже разрешена для вызова версии базового класса.

```

struct DefaultAbstract {
    virtual void f() = 0;
};
void DefaultAbstract::f() {}

struct WhyWouldWeDoThis : DefaultAbstract {
    void f() override { DefaultAbstract::f(); }
};

```

Есть несколько причин, почему мы могли бы сделать это:

- Если мы хотим создать класс, который сам не может быть создан, но не препятствует созданию его производных классов, мы можем объявить деструктор чистым виртуальным. Будучи деструктором, он должен быть определен так или иначе, если мы хотим освободить экземпляр. И **поскольку деструктор, скорее всего, уже является виртуальным, чтобы предотвратить утечку памяти во время полиморфного использования**, мы не будем нести лишний удар производительности при объявлении другой `virtual` функции. Это может быть полезно при создании интерфейсов.

```

struct Interface {
    virtual ~Interface() = 0;
};
Interface::~~Interface() = default;

struct Implementation : Interface {};
// ~Implementation() is automatically defined by the compiler if not explicitly
// specified, meeting the "must be defined before instantiation" requirement.

```

- Если большинство или все реализации чистой виртуальной функции будут содержать повторяющийся код, этот код вместо этого может быть перенесен в версию базового класса, что упростит работу кода.

```

class SharedBase {
    State my_state;
    std::unique_ptr<Helper> my_helper;
    // ...

```

```

    public:
        virtual void config(const Context& cont) = 0;
        // ...
};
/* virtual */ void SharedBase::config(const Context& cont) {
    my_helper = new Helper(my_state, cont.relevant_field);
    do_this();
    and_that();
}

class OneImplementation : public SharedBase {
    int i;
    // ...

    public:
        void config(const Context& cont) override;
        // ...
};
void OneImplementation::config(const Context& cont) /* override */ {
    my_state = { cont.some_field, cont.another_field, i };
    SharedBase::config(cont);
    my_unique_setup();
};

// And so on, for other classes derived from SharedBase.

```

Прочитайте **Виртуальные функции участника онлайн:**

<https://riptutorial.com/ru/cplusplus/topic/1752/виртуальные-функции-участника>

глава 24: Возвращение нескольких значений из функции

Вступление

Существует много ситуаций, когда полезно возвращать несколько значений из функции: например, если вы хотите ввести элемент и вернуть цену и количество на складе, эта функция может быть полезна. Есть много способов сделать это на C++, и большинство из них связано с STL. Однако, если вы хотите избежать STL по какой-то причине, есть еще несколько способов сделать это, включая `structs/classes` и `arrays`.

Examples

Использование выходных параметров

Параметры могут использоваться для возврата одного или нескольких значений; эти параметры должны быть `const` указателями или ссылками.

Рекомендации:

```
void calculate(int a, int b, int& c, int& d, int& e, int& f) {  
    c = a + b;  
    d = a - b;  
    e = a * b;  
    f = a / b;  
}
```

указатели:

```
void calculate(int a, int b, int* c, int* d, int* e, int* f) {  
    *c = a + b;  
    *d = a - b;  
    *e = a * b;  
    *f = a / b;  
}
```

Некоторые библиотеки или фреймворки используют пустой 'OUT' `#define` чтобы сделать его совершенно очевидным, какие параметры являются выходными параметрами в сигнатуре функции. Это не имеет никакого функционального воздействия и будет скомпилировано, но делает подпись функции более понятной;

```
#define OUT  
  
void calculate(int a, int b, OUT int& c) {  
    c = a + b;  
}
```

```
}
```

Использование `std::tuple`

C++ 11

Тип `std::tuple` может связывать любое количество значений, потенциально включающих значения разных типов, в один возвращаемый объект:

```
std::tuple<int, int, int, int> foo(int a, int b) { // or auto (C++14)
    return std::make_tuple(a + b, a - b, a * b, a / b);
}
```

В C++ 17 можно использовать список упрощенных инициализаторов:

C++ 17

```
std::tuple<int, int, int, int> foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}
```

Получение значений из возвращаемого `tuple` может быть громоздким, требуя использования функции `std::get` template:

```
auto mrvs = foo(5, 12);
auto add = std::get<0>(mrvs);
auto sub = std::get<1>(mrvs);
auto mul = std::get<2>(mrvs);
auto div = std::get<3>(mrvs);
```

Если типы могут быть объявлены до возвращения функции, то `std::tie` можно использовать для распаковки `tuple` в существующие переменные:

```
int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 12);
```

Если одно из возвращаемых значений не требуется, можно использовать `std::ignore`:

```
std::tie(add, sub, std::ignore, div) = foo(5, 12);
```

C++ 17

Структурные привязки можно использовать, чтобы избежать `std::tie`:

```
auto [add, sub, mul, div] = foo(5, 12);
```

Если вы хотите вернуть кортеж ссылок lvalue вместо кортежа значений, используйте `std::tie` ВМЕСТО `std::make_tuple`.

```
std::tuple<int&, int&> minmax( int& a, int& b ) {
    if (b<a)
        return std::tie(b,a);
    else
        return std::tie(a,b);
}
```

который разрешает

```
void increase_least(int& a, int& b) {
    std::get<0>(minmax(a,b))++;
}
```

В некоторых редких случаях вы будете использовать `std::forward_as_tuple` вместо `std::tie`; будьте осторожны, если вы это сделаете, поскольку временные периоды могут длиться недостаточно долго, чтобы их можно было потреблять.

Использование `std::array`

C ++ 11

Контейнер `std::array` может связывать фиксированное число возвращаемых значений. Этот номер должен быть известен во время компиляции, и все возвращаемые значения должны быть одного типа:

```
std::array<int, 4> bar(int a, int b) {
    return { a + b, a - b, a * b, a / b };
}
```

Это заменяет с массивами стиля формы `int bar[4]`. Преимущество состоит в том, что теперь можно использовать различные функции `std` c++. Он также содержит полезные функции - члены, как `at` которым является безопасной функцией доступа члена с связанной проверкой, и `size`, который позволяет вернуть размер массива без расчета.

Использование `std::pair`

Структурный шаблон `std::pair` может связывать *ровно* два возвращаемых значения любых двух типов:

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return std::make_pair(a+b, a-b);
}
```

С C ++ 11 или более поздней `std::make_pair` вместо `std::make_pair` можно использовать список инициализаторов:

C ++ 11

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return {a+b, a-b};
}
```

Отдельные значения возвращенной `std::pair` могут быть получены с использованием `first` и `second` объектов-членов пары:

```
std::pair<int, int> mrvs = foo(5, 12);
std::cout << mrvs.first + mrvs.second << std::endl;
```

Выход:

10

Использование структуры

`struct` можно использовать для объединения нескольких возвращаемых значений:

C ++ 11

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
};

foo_return_type foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}

auto calc = foo(5, 12);
```

C ++ 11

Вместо назначения отдельным полям конструктор может использоваться для упрощения построения возвращаемых значений:

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
    foo_return_type(int add, int sub, int mul, int div)
        : add(add), sub(sub), mul(mul), div(div) {}
};

foo_return_type foo(int a, int b) {
    return foo_return_type(a + b, a - b, a * b, a / b);
}

foo_return_type calc = foo(5, 12);
```

Отдельные результаты, возвращаемые функцией `foo()` могут быть получены путем доступа к переменным-членам `struct calc`:

```
std::cout << calc.add << ' ' << calc.sub << ' ' << calc.mul << ' ' << calc.div << '\n';
```

Выход:

17 -7 60 0

Примечание. При использовании `struct` возвращаемые значения группируются вместе в одном объекте и доступны с использованием значимых имен. Это также помогает уменьшить количество посторонних переменных, созданных в области возвращаемых значений.

C ++ 17

Чтобы распаковать `struct` возвращаемую функцией, можно использовать [структурированные привязки](#). Это приводит к тому, что параметры вывода на четной основе имеют параметры:

```
int a=5, b=12;
auto[add, sub, mul, div] = foo(a, b);
std::cout << add << ' ' << sub << ' ' << mul << ' ' << div << '\n';
```

Результат этого кода идентичен выходу. `struct` по-прежнему используется для возврата значения из функции. Это позволяет вам обрабатывать поля отдельно.

Структурированные привязки

C ++ 17

В C ++ 17 реализованы структурированные привязки, что упрощает работу с несколькими типами возвращаемых данных, так как вам не нужно полагаться на `std::tie()` или выполнять ручную распаковку:

```
std::map<std::string, int> m;

// insert an element into the map and check if insertion succeeded
auto [iterator, success] = m.insert({"Hello", 42});

if (success) {
    // your code goes here
}

// iterate over all elements without having to use the cryptic 'first' and 'second' names
for (auto const& [key, value] : m) {
    std::cout << "The value for " << key << " is " << value << '\n';
}
```

Структурированные привязки могут использоваться по умолчанию с помощью `std::pair`,

`std::tuple` и любого типа, чьи нестатические члены данных являются либо публичными прямыми членами, либо членами недвусмысленного базового класса:

```
struct A { int x; };
struct B : A { int y; };
B foo();

// with structured bindings
const auto [x, y] = foo();

// equivalent code without structured bindings
const auto result = foo();
auto& x = result.x;
auto& y = result.y;
```

Если вы сделаете свой тип «tuple-like», он также автоматически будет работать с вашим типом. Кортеж, как тип с соответствующим `tuple_size`, `tuple_element` и `get` написано:

```
namespace my_ns {
    struct my_type {
        int x;
        double d;
        std::string s;
    };
    struct my_type_view {
        my_type* ptr;
    };
}

namespace std {
    template<>
    struct tuple_size<my_ns::my_type_view> : std::integral_constant<std::size_t, 3>
    {};

    template<> struct tuple_element<my_ns::my_type_view, 0>{ using type = int; };
    template<> struct tuple_element<my_ns::my_type_view, 1>{ using type = double; };
    template<> struct tuple_element<my_ns::my_type_view, 2>{ using type = std::string; };
}

namespace my_ns {
    template<std::size_t I>
    decltype(auto) get(my_type_view const& v) {
        if constexpr (I == 0)
            return v.ptr->x;
        else if constexpr (I == 1)
            return v.ptr->d;
        else if constexpr (I == 2)
            return v.ptr->s;
        static_assert(I < 3, "Only 3 elements");
    }
}
```

теперь это работает:

```
my_ns::my_type t{1, 3.14, "hello world"};

my_ns::my_type_view foo() {
```

```

    return {&t};
}

int main() {
    auto[x, d, s] = foo();
    std::cout << x << ',' << d << ',' << s << '\n';
}

```

Использование функции Object Consumer

Мы можем предоставить потребителю, который будет вызываться с несколькими соответствующими значениями:

C ++ 11

```

template <class F>
void foo(int a, int b, F consumer) {
    consumer(a + b, a - b, a * b, a / b);
}

// use is simple... ignoring some results is possible as well
foo(5, 12, [](int sum, int , int , int ){
    std::cout << "sum is " << sum << '\n';
});

```

Это называется «[стиль продолжения прохождения](#)» .

Вы можете адаптировать функцию, возвращающую кортеж, в функцию продолжения прохождения через:

C ++ 17

```

template<class Tuple>
struct continuation {
    Tuple t;
    template<class F>
    decltype(auto) operator->*(F&& f)&&{
        return std::apply( std::forward<F>(f), std::move(t) );
    }
};

std::tuple<int,int,int,int> foo(int a, int b);

continuation(foo(5,12))->*[](int sum, auto&&...) {
    std::cout << "sum is " << sum << '\n';
};

```

причем более сложные версии записываются на C ++ 14 или C ++ 11.

Использование std :: vector

`std::vector` может быть полезен для возврата динамического числа переменных того же типа. Следующий пример использует `int` как тип данных, но `std::vector` может содержать любой тип, который тривиально можно копировать:

```

#include <vector>
#include <iostream>

// the following function returns all integers between and including 'a' and 'b' in a vector
// (the function can return up to std::vector::max_size elements with the vector, given that
// the system's main memory can hold that many items)
std::vector<int> fillVectorFrom(int a, int b) {
    std::vector<int> temp;
    for (int i = a; i <= b; i++) {
        temp.push_back(i);
    }
    return temp;
}

int main() {
    // assigns the filled vector created inside the function to the new vector 'v'
    std::vector<int> v = fillVectorFrom(1, 10);

    // prints "1 2 3 4 5 6 7 8 9 10 "
    for (int i = 0; i < v.size(); i++) {
        std::cout << v[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}

```

Использование выходного итератора

Несколько значений одного и того же типа могут быть возвращены путем передачи выходного итератора в функцию. Это особенно характерно для общих функций (таких как алгоритмы стандартной библиотеки).

Пример:

```

template<typename Incrementable, typename OutputIterator>
void generate_sequence(Incrementable from, Incrementable to, OutputIterator output) {
    for (Incrementable k = from; k != to; ++k)
        *output++ = k;
}

```

Пример использования:

```

std::vector<int> digits;
generate_sequence(0, 10, std::back_inserter(digits));
// digits now contains {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

```

Прочитайте [Возвращение нескольких значений из функции онлайн](https://riptutorial.com/ru/cplusplus/topic/487/возвращение-нескольких-значений-из-функции):

<https://riptutorial.com/ru/cplusplus/topic/487/возвращение-нескольких-значений-из-функции>

глава 25: Встроенные переменные

Вступление

Встроенная переменная разрешается определять в нескольких единицах перевода без нарушения [правила Единого определения](#). Если он многократно определен, компоновщик объединит все определения в один объект в конечной программе.

Examples

Определение статического члена данных в определении класса

Статический член данных класса может быть полностью определен в определении класса, если он объявлен `inline`. Например, следующий класс может быть определен в заголовке. До C ++ 17 было бы необходимо предоставить файл `.cpp` для определения определения `Foo::num_instances` чтобы он определялся только один раз, но в C ++ 17 несколько определений встроенной переменной `Foo::num_instances` все относятся к одному и тому же объекту `int`.

```
// warning: not thread-safe...
class Foo {
public:
    Foo() { ++num_instances; }
    ~Foo() { --num_instances; }
    inline static int num_instances = 0;
};
```

В качестве особого случая `constexpr` статических данных `constexpr` неявно встроен.

```
class MyString {
public:
    MyString() { /* ... */ }
    // ...
    static constexpr int max_size = INT_MAX / 2;
};
// in C++14, this definition was required in a single translation unit:
// constexpr int MyString::max_size;
```

Прочитайте Встроенные переменные онлайн: <https://riptutorial.com/ru/cplusplus/topic/9265/встроенные-переменные>

глава 26: Встроенные функции

Вступление

Функция, определенная с помощью `inline` спецификатора, является встроенной функцией. Встроенная функция может быть размножена без нарушения [правила Единого определения](#) и поэтому может быть определена в заголовке с внешней связью. Объявление функции `inline` подсказывает компилятору, что функция должна быть встроена во время генерации кода, но не дает гарантии.

Синтаксис

- `inline function_declaration`
- `inline function_definition`
- `class {function_definition};`

замечания

Обычно, если код, сгенерированный для функции, *достаточно* мал, это хороший кандидат, который должен быть встроен. Почему так? Если функция большая и встроена в цикл, для всех сделанных вызовов код большой функции будет дублироваться, что приведет к созданию раздутого бинарного размера. Но насколько мала?

Хотя встроенные функции кажутся отличным способом избежать накладных расходов на функционирование, следует отметить, что не все функции, отмеченные `inline`, встроены. Другими словами, когда вы говорите `inline`, это всего лишь подсказка для компилятора, а не порядок: компилятор не обязан встраивать функцию, ее можно игнорировать - большинство из них это делает. Современные компиляторы лучше делают такие оптимизации, что это ключевое слово теперь остается следствием прошлого, когда это предложение функции, вложенное программистом, было серьезно воспринято компиляторами. Даже функции, не отмеченные `inline`, встроены компилятором, когда он видит преимущество в этом.

Inline как директива привязки

Более практическое использование `inline` в современном C++ происходит от использования его в качестве директивы `linkage`. При *определении*, а не *объявлении* функции в заголовке, который будет включен в несколько источников, тогда каждая единица перевода будет иметь свою собственную копию этой функции, ведущую к нарушению [ODR](#) (одно определение правила); это правило грубо говорит, что может быть

только одно определение функции, переменной и т. д. Чтобы обойти это нарушение, обозначение определения функции `inline` неявно делает внутреннюю связь функции.

Вопросы и ответы

Когда следует писать ключевое слово 'inline' для функции / метода в C ++?

Только когда вы хотите, чтобы функция была определена в заголовке. Точнее, только когда определение функции может отображаться в нескольких единицах компиляции. Это хорошая идея, чтобы определить небольшие (как в одном лайнере) функции в файле заголовка, поскольку он дает компилятору больше информации для работы с оптимизацией кода. Это также увеличивает время компиляции.

Когда я не должен писать ключевое слово 'inline' для функции / метода в C ++?

Не добавляйте `inline` строку, если вы считаете, что ваш код будет работать быстрее, если компилятор сделает это.

Когда компилятор не знает, когда сделать функцию / метод встроенным?

Как правило, компилятор сможет сделать это лучше вас. Однако компилятор не имеет возможности встроить код, если он не имеет определения функции. В максимально оптимизированном коде обычно используются все частные методы, независимо от того, запрашиваете вы это или нет.

Смотрите также

- [Когда следует писать ключевое слово 'inline' для функции / метода?](#)
- [Есть ли еще возможность для встроенного?](#)

Examples

Объявление функции не-членной строки

```
inline int add(int x, int y);
```

Определение функции встроенной функции без члена

```
inline int add(int x, int y)
{
    return x + y;
}
```

Встроенные функции пользователя

```
// header (.hpp)
struct A
{
    void i_am_inlined()
    {
    }
};

struct B
{
    void i_am_NOT_inlined();
};

// source (.cpp)
void B::i_am_NOT_inlined()
{
}
```

Что такое функция вложения?

```
inline int add(int x, int y)
{
    return x + y;
}

int main()
{
    int a = 1, b = 2;
    int c = add(a, b);
}
```

В приведенном выше коде, когда `add` вложенный текст, полученный код станет чем-то вроде этого

```
int main()
{
    int a = 1, b = 2;
    int c = a + b;
}
```

Встроенную функцию нигде не видно, ее тело *встраивается* в тело вызывающего. Если бы `add` не встраивается, функция будет называться. Накладные расходы на вызов функции - например, создание нового [фрейма стека](#), копирование аргументов, создание локальных переменных, переход (потеря локальности ссылки и отсутствие промахов кэш-памяти) и т. Д. - должны быть понесены.

Прочитайте Встроенные функции онлайн: <https://riptutorial.com/ru/cplusplus/topic/7150/встроенные-функции>

глава 27: Вывод типа

Вступление

В этом разделе обсуждается тип ввода, включающий ключевое слово `auto` type, которое доступно на C ++ 11.

замечания

Обычно лучше объявлять `const`, `&` и `constexpr` всякий раз, когда вы используете `auto` если это требуется для предотвращения нежелательного поведения, такого как копирование или мутации. Эти дополнительные подсказки гарантируют, что компилятор не генерирует никаких других форм вывода. Также не рекомендуется использовать `auto` использование и должно использоваться только тогда, когда фактическое объявление очень длинное, особенно с шаблонами STL.

Examples

Тип данных: Авто

В этом примере показаны основные выводы типа, которые может выполнить компилятор.

```
auto a = 1;           // a = int
auto b = 2u;          // b = unsigned int
auto c = &a;          // c = int*
const auto d = c;     // d = const int*
const auto& e = b;    // e = const unsigned int&

auto x = a + b        // x = int, #compiler warning unsigned and signed

auto v = std::vector<int>; // v = std::vector<int>
```

Однако ключевое слово `auto` не всегда выполняет ожидаемый вывод типа без дополнительных подсказок для `&` или `const` или `constexpr`

```
// y = unsigned int,
// note that y does not infer as const unsigned int&
// The compiler would have generated a copy instead of a reference value to e or b
auto y = e;
```

Lambda auto

Автоматическое ключевое слово типа данных - удобный способ для программистов объявить лямбда-функции. Это помогает, сокращая количество текстовых программистов, которые нужно ввести, чтобы объявить указатель на функцию.

```

auto DoThis = [](int a, int b) { return a + b; };
//    Do this is of type (int)(*DoThis)(int, int)
//    else we would have to write this long
int(*pDoThis)(int, int)= [](int a, int b) { return a + b; };

auto c = Dothis(1, 2);    //    c = int
auto d = pDothis(1, 2);  //    d = int

//    using 'auto' shortens the definition for lambda functions

```

По умолчанию, если возвращаемый тип лямбда-функций не определен, он будет автоматически выведен из типов возвращаемых выражений.

Эти 3 - это в основном одно и то же

```

[](int a, int b) -> int  { return a + b; };
[](int a, int b) -> auto { return a + b; };
[](int a, int b) { return a + b; };

```

Петли и авто

В этом примере показано, как auto можно использовать для сокращения объявления типа для циклов

```

std::map<int, std::string> Map;
for (auto pair : Map)           //    pair = std::pair<int, std::string>
for (const auto pair : Map)      //    pair = const std::pair<int, std::string>
for (const auto& pair : Map)     //    pair = const std::pair<int, std::string>&
for (auto i = 0; i < 1000; ++i)  //    i = int
for (auto i = 0; i < Map.size(); ++i) //    Note that i = int and not size_t
for (auto i = Map.size(); i > 0; --i) //    i = size_t

```

Прочитайте Вывод типа онлайн: <https://riptutorial.com/ru/cplusplus/topic/8233/вывод-типа>

глава 28: Вызываемые объекты

Вступление

Вызываемые объекты - это совокупность всех структур C ++, которые могут использоваться как функция. На практике это все, что вы можете передать функции C ++ 17 STL invoke () или которые могут быть использованы в конструкторе std :: function, в том числе: указатели на функции, классы с оператором (), классы с неявным преобразованием, ссылки на функции, указатели на функции-члены, указатели на данные участника, lambdas. Вызываемые объекты используются во многих алгоритмах STL в качестве предиката.

замечания

Очень полезный разговор Стефана Т. Лававей ([<функциональный>: что нового и правильное использование](#)) ([слайды](#)) приводит к основанию этой документации.

Examples

Указатели функций

Указатели функций - это самый простой способ передачи функций, которые также могут использоваться в C. (Более подробную информацию см. [В документации по C](#)).

Для целей вызываемых объектов указатель функции может быть определен как:

```
typedef returnType(*name)(arguments);           // All
using name = returnType(*) (arguments);         // <= C++11
using name = std::add_pointer<returnType(arguments)>::type; // <= C++11
using name = std::add_pointer_t<returnType(arguments)>;    // <= C++14
```

Если бы мы использовали указатель функции для написания нашего собственного векторного сортировки, это выглядело бы так:

```
using LessThanFunctionPtr = std::add_pointer_t<bool(int, int)>;
void sortVectorInt(std::vector<int>&v, LessThanFunctionPtr lessThan) {
    if (v.size() < 2)
        return;
    if (v.size() == 2) {
        if (!lessThan(v.front(), v.back())) // Invoke the function pointer
            std::swap(v.front(), v.back());
        return;
    }
    std::sort(v, lessThan);
}
```

```
bool lessThanInt(int lhs, int rhs) { return lhs < rhs; }
sortVectorInt(vectorOfInt, lessThanInt); // Passes the pointer to a free function

struct GreaterThanInt {
    static bool cmp(int lhs, int rhs) { return lhs > rhs; }
};
sortVectorInt(vectorOfInt, &GreaterThanInt::cmp); // Passes the pointer to a static member
function
```

В качестве альтернативы мы могли бы вызвать указатель функции одним из следующих способов:

- `(*lessThan)(v.front(), v.back()) // All`
- `std::invoke(lessThan, v.front(), v.back()) // <= C++17`

Классы с оператором () (Функторы)

Каждый класс, который перегружает `operator()` может использоваться как объект функции. Эти классы могут быть написаны вручную (часто называются функторами) или автоматически генерируются компилятором, написав [Lambdas](#) из C++ 11 on.

```
struct Person {
    std::string name;
    unsigned int age;
};

// Functor which find a person by name
struct FindPersonByName {
    FindPersonByName(const std::string &name) : _name(name) {}

    // Overloaded method which will get called
    bool operator()(const Person &person) const {
        return person.name == _name;
    }
private:
    std::string _name;
};

std::vector<Person> v; // Assume this contains data
std::vector<Person>::iterator iFind =
    std::find_if(v.begin(), v.end(), FindPersonByName("Foobar"));
// ...
```

Поскольку функторы имеют свою личность, они не могут быть введены в `typedef`, и их необходимо принять с помощью аргумента шаблона. Определение `std::find_if` может выглядеть так:

```
template<typename Iterator, typename Predicate>
Iterator find_if(Iterator begin, Iterator end, Predicate &predicate) {
    for (Iterator i = begin, i != end, ++i)
        if (predicate(*i))
            return i;
    return end;
}
```


Начиная с C ++ 17, вызов предиката может быть выполнен с помощью invoke:

```
std::invoke(predicate, *i) .
```

Прочитайте Вызываемые объекты онлайн: <https://riptutorial.com/ru/cplusplus/topic/6073/вызываемые-объекты>

глава 29: вычет типа

замечания

В ноябре 2014 года Комитет по стандартизации C++ принял предложение N3922, которое устраняет правило исключения специального типа для автоматических и согласованных инициализаторов с использованием прямого синтаксиса инициализации. Это не является частью стандарта C++, но было реализовано некоторыми компиляторами.

Examples

Вычисление параметра шаблона для конструкторов

До C++ 17 вычитание шаблона не может вывести тип класса для вас в конструкторе. Он должен быть явно указан. Иногда, однако, эти типы могут быть очень громоздкими или (в случае лямбда), которые невозможно назвать, поэтому мы получили распространение типов фабрик (таких как `make_pair()`, `make_tuple()`, `back_inserter()` и т. Д.).

C++ 17

Это больше не нужно:

```
std::pair p(2, 4.5);      // std::pair<int, double>
std::tuple t(4, 3, 2.5); // std::tuple<int, int, double>
std::copy_n(vil.begin(), 3,
            std::back_inserter(vi2)); // constructs a back_inserter<std::vector<int>>
std::lock_guard lk(mtx); // std::lock_guard<decltype(mtx)>
```

Считается, что конструкторы выводят параметры шаблона класса, но в некоторых случаях этого недостаточно, и мы можем предоставить явные инструкции для вычитания:

```
template <class Iter>
vector<Iter, Iter> -> vector<typename iterator_traits<Iter>::value_type>

int array[] = {1, 2, 3};
std::vector v(std::begin(array), std::end(array)); // deduces std::vector<int>
```

Выделение типа шаблона

Синтаксис шаблонов

```
template<typename T>
void f(ParamType param);

f(expr);
```

Случай 1: `ParamType` является ссылкой или указателем, но не универсальной или прямой ссылкой. В этом случае подобная дедукция работает таким образом. Компилятор игнорирует ссылочную часть, если она существует в `expr`. Затем компилятор шаблонов совпадение `expr` типа «ы против `ParamType` к определившим `T`.

```
template<typename T>
void f(T& param);           //param is a reference

int x = 27;                 // x is an int
const int cx = x;           // cx is a const int
const int& rx = x;          // rx is a reference to x as a const int

f(x);                       // T is int, param's type is int&
f(cx);                      // T is const int, param's type is const int&
f(rx);                      // T is const int, param's type is const int&
```

Случай 2: `ParamType` является универсальной ссылкой или прямой ссылкой. В этом случае вывод типа такой же, как в случае 1, если `expr` является значением `r`. Если `expr` является `lvalue`, то и `T` и `ParamType` выводятся как значения `lvalue`.

```
template<typename T>
void f(T&& param);          // param is a universal reference

int x = 27;                 // x is an int
const int cx = x;           // cx is a const int
const int& rx = x;          // rx is a reference to x as a const int

f(x);                      // x is lvalue, so T is int&, param's type is also int&
f(cx);                     // cx is lvalue, so T is const int&, param's type is also const int&
f(rx);                     // rx is lvalue, so T is const int&, param's type is also const int&
f(27);                    // 27 is rvalue, so T is int, param's type is therefore int&&
```

Случай 3: `ParamType` - это ни указатель, ни ссылка. Если `expr` является ссылкой, эталонная часть игнорируется. Если `expr` является `const`, это также игнорируется. Если он является `volatile`, который также игнорируется при выводе типа `T`.

```
template<typename T>
void f(T param);           // param is now passed by value

int x = 27;                 // x is an int
const int cx = x;           // cx is a const int
const int& rx = x;          // rx is a reference to x as a const int

f(x);                      // T's and param's types are both int
f(cx);                     // T's and param's types are again both int
f(rx);                     // T's and param's types are still both int
```

Автоматическое удержание

C ++ 11

Вывод типа с использованием **ключевого слова** `auto` работает почти так же, как и

выделение типа шаблона. Ниже приведены несколько примеров:

```
auto x = 27;           // (x is neither a pointer nor a reference), x's type is int
const auto cx = x;     // (cx is neither a pointer nor a reference), cx's type is const int
const auto& rx = x;    // (rx is a non-universal reference), rx's type is a reference to a
const int

auto&& uref1 = x;      // x is int and lvalue, so uref1's type is int&
auto&& uref2 = cx;     // cx is const int and lvalue, so uref2's type is const int &
auto&& uref3 = 27;     // 27 is an int and rvalue, so uref3's type is int&&
```

Ниже перечислены различия:

```
auto x1 = 27;          // type is int, value is 27
auto x2(27);           // type is int, value is 27
auto x3 = { 27 };      // type is std::initializer_list<int>, value is { 27 }
auto x4{ 27 };         // type is std::initializer_list<int>, value is { 27 }
                        // in some compilers type may be deduced as an int with a
                        // value of 27. See remarks for more information.
auto x5 = { 1, 2.0 }   // error! can't deduce T for std::initializer_list<t>
```

Как вы можете видеть, используете ли вы принудительные инициализаторы, `auto` принудительно создает переменную типа `std::initializer_list<T>`. Если он не может вывести `T`, код отклоняется.

Когда `auto` используется как возвращаемый тип функции, он указывает, что функция имеет **возвращаемый тип возврата**.

```
auto f() -> int {
    return 42;
}
```

C ++ 14

C ++ 14 позволяет, помимо использования авто разрешенных в C ++ 11, следующих:

1. При использовании в качестве возвращаемого типа функции без возвращаемого возвращаемого типа указывает, что тип возвращаемого значения функции должен выводиться из операторов `return` в теле функции, если таковые имеются.

```
// f returns int:
auto f() { return 42; }
// g returns void:
auto g() { std::cout << "hello, world!\n"; }
```

2. При использовании в типе параметра лямбда определяется лямбда как **общая лямбда**.

```
auto triple = [](auto x) { return 3*x; };
const auto x = triple(42); // x is a const int with value 126
```

Специальная форма `decltype(auto)` выводит тип, используя правила вывода типа `decltype` а не `auto`.

```
int* p = new int(42);  
auto x = *p;           // x has type int  
decltype(auto) y = *p; // y is a reference to *p
```

В C++ 03 и ранее ключевое слово `auto` имело совершенно иное значение как **спецификатор класса хранения**, который был унаследован от C.

Прочитайте вычет типа онлайн: <https://riptutorial.com/ru/cplusplus/topic/7863/вычет-типа>

глава 30: Генерация случайных чисел

замечания

Генерация случайных чисел в C++ обеспечивается заголовком `<random>`. Этот заголовок определяет случайные устройства, псевдослучайные генераторы и распределения.

Случайные устройства возвращают случайные числа, предоставляемые операционной системой. Они должны либо использоваться для инициализации псевдослучайных генераторов, либо непосредственно для криптографических целей.

Псевдослучайные генераторы возвращают целочисленные псевдослучайные числа, основанные на их исходном семени. Диапазон псевдослучайных чисел обычно охватывает все значения неподписанного типа. Все псевдослучайные генераторы в стандартной библиотеке возвращают одинаковые числа для одного и того же начального семестра для всех платформ.

Распределения потребляют случайные числа от псевдослучайных генераторов или случайных устройств и производят случайные числа с необходимым распределением. Распределения не являются независимыми от платформы и могут создавать разные числа для тех же генераторов с одинаковыми начальными семенами на разных платформах.

Examples

Истинное генератор случайных значений

Чтобы генерировать истинные случайные значения, которые могут использоваться для криптографии, `std::random_device` должен использоваться как генератор.

```
#include <iostream>
#include <random>

int main()
{
    std::random_device crypto_random_generator;
    std::uniform_int_distribution<int> int_distribution(0,9);

    int actual_distribution[10] = {0,0,0,0,0,0,0,0,0,0};

    for(int i = 0; i < 10000; i++) {
        int result = int_distribution(crypto_random_generator);
        actual_distribution[result]++;
    }

    for(int i = 0; i < 10; i++) {
        std::cout << actual_distribution[i] << " ";
    }
}
```

```
    return 0;
}
```

`std::random_device` используется так же, как используется генератор псевдослучайных значений.

Однако `std::random_device` **может быть реализовано в терминах механизма псевдослучайного числа**, определенного для реализации, если для реализации не доступен недетерминированный источник (например, аппаратное устройство).

Обнаружение таких реализаций должно быть возможно через [функцию члена `entropy`](#) (которая возвращает ноль, когда генератор полностью детерминирован), но многие популярные библиотеки (как `libstdc++`, так и `LLVM LCC`), всегда возвращают ноль, даже если они используют высококачественную внешнюю случайность ,

Создание псевдослучайного числа

Генератор псевдослучайных чисел генерирует значения, которые можно угадать на основе ранее сгенерированных значений. Другими словами: он детерминирован. Не используйте генератор псевдослучайных чисел в ситуациях, когда требуется истинное случайное число.

```
#include <iostream>
#include <random>

int main()
{
    std::default_random_engine pseudo_random_generator;
    std::uniform_int_distribution<int> int_distribution(0, 9);

    int actual_distribution[10] = {0,0,0,0,0,0,0,0,0,0};

    for(int i = 0; i < 10000; i++) {
        int result = int_distribution(pseudo_random_generator);
        actual_distribution[result]++;
    }

    for(int i = 0; i <= 9; i++) {
        std::cout << actual_distribution[i] << " ";
    }

    return 0;
}
```

Этот код создает генератор случайных чисел и распределение, которое генерирует целые числа в диапазоне [0,9] с равным правдоподобием. Затем он подсчитывает, сколько раз каждый результат был сгенерирован.

Параметр шаблона `std::uniform_int_distribution<T>` указывает тип целого, который должен быть сгенерирован. Используйте `std::uniform_real_distribution<T>` для создания поплавок или удвоений.

Использование генератора для множества распределений

Генератор случайных чисел может (и должен) использоваться для нескольких распределений.

```
#include <iostream>
#include <random>

int main()
{
    std::default_random_engine pseudo_random_generator;
    std::uniform_int_distribution<int> int_distribution(0, 9);
    std::uniform_real_distribution<float> float_distribution(0.0, 1.0);
    std::discrete_distribution<int> rigged_dice({1,1,1,1,1,100});

    std::cout << int_distribution(pseudo_random_generator) << std::endl;
    std::cout << float_distribution(pseudo_random_generator) << std::endl;
    std::cout << (rigged_dice(pseudo_random_generator) + 1) << std::endl;

    return 0;
}
```

В этом примере определяется только один генератор. Затем он используется для генерации случайного значения в трех разных распределениях. Распределение `rigged_dice` будет генерировать значение от 0 до 5, но почти всегда генерирует 5, потому что вероятность создания 5 равна $100 / 105$.

Прочитайте Генерация случайных чисел онлайн: <https://riptutorial.com/ru/cplusplus/topic/1541/генерация-случайных-чисел>

глава 31: Дата и время использования заголовков

Examples

Измерение времени с использованием

`system_clock` может использоваться для измерения времени, прошедшего в течение некоторой части выполнения программы.

C ++ 11

```
#include <iostream>
#include <chrono>
#include <thread>

int main() {
    auto start = std::chrono::system_clock::now(); // This and "end"'s type is
std::chrono::time_point
    { // The code to test
        std::this_thread::sleep_for(std::chrono::seconds(2));
    }
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Elapsed time: " << elapsed.count() << "s";
}
```

В этом примере `sleep_for` был использован для того, чтобы сделать активный поток сна за период времени, измеренный в `std::chrono::seconds`, но код между фигурными скобками может быть любым вызовом функции, который занимает некоторое время для выполнения.

Найдите количество дней между двумя датами

В этом примере показано, как найти количество дней между двумя датами. Дата указана по году / месяцу / дню месяца и дополнительно к часу / минуте / секунде.

Программа рассчитывает количество дней в годах с 2000 года.

```
#include <iostream>
#include <string>
#include <chrono>
#include <ctime>

/**
 * Creates a std::tm structure from raw date.
 *
 * \param year (must be 1900 or greater)
 * \param month months since January - [1, 12]
```

```

* \param day day of the month - [1, 31]
* \param minutes minutes after the hour - [0, 59]
* \param seconds seconds after the minute - [0, 61] (until C++11) / [0, 60] (since C++11)
*
* Based on http://en.cppreference.com/w/cpp/chrono/c/tm
*/
std::tm CreateTmStruct(int year, int month, int day, int hour, int minutes, int seconds) {
    struct tm tm_ret = {0};

    tm_ret.tm_sec = seconds;
    tm_ret.tm_min = minutes;
    tm_ret.tm_hour = hour;
    tm_ret.tm_mday = day;
    tm_ret.tm_mon = month - 1;
    tm_ret.tm_year = year - 1900;

    return tm_ret;
}

int get_days_in_year(int year) {

    using namespace std;
    using namespace std::chrono;

    // We want results to be in days
    typedef duration<int, ratio_multiply<hours::period, ratio<24> >::type> days;

    // Create start time span
    std::tm tm_start = CreateTmStruct(year, 1, 1, 0, 0, 0);
    auto tms = system_clock::from_time_t(std::mktime(&tm_start));

    // Create end time span
    std::tm tm_end = CreateTmStruct(year + 1, 1, 1, 0, 0, 0);
    auto tme = system_clock::from_time_t(std::mktime(&tm_end));

    // Calculate time duration between those two dates
    auto diff_in_days = std::chrono::duration_cast<days>(tme - tms);

    return diff_in_days.count();
}

int main()
{
    for ( int year = 2000; year <= 2016; ++year )
        std::cout << "There are " << get_days_in_year(year) << " days in " << year << "\n";
}

```

Прочитайте Дата и время использования заголовков онлайн:

<https://riptutorial.com/ru/cplusplus/topic/3936/дата-и-время-использования--chrono--заголовки>

глава 32: Значение и справочная семантика

Examples

Поддержка глубокого копирования и перемещения

Если тип хочет иметь семантику значений, и ему нужно хранить объекты, которые динамически распределены, то при копировании операции тип должен будет выделять новые копии этих объектов. Он также должен сделать это для назначения копии.

Такой вид копирования называется «глубокой копией». Он эффективно принимает то, что в противном случае было бы ссылкой на семантику и превращало бы ее в семантику значений:

```
struct Inner {int i;};

const int NUM_INNER = 5;
class Value
{
private:
    Inner *array_; //Normally has reference semantics.

public:
    Value() : array_(new Inner[NUM_INNER]){}

    ~Value() {delete[] array_;}

    Value(const Value &val) : array_(new Inner[NUM_INNER])
    {
        for(int i = 0; i < NUM_INNER; ++i)
            array_[i] = val.array_[i];
    }

    Value &operator=(const Value &val)
    {
        for(int i = 0; i < NUM_INNER; ++i)
            array_[i] = val.array_[i];
        return *this;
    }
};
```

C ++ 11

Перемещение семантики допускает тип типа `Value` чтобы избежать по-настоящему копирования своих данных, на которые ссылаются. Если пользователь использует значение таким образом, чтобы спровоцировать ход, «скопированный» из объекта можно оставить пустым из данных, на которые он ссылается:

```

struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
    Inner *array_; //Normally has reference semantics.

public:
    Value() : array_(new Inner[NUM_INNER]){}

    //OK to delete even if nullptr
    ~Value() {delete[] array_;}

    Value(const Value &val) : array_(new Inner[NUM_INNER])
    {
        for(int i = 0; i < NUM_INNER; ++i)
            array_[i] = val.array_[i];
    }

    Value &operator=(const Value &val)
    {
        for(int i = 0; i < NUM_INNER; ++i)
            array_[i] = val.array_[i];
        return *this;
    }

    //Movement means no memory allocation.
    //Cannot throw exceptions.
    Value(Value &&val) noexcept : array_(val.array_)
    {
        //We've stolen the old value.
        val.array_ = nullptr;
    }

    //Cannot throw exceptions.
    Value &operator=(Value &&val) noexcept
    {
        //Clever trick. Since `val` is going to be destroyed soon anyway,
        //we swap his data with ours. His destructor will destroy our data.
        std::swap(array_, val.array_);
    }
};

```

Действительно, мы можем даже сделать такой тип не скопируемым, если мы хотим запретить глубокие копии, все еще позволяя перемещать объект.

```

struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
    Inner *array_; //Normally has reference semantics.

public:
    Value() : array_(new Inner[NUM_INNER]){}

    //OK to delete even if nullptr

```

```

~Value() {delete[] array_;}

Value(const Value &val) = delete;
Value &operator=(const Value &val) = delete;

//Movement means no memory allocation.
//Cannot throw exceptions.
Value(Value &&val) noexcept : array_(val.array_)
{
    //We've stolen the old value.
    val.array_ = nullptr;
}

//Cannot throw exceptions.
Value &operator=(Value &&val) noexcept
{
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
}
};

```

Мы можем даже применить правило Zero, используя `unique_ptr`:

```

struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
    unique_ptr<Inner []>array_; //Move-only type.

public:
    Value() : array_(new Inner[NUM_INNER]){}

    //No need to explicitly delete. Or even declare.
    ~Value() = default; {delete[] array_;}

    //No need to explicitly delete. Or even declare.
    Value(const Value &val) = default;
    Value &operator=(const Value &val) = default;

    //Will perform an element-wise move.
    Value(Value &&val) noexcept = default;

    //Will perform an element-wise move.
    Value &operator=(Value &&val) noexcept = default;
};

```

Определения

Тип имеет семантику значений, если наблюдаемое состояние объекта функционально отличается от всех других объектов этого типа. Это означает, что если вы копируете объект, у вас есть новый объект, и модификации нового объекта никоим образом не будут видны из старого объекта.

Большинство базовых типов C ++ имеют семантику значений:

```
int i = 5;
int j = i; //Copied
j += 20;
std::cout << i; //Prints 5; i is unaffected by changes to j.
```

В большинстве стандартных типов стандартной библиотеки также есть семантика значений:

```
std::vector<int> v1(5, 12); //array of 5 values, 12 in each.
std::vector<int> v2 = v1; //Copies the vector.
v2[3] = 6; v2[4] = 9;
std::cout << v1[3] << " " << v1[4]; //Writes "12 12", since v1 is unchanged.
```

Говорят, что тип имеет ссылочную семантику, если экземпляр этого типа может делиться своим наблюдаемым состоянием с другим объектом (внешним по отношению к нему), так что манипуляция одним объектом приведет к изменению состояния в другом объекте.

Указатели C ++ имеют семантику значений относительно того, к какому объекту они указывают, но имеют ссылочную семантику относительно *состояния* объекта, на который они указывают:

```
int *pi = new int(4);
int *pi2 = pi;
pi = new int(16);
assert(pi2 != pi); //Will always pass.

int *pj = pi;
*pj += 5;
std::cout << *pi; //Writes 9, since `pi` and `pj` reference the same object.
```

Ссылки на C ++ также имеют ссылочную семантику.

Прочитайте [Значение и справочная семантика онлайн](https://riptutorial.com/ru/cplusplus/topic/1955/значение-и-справочная-семантика):

<https://riptutorial.com/ru/cplusplus/topic/1955/значение-и-справочная-семантика>

глава 33: Идиома Pimpl

замечания

Pimpl идиома (p ointer к о сущ еmentation, иногда называют *непрозрачным указателем* или *чеширский кот техники*), сокращает время компиляции класса, переместив все свои личные элементы данных в структуры, определенной в файле .cpp.

Класс владеет указателем на реализацию. Таким образом, он может быть объявлен вперед, так что заголовочный файл не должен включать `#include` классы, которые используются в частных переменных-членах.

При использовании идиомы pimpl изменение частного элемента данных не требует перекомпиляции классов, которые зависят от него.

Examples

Основная идиома Pimpl

C ++ 11

В файле заголовка:

```
// widget.h

#include <memory> // std::unique_ptr
#include <experimental/propagate_const>

class Widget
{
public:
    Widget();
    ~Widget();
    void DoSomething();

private:
    // the pImpl idiom is named after the typical variable name used
    // ie, pImpl:
    struct Impl; // forward declaration
    std::experimental::propagate_const<std::unique_ptr< Impl >> pImpl; // ptr to actual
    implementation
};
```

В файле реализации:

```
// widget.cpp

#include "widget.h"
#include "reallycomplextypes.h" // no need to include this header inside widget.h
```

```

struct Widget::Impl
{
    // the attributes needed from Widget go here
    ReallyComplexType rct;
};

Widget::Widget() :
    pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;

void Widget::DoSomething()
{
    // do the stuff here with pImpl
}

```

В `pImpl` содержится состояние `Widget` (или часть / большая часть его). Вместо описания `Widget` состояния, которое отображается в файле заголовка, оно может быть раскрыто только в ходе реализации.

`pImpl` означает «указатель на реализацию». «Реальная» реализация `Widget` находится в `pImpl`.

Опасность: обратите внимание, что для работы с `unique_ptr ~Widget()` должен быть реализован в точке в файле, где `Impl` полностью виден. Вы можете `=default` там, но `if =default` где `Impl` не определено, программа может легко стать плохо сформированной, не требуется диагностика.

Прочитайте Идиома Pimpl онлайн: <https://riptutorial.com/ru/cplusplus/topic/2143/идиома-pimpl>

CString Длина: 24

Это показывает, что в MSVC каждый из расширенных символов считается единственным «символом», и эта платформа полностью поддерживает интернационализированные языки.

Следует отметить, однако, что это поведение необычно, эти международные символы хранятся внутри как Unicode и, следовательно, на самом деле составляют несколько байтов. Это может вызвать непредвиденные ошибки

В компиляторе GNC / GCC вывод программы:

Длина строки: 31

Длина строки: 36

CString Длина: 24

CString Длина: 26

Этот пример демонстрирует, что, хотя компилятор GCC, используемый на этой платформе (Linux), поддерживает эти расширенные символы, он также использует (*правильно*) несколько байтов для хранения отдельного символа.

В этом случае использование символов Unicode возможно, но программист должен проявлять большую осторожность, помня, что длина «строки» в этом сценарии - это **длина в байтах** , а не **длина в читаемых символах** .

Эти различия связаны с тем, как обрабатываются международные языки на основе каждой платформы, и что более важно, что строки C и C ++, используемые в этом примере, можно рассматривать **как массив байтов** , такой, что (для этого использования) **язык C ++ считает символ (char) должен быть одним байтом** .

Прочитайте Интернационализация в C ++ онлайн:

<https://riptutorial.com/ru/cplusplus/topic/5270/интернационализация-в-c-plusplus>

глава 35: Исключения

Examples

Исключение исключений

Блок `try/catch` используется для обнаружения исключений. Код в разделе `try` - это код, который может вызывать исключение, а код в выражении `(-ax)` `catch` обрабатывает исключение.

```
#include <iostream>
#include <string>
#include <stdexcept>

int main() {
    std::string str("foo");

    try {
        str.at(10); // access element, may throw std::out_of_range
    } catch (const std::out_of_range& e) {
        // what() is inherited from std::exception and contains an explanatory message
        std::cout << e.what();
    }
}
```

Несколько `catch` ключем могут использоваться для обработки нескольких типов исключений. Если присутствуют множественные предложения `catch`, механизм обработки исключений пытается сопоставить их **в порядке** их появления в коде:

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
} catch (const std::length_error& e) {
    std::cout << e.what();
} catch (const std::out_of_range& e) {
    std::cout << e.what();
}
```

Классы исключений, которые выведены из общего базового класса, могут быть пойманы с помощью одного предложения `catch` для общего базового класса. Вышеприведенный пример может заменить два предложения `catch` для `std::length_error` и `std::out_of_range` с одним предложением для `std::exception`:

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
}
```

```

} catch (const std::exception& e) {
    std::cout << e.what();
}

```

Поскольку предложения `catch` проверяются по порядку, не забудьте сначала написать более конкретные предложения `catch`, иначе ваш код обработки исключений никогда не будет вызван:

```

try {
    /* Code throwing exceptions omitted. */
} catch (const std::exception& e) {
    /* Handle all exceptions of type std::exception. */
} catch (const std::runtime_error& e) {
    /* This block of code will never execute, because std::runtime_error inherits
       from std::exception, and all exceptions of type std::exception were already
       caught by the previous catch clause. */
}

```

Другая возможность - обработчик `catch-all`, который поймает любой брошенный объект:

```

try {
    throw 10;
} catch (...) {
    std::cout << "caught an exception";
}

```

Исключение Rethrow (распространение)

Иногда вы хотите что-то сделать, за исключением того, что вы улавливаете (например, записываете в журнал или печатаете предупреждение) и позволяете ему переходить в верхнюю область обработки. Для этого вы можете восстановить любое исключение, которое вы поймаете:

```

try {
    ... // some code here
} catch (const SomeException& e) {
    std::cout << "caught an exception";
    throw;
}

```

Использование `throw;` без аргументов будет повторно выбрано текущее исключение.

C++ 11

Чтобы восстановить управляемую `std::exception_ptr`, стандартная библиотека C++ имеет функцию `rethrow_exception` которая может быть использована путем включения заголовка `<exception>` в вашу программу.

```

#include <iostream>
#include <string>
#include <exception>

```

```
#include <stdexcept>

void handle_eptr(std::exception_ptr eptr) // passing by value is ok
{
    try {
        if (eptr) {
            std::rethrow_exception(eptr);
        }
    } catch(const std::exception& e) {
        std::cout << "Caught exception \"" << e.what() << "\"\n";
    }
}

int main()
{
    std::exception_ptr eptr;
    try {
        std::string().at(1); // this generates an std::out_of_range
    } catch(...) {
        eptr = std::current_exception(); // capture
    }
    handle_eptr(eptr);
} // destructor for std::out_of_range called here, when the eptr is destructed
```

Функция Try Blocks В конструкторе

Единственный способ исключить исключение в списке инициализаторов:

```
struct A : public B
{
    A() try : B(), foo(1), bar(2)
    {
        // constructor body
    }
    catch (...)
    {
        // exceptions from the initializer list and constructor are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }

private:
    Foo foo;
    Bar bar;
};
```

Функция Try Block для регулярной функции

```
void function_with_try_block()
try
{
    // try block body
}
catch (...)
{
    // catch block body
}
```

Это эквивалентно

```
void function_with_try_block()
{
    try
    {
        // try block body
    }
    catch (...)
    {
        // catch block body
    }
}
```

Обратите внимание, что для конструкторов и деструкторов поведение отличается от того, что блок `catch` повторно выбрасывает исключение (пойманное, если в блоке блока `catch` нет другого броска).

Функция `main` разрешено иметь функцию попробовать блок , как и любой другой функции, но `main` функция попытка блока «s не перехватывать исключения , которые происходят во время строительства нелокальной статической переменной или уничтожения любой статической переменной. Вместо этого вызывается `std::terminate` .

Функция Try Blocks В деструкторе

```
struct A
{
    ~A() noexcept(false) try
    {
        // destructor body
    }
    catch (...)
    {
        // exceptions of destructor body are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }
};
```

Обратите внимание, что, хотя это возможно, нужно быть очень осторожным с бросанием из деструктора, как если бы деструктор, вызванный во время разматывания стека, выдает исключение, вызывается `std::terminate` .

Лучшая практика: бросать по значению, вызывать по ссылке `const`

В общем, считается хорошей практикой бросать по значению (а не по указателю), а ухватить ссылкой (`const`).

```
try {
    // throw new std::runtime_error("Error!");    // Don't do this!
    // This creates an exception object
```

```

// on the heap and would require you to catch the
// pointer and manage the memory yourself. This can
// cause memory leaks!

throw std::runtime_error("Error!");
} catch (const std::runtime_error& e) {
    std::cout << e.what() << std::endl;
}

```

Одной из причин, почему ловить по ссылке является хорошей практикой, является то, что она устраняет необходимость в восстановлении объекта при передаче в блок `catch` (или при распространении на другие блоки `catch`). Захват по ссылке также позволяет обрабатывать исключения из-за полиморфизма и избегать фрагментации объектов. Однако, если вы воссоздаете исключение (например, `throw e;` см. Пример ниже), вы все равно можете получить фрагмент объекта, потому что `throw e;` оператор делает копию исключения как любой тип объявлен:

```

#include <iostream>

struct BaseException {
    virtual const char* what() const { return "BaseException"; }
};

struct DerivedException : BaseException {
    // "virtual" keyword is optional here
    virtual const char* what() const { return "DerivedException"; }
};

int main(int argc, char** argv) {
    try {
        try {
            throw DerivedException();
        } catch (const BaseException& e) {
            std::cout << "First catch block: " << e.what() << std::endl;
            // Output ==> First catch block: DerivedException

            throw e; // This changes the exception to BaseException
                    // instead of the original DerivedException!
        }
    } catch (const BaseException& e) {
        std::cout << "Second catch block: " << e.what() << std::endl;
        // Output ==> Second catch block: BaseException
    }
    return 0;
}

```

Если вы уверены, что не собираетесь делать что-либо, чтобы изменить исключение (например, добавить информацию или изменить сообщение), `catching by const reference` позволяет компилятору сделать оптимизацию и повысить производительность. Но это все равно может привести к срачиванию объектов (как видно из приведенного выше примера).

Предупреждение. Не забывайте бросать непреднамеренные исключения в блоки `catch`, особенно связанные с распределением дополнительной памяти или ресурсов. Например, при построении `logic_error`, `runtime_error` или их подклассов может возникнуть `bad_alloc` из-

за `bad_alloc` памяти при копировании строки исключения, потоки ввода-вывода могут возникать при регистрации с соответствующими масками маски исключений и т. Д.

Вложенное исключение

C++ 11

Во время обработки исключений существует общий прецедент, когда вы обнаруживаете общее исключение из низкоуровневой функции (например, ошибку файловой системы или ошибку передачи данных) и бросаете более конкретное исключение высокого уровня, что указывает на то, что некоторые операции высокого уровня могут не выполняться (например, не удастся опубликовать фотографию в Интернете). Это позволяет обработке исключений реагировать на конкретные проблемы с операциями высокого уровня, а также позволяет, имея только сообщение об ошибке, программист найти место в приложении, в котором произошло исключение. Недостатком этого решения является то, что столбец исключений усечен и исходное исключение теряется. Это заставляет разработчиков вручную включать текст исходного исключения во вновь созданный.

Вложенные исключения направлены на решение проблемы путем присоединения исключения низкого уровня, которое описывает причину, к исключению высокого уровня, которое описывает, что это означает в данном конкретном случае.

`std::nested_exception` **ПОЗВОЛЯЕТ** `std::nested_exception` **ИСКЛЮЧЕНИЯ ИЗ-ЗА** `std::throw_with_nested` :

```
#include <stdexcept>
#include <exception>
#include <string>
#include <fstream>
#include <iostream>

struct MyException
{
    MyException(const std::string& message) : message(message) {}
    std::string message;
};

void print_current_exception(int level)
{
    try {
        throw;
    } catch (const std::exception& e) {
        std::cerr << std::string(level, ' ') << "exception: " << e.what() << '\n';
    } catch (const MyException& e) {
        std::cerr << std::string(level, ' ') << "MyException: " << e.message << '\n';
    } catch (...) {
        std::cerr << "Unkown exception\n";
    }
}

void print_current_exception_with_nested(int level = 0)
{

```



```

    try {
        throw;
    } catch (...) {
        print_current_exception(level);
    }
    try {
        throw;
    } catch (const std::nested_exception& nested) {
        try {
            nested.rethrow_nested();
        } catch (...) {
            print_current_exception_with_nested(level + 1); // recursion
        }
    } catch (...) {
        //Empty // End recursion
    }
}

// sample function that catches an exception and wraps it in a nested exception
void open_file(const std::string& s)
{
    try {
        std::ifstream file(s);
        file.exceptions(std::ios_base::failbit);
    } catch (...) {
        std::throw_with_nested(MyException{"Couldn't open " + s});
    }
}

// sample function that catches an exception and wraps it in a nested exception
void run()
{
    try {
        open_file("nonexistent.file");
    } catch (...) {
        std::throw_with_nested( std::runtime_error("run() failed") );
    }
}

// runs the sample function above and prints the caught exception
int main()
{
    try {
        run();
    } catch (...) {
        print_current_exception_with_nested();
    }
}

```

Возможный выход:

```

exception: run() failed
MyException: Couldn't open nonexistent.file
exception: basic_ios::clear

```

Если вы работаете только с исключениями, унаследованными из `std::exception`, код может быть даже упрощен.

STD :: `uncaught_exceptions`

C++ 17

C++ 17 вводит `int std::uncaught_exceptions()` (чтобы заменить ограниченный `bool std::uncaught_exception()`), чтобы узнать, сколько исключений в настоящее время не реализовано. Это позволяет классу определить, уничтожен ли он во время разворачивания стека или нет.

```
#include <exception>
#include <string>
#include <iostream>

// Apply change on destruction:
// Rollback in case of exception (failure)
// Else Commit (success)
class Transaction
{
public:
    Transaction(const std::string& s) : message(s) {}
    Transaction(const Transaction&) = delete;
    Transaction& operator=(const Transaction&) = delete;
    void Commit() { std::cout << message << ": Commit\n"; }
    void RollBack() noexcept(true) { std::cout << message << ": Rollback\n"; }

    // ...

    ~Transaction() {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            Commit(); // May throw.
        } else { // current stack unwinding
            RollBack();
        }
    }

private:
    std::string message;
    int uncaughtExceptionCount = std::uncaught_exceptions();
};

class Foo
{
public:
    ~Foo() {
        try {
            Transaction transaction("In ~Foo"); // Commit,
                                                // even if there is an uncaught exception
            //...
        } catch (const std::exception& e) {
            std::cerr << "exception/~Foo:" << e.what() << std::endl;
        }
    }
};

int main()
{
    try {
        Transaction transaction("In main"); // RollBack
```

```

    Foo foo; // ~Foo commit its transaction.
    //...
    throw std::runtime_error("Error");
} catch (const std::exception& e) {
    std::cerr << "exception/main:" << e.what() << std::endl;
}
}

```

Выход:

```

In ~Foo: Commit
In main: Rollback
exception/main:Error

```

Пользовательское исключение

Вы не должны бросать исходные значения в качестве исключений, вместо этого используйте один из стандартных классов исключений или создавайте свои собственные.

Наличие собственного класса исключений, унаследованного от `std::exception` - хороший способ обойти это. Вот специальный класс исключений, который непосредственно наследуется от `std::exception`:

```

#include <exception>

class Except: virtual public std::exception {

protected:

    int error_number;           ///< Error number
    int error_offset;           ///< Error offset
    std::string error_message;   ///< Error message

public:

    /** Constructor (C++ STL string, int, int).
     * @param msg The error message
     * @param err_num Error number
     * @param err_off Error offset
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        error_number(err_num),
        error_offset(err_off),
        error_message(msg)
    {}

    /** Destructor.
     * Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}

    /** Returns a pointer to the (constant) error description.
     * @return A pointer to a const char*. The underlying memory
     * is in possession of the Except object. Callers must
     * not attempt to free the memory.
     */

```

```

    */
    virtual const char* what() const throw () {
        return error_message.c_str();
    }

    /** Returns error number.
     * @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /**Returns error offset.
     * @return #error_offset
     */
    virtual int getErrorOffset() const throw() {
        return error_offset;
    }
};

```

Пример броска catch:

```

try {
    throw(Except("Couldn't do what you were expecting", -12, -34));
} catch (const Except& e) {
    std::cout<<e.what()
              <<"\nError number: "<<e.getErrorNumber()
              <<"\nError offset: "<<e.getErrorOffset();
}

```

Поскольку вы не просто бросаете сообщение об ошибке, а также некоторые другие значения, представляющие, что именно было в точности, ваша обработка ошибок становится намного более эффективной и значимой.

Есть класс исключений, который позволяет вам обрабатывать сообщения об ошибках красиво: `std::runtime_error`

Вы также можете унаследовать этот класс:

```

#include <stdexcept>

class Except: virtual public std::runtime_error {
protected:
    int error_number;           ///< Error number
    int error_offset;           ///< Error offset

public:
    /** Constructor (C++ STL string, int, int).
     * @param msg The error message
     * @param err_num Error number
     * @param err_off Error offset
     */

```

```

explicit
Except(const std::string& msg, int err_num, int err_off):
    std::runtime_error(msg)
{
    error_number = err_num;
    error_offset = err_off;

}

/** Destructor.
 * Virtual to allow for subclassing.
 */
virtual ~Except() throw () {}

/** Returns error number.
 * @return #error_number
 */
virtual int getErrorNumber() const throw() {
    return error_number;
}

/**Returns error offset.
 * @return #error_offset
 */
virtual int getErrorOffset() const throw() {
    return error_offset;
}

};

```

Обратите внимание, что я не переопределил функцию `what()` из базового класса (`std::runtime_error`), то есть мы будем использовать версию класса `base` `what()`. Вы можете переопределить его, если у вас есть дальнейшая повестка дня.

Прочитайте Исключения онлайн: <https://riptutorial.com/ru/cplusplus/topic/1354/исключения>

глава 36: Использование std :: unordered_map

Вступление

std :: unordered_map - это просто ассоциативный контейнер. Он работает с ключами и их картами. Ключ, когда имена идут, помогает иметь уникальность на карте. Пока отображаемое значение представляет собой только контент, связанный с ключом. Типы данных этого ключа и карты могут быть любыми предопределенными типами данных или определенными пользователем.

замечания

Как следует из названия, элементы в неупорядоченной карте не сохраняются в отсортированной последовательности. Они хранятся в соответствии с их хэш-значениями, и, следовательно, использование неупорядоченной карты имеет много преимуществ, таких как только O (1) для поиска любого элемента из него. Он также быстрее, чем другие контейнеры карт. Из примера видно, что его очень легко реализовать, поскольку оператор ([]) помогает нам напрямую обращаться к отображаемому значению.

Examples

Декларация и использование

Как уже упоминалось, вы можете объявить неупорядоченную карту любого типа. Давайте будем иметь неупорядоченную карту с именем first с типом string и integer.

```
unordered_map<string, int> first; //declaration of the map
first["One"] = 1; // [] operator used to insert the value
first["Two"] = 2;
first["Three"] = 3;
first["Four"] = 4;
first["Five"] = 5;

pair <string,int> bar = make_pair("Nine", 9); //make a pair of same type
first.insert(bar); //can also use insert to feed the values
```

Некоторые основные функции

```
unordered_map<data_type, data_type> variable_name; //declaration

variable_name[key_value] = mapped_value; //inserting values

variable_name.find(key_value); //returns iterator to the key value
```

```
variable_name.begin(); // iterator to the first element  
variable_name.end(); // iterator to the last + 1 element
```

Прочитайте [Использование std :: unordered_map](https://riptutorial.com/ru/cplusplus/topic/10540/использование-std-unordered-map) онлайн:

<https://riptutorial.com/ru/cplusplus/topic/10540/использование-std-unordered-map>

глава 37: Использование декларации

Вступление

Объявление `using` вводит одно имя в текущую область действия, которая ранее была объявлена в другом месте.

Синтаксис

- `using typename ( opt ) nested-name-specifier unqualified-id ;`
- использование `:: unqualified-id ;`

замечания

Объявление- *использование* отличается от [директивы using](#) , которая сообщает компилятору, чтобы он смотрел в определенном пространстве имен при поиске *любого* имени. *Директива-использование* начинается с `using namespace .`

Объявление- *использование* также отличается от объявления псевдонима, которое дает новое имя для существующего типа таким же образом, как [typedef](#) . Объявление псевдонима содержит знак равенства.

Examples

Импорт имен отдельно из пространства имен

После `using` используется для ввода имени `cout` из пространства имен `std` в область `main` функции, объект `std::cout` можно называть только `cout` .

```
#include <iostream>
int main() {
    using std::cout;
    cout << "Hello, world!\n";
}
```

Повторное использование элементов из базового класса, чтобы избежать скрытия имени

Если в области класса применяется *декларация использования*, разрешается только переопределять член базового класса. Например, `using std::cout` в классе не допускается.

Часто имя, о котором было объявлено, является тем, которое иначе было бы скрыто.

Например, в приведенном ниже коде `d1.foo` относится только к `Derived1::foo(const char*)` и произойдет ошибка компиляции. Функция `Base::foo(int)` скрыта вообще не рассматриваемой. Тем не менее, `d2.foo(42)` отлично, потому что *using-declaration* приносит `Base::foo(int)` в набор объектов с именем `foo` в `Derived2`. Затем поиск имени находит и `foo s`, и разрешение перегрузки выбирает `Base::foo`.

```
struct Base {
    void foo(int);
};
struct Derived1 : Base {
    void foo(const char*);
};
struct Derived2 : Base {
    using Base::foo;
    void foo(const char*);
};
int main() {
    Derived1 d1;
    d1.foo(42); // error
    Derived2 d2;
    d2.foo(42); // OK
}
```

Наследование конструкторов

C++ 11

В качестве специального случая *использование-объявления* в области класса может ссылаться на конструкторы прямого базового класса. Эти конструкторы затем наследуются производным классом и могут использоваться для инициализации производного класса.

```
struct Base {
    Base(int x, const char* s);
};
struct Derived1 : Base {
    Derived1(int x, const char* s) : Base(x, s) {}
};
struct Derived2 : Base {
    using Base::Base;
};
int main() {
    Derived1 d1(42, "Hello, world");
    Derived2 d2(42, "Hello, world");
}
```

В приведенном выше коде обе `Derived1` и `Derived2` имеют конструкторы, которые `Derived2` аргументы непосредственно соответствующему конструктору `Base`. `Derived1` выполняет переадресацию, а `Derived2`, используя функцию C++ 11 наследующих конструкторов, делает это неявно.

Прочитайте Использование декларации онлайн: <https://riptutorial.com/ru/cplusplus/topic/9301/>

глава 38: итераторы

Examples

С Итераторы (указатели)

```
// This creates an array with 5 values.
const int array[] = { 1, 2, 3, 4, 5 };

#ifdef BEFORE_CPP11

// You can use `sizeof` to determine how many elements are in an array.
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);

// Then you can iterate over the array by incrementing a pointer until
// it reaches past the end of our array.
for (const int* i = first; i < afterLast; ++i) {
    std::cout << *i << std::endl;
}

#else

// With C++11, you can let the STL compute the start and end iterators:
for (auto i = std::begin(array); i != std::end(array); ++i) {
    std::cout << *i << std::endl;
}

#endif
```

Этот код выводит числа от 1 до 5, по одному на каждую строку:

```
1
2
3
4
5
```

Нарушение этого

```
const int array[] = { 1, 2, 3, 4, 5 };
```

Эта строка создает новый целочисленный массив с 5 значениями. С - это просто указатели на память, где каждое значение хранится вместе в непрерывном блоке.

```
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);
```

Эти строки создают два указателя. Первому указателю присваивается значение указателя

массива, который является адресом первого элемента в массиве. Оператор `sizeof` используемый в массиве `C`, возвращает размер массива в байтах. Разделенный на размер элемента, это дает количество элементов в массиве. Мы можем использовать это, чтобы найти адрес блока *после* массива.

```
for (const int* i = first; i < afterLast; ++i) {
```

Здесь мы создаем указатель, который мы будем использовать в качестве итератора. Он инициализируется адресом первого элемента, который мы хотим перебрать, и мы будем продолжать итерацию, пока `i` меньше, чем `afterLast`, что означает, что пока `i` `afterLast` на адрес внутри `array`.

```
std::cout << *i << std::endl;
```

Наконец, в цикле мы можем получить доступ к значению, на которое указывает наш итератор `i`, разыменовывая его. Здесь оператор разыменовывания `*` возвращает значение по адресу в `i`.

обзор

Итераторы - это позиции

Итераторы - это средство навигации и работы над последовательностью элементов и являются обобщенным расширением указателей. Концептуально важно помнить, что итераторы - это позиции, а не элементы. Например, выполните следующую последовательность:

```
A B C
```

Последовательность содержит три элемента и четыре позиции

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
```

Элементы - это вещи в последовательности. Позиции - это места, где значительная операция может произойти с последовательностью. Например, один вставляет в позицию, *до* или *после* элемента `A`, а не в элемент. Даже удаление элемента (`erase(A)`) выполняется сначала, найдя его положение, а затем удаляя его.

От итераторов до значений

Чтобы преобразовать из позиции в значение, итератор *разыменован* :

```
auto my_iterator = my_vector.begin(); // position
auto my_value = *my_iterator; // value
```

Можно рассматривать итератор как разыменовывание значения, на которое он ссылается в последовательности. Это особенно полезно для понимания того, почему вы никогда не должны разыгрывать итератор `end()` в последовательности:

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
  ↑           ↑
  |           +-- An iterator here has no value. Do not dereference it!
+----- An iterator here dereferences to the value A.
```

Во всех последовательностях и контейнерах, найденных в стандартной библиотеке C++, `begin()` вернет итератор в первую позицию, а `end()` вернет итератор в прошлое (*не* последнюю позицию!). Следовательно, имена этих итераторов в алгоритмах часто помечены как `first` и `last` :

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
  ↑           ↑
  |           |
+- first      +- last
```

Также возможно получить итератор для *любой последовательности* , потому что даже пустая последовательность содержит по крайней мере одну позицию:

```
+---+
|   |
+---+
```

В пустой последовательности `begin()` и `end()` будут одинаковой позиции, и *ни одно из них не может быть разыменовано*:

```
+---+
|   |
+---+
  ↑
  |
+- empty_sequence.begin()
  |
+- empty_sequence.end()
```

Альтернативная визуализация итераторов заключается в том, что они отмечают позиции *между* элементами:

```

+---+---+---+
| A | B | C |
+---+---+---+
^   ^   ^
|   |   |
+- first  +- last

```

и разыменован итератора возвращает ссылку на элемент, идущий после итератора. В некоторых ситуациях, когда это мнение особенно полезно,

- операции `insert` будут вставлять элементы в позицию, указанную итератора,
- операции `erase` возвращают итератор, соответствующий той же позиции, что и тот,
- итератор и его соответствующий **обратный итератор** расположены в одной и той же позиции между элементами

Недействительные итераторы

Итератор становится *недействительным*, если (скажем, в ходе операции) его позиция больше не является частью последовательности. Недействительный итератор не может быть разыменован до тех пор, пока он не будет переназначен в действительную позицию. Например:

```

std::vector<int>::iterator first;
{
    std::vector<int> foo;
    first = foo.begin(); // first is now valid
} // foo falls out of scope and is destroyed
// At this point first is now invalid

```

Многие алгоритмы и функции членов последовательности в стандартной библиотеке C++ имеют правила, определяющие, когда итераторы недействительны. Каждый алгоритм отличается тем, как они обрабатывают (и недействительны) итераторы.

Навигация с итераторами

Как мы знаем, итераторы предназначены для навигации последовательностей. Чтобы сделать это, итератор должен перенести свое положение на протяжении всей последовательности. Итераторы могут продвигаться вперед в последовательности, а некоторые могут продвигаться вперед:

```

auto first = my_vector.begin();
++first; // advance the iterator 1 position
std::advance(first, 1); // advance the iterator 1 position
first = std::next(first); // returns iterator to the next element
std::advance(first, -1); // advance the iterator 1 position
backwards

```

```
first = std::next(first, 20);           // returns iterator to the element 20
position forward
first = std::prev(first, 5);           // returns iterator to the element 5
position backward
auto dist = std::distance(my_vector.begin(), first); // returns distance between two
iterators.
```

Обратите внимание, что второй аргумент `std::distance` должен быть доступен с первого (или, другими словами, `first` должен быть меньше или равен `second`).

Несмотря на то, что вы можете выполнять арифметические операции с итераторами, не все операции определены для всех типов итераторов. `a = b + 3;` будет работать для Итераторов случайного доступа, но не будет работать для форвардных или двунаправленных итераторов, которые все еще могут быть продвинуты на 3 позиции с чем-то вроде `b = a; ++b; ++b; ++b;`, Поэтому рекомендуется использовать специальные функции, если вы не уверены, что такое тип итератора (например, в функции шаблона, принимающей итератор).

Концепции итератора

Стандарт C++ описывает несколько различных концепций итератора. Они группируются в соответствии с тем, как они ведут себя в последовательности, на которую они ссылаются. Если вы знаете концепцию, то итератор *моделирует* (ведет себя как), вы можете быть уверены в поведении этого итератора *независимо от последовательности, к которой он принадлежит*. Они часто описываются в порядке от максимального до наименее ограничительного (поскольку следующая концепция итератора на шаг лучше, чем его предшественник):

- Итераторы ввода: могут быть разыменованы *только один раз* для каждой позиции. Может только продвигаться и только по одной позиции за раз.
- Forward Iterators: Итератор ввода, который можно разыменовывать сколько угодно раз.
- Двунаправленные итераторы: передний итератор, который также может перемещаться *назад* по одной позиции за раз.
- Итераторы случайного доступа: двунаправленный итератор, который может продвигать вперед или назад любое количество позиций за раз.
- Смежные итераторы (начиная с C++ 17): Итератор с произвольным доступом, который гарантирует, что базовые данные смежны в памяти.

Алгоритмы могут варьироваться в зависимости от концепции, моделируемой итераторами, которые они задают. Например, хотя `random_shuffle` может быть реализован для форвардных итераторов, может быть предоставлен более эффективный вариант, требующий итераторов с произвольным доступом.

Итераторские черты

Итераторные черты обеспечивают единообразный интерфейс со свойствами итераторов. Они позволяют вам извлекать значения, разницу, указатель, ссылочные типы, а также категорию итератора:

```
template<class Iter>
Iter find(Iter first, Iter last, typename std::iterator_traits<Iter>::value_type val) {
    while (first != last) {
        if (*first == val)
            return first;
        ++first;
    }
    return last;
}
```

Категория итератора может использоваться для специализации алгоритмов:

```
template<class BidirIt>
void test(BidirIt a, std::bidirectional_iterator_tag) {
    std::cout << "Bidirectional iterator is used" << std::endl;
}

template<class ForwIt>
void test(ForwIt a, std::forward_iterator_tag) {
    std::cout << "Forward iterator is used" << std::endl;
}

template<class Iter>
void test(Iter a) {
    test(a, typename std::iterator_traits<Iter>::iterator_category());
}
```

Категории итераторов - это, в основном, понятия итераторов, за исключением того, что Contiguous Iterators не имеют собственного тега, так как было обнаружено, что он нарушает код.

Обратные итераторы

Если мы хотим итерации назад через список или вектор, мы можем использовать `reverse_iterator`. Обратный итератор выполнен из двунаправленного или произвольного итератора доступа, который он хранит как элемент, к которому можно получить доступ через `base()`.

Чтобы `rbegin()` итерацию назад, используйте `rbegin()` и `rend()` как итераторы для конца коллекции и начало коллекции соответственно.

Например, для повторного итерационного использования:


```
std::vector<int> v{1, 2, 3, 4, 5};
for (std::vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it)
{
    cout << *it;
} // prints 54321
```

Обратный итератор может быть преобразован в итератор вперед через функцию-член `base()`. Связь заключается в том, что обратный итератор ссылается на один элемент за `base()` итератором:

```
std::vector<int>::reverse_iterator r = v.rbegin();
std::vector<int>::iterator i = r.base();
assert(&*r == &*(i-1)); // always true if r, (i-1) are dereferenceable
                        // and are not proxy iterators
```

```
+---+---+---+---+---+---+
|   | 1 | 2 | 3 | 4 | 5 |   |
+---+---+---+---+---+---+
      ↑      ↑                ↑      ↑
      |      |                |      |
rend() |          rbegin()    end()
      |          |          rbegin().base()
      |          |
begin()
rend().base()
```

При визуализации, где итераторы отмечают позиции между элементами, связь проще:

```
+---+---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 |   |
+---+---+---+---+---+---+
      ↑                ↑
      |                |
      |                |
      |                |
begin()                end()
rend()                rbegin()
rend().base()         rbegin().base()
```

Векторный Итератор

`begin` возвращает `iterator` в первый элемент в контейнере последовательности.

`end` возвращает `iterator` в первый элемент за конец.

Если векторный объект является `const`, то `begin` и `end` возвращают `const_iterator`. Если вы хотите, чтобы `const_iterator` возвращался, даже если ваш вектор не `const`, вы можете использовать `cbegin` и `cend`.

Пример:

```
#include <vector>
#include <iostream>
```

```
int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 }; //initialize vector using an initializer_list

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

Выход:

1 2 3 4 5

Итератор карты

Итератор к первому элементу в контейнере.

Если объект карты имеет const-квалификацию, функция возвращает `const_iterator`. В противном случае он возвращает `iterator`.

```
// Create a map and insert some values
std::map<char,int> mymap;
mymap['b'] = 100;
mymap['a'] = 200;
mymap['c'] = 300;

// Iterate over all tuples
for (std::map<char,int>::iterator it = mymap.begin(); it != mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
```

Выход:

a => 200
b => 100
c => 300

Итераторы потока

Итераторы потоков полезны, когда нам нужно прочитать последовательность или распечатать отформатированные данные из контейнера:

```
// Data stream. Any number of various whitespace characters will be OK.
std::istringstream istr("1\t 2      3 4");
std::vector<int> v;

// Constructing stream iterators and copying data from stream into vector.
std::copy(
    // Iterator which will read stream data as integers.
    std::istream_iterator<int>(istr),
    // Default constructor produces end-of-stream iterator.
    std::istream_iterator<int>(),
    std::back_inserter(v));
```

```
// Print vector contents.
std::copy(v.begin(), v.end(),
    //Will print values to standard output as integers delimited by " -- ".
    std::ostream_iterator<int>(std::cout, " -- "));
```

Пример программы будет печатать 1 -- 2 -- 3 -- 4 -- до стандартного вывода.

Напишите свой собственный итератор с поддержкой генератора

Общепринятая модель на других языках имеет функцию, которая создает «поток» объектов и может использовать цикл цикла для его перебора.

Мы можем моделировать это на C ++ как

```
template<class T>
struct generator_iterator {
    using difference_type=std::ptrdiff_t;
    using value_type=T;
    using pointer=T*;
    using reference=T;
    using iterator_category=std::input_iterator_tag;
    std::optional<T> state;
    std::function< std::optional<T>() > operation;
    // we store the current element in "state" if we have one:
    T operator*() const {
        return *state;
    }
    // to advance, we invoke our operation. If it returns a nullopt
    // we have reached the end:
    generator_iterator& operator++() {
        state = operation();
        return *this;
    }
    generator_iterator operator++(int) {
        auto r = *this;
        ++(*this);
        return r;
    }
    // generator iterators are only equal if they are both in the "end" state:
    friend bool operator==( generator_iterator const& lhs, generator_iterator const& rhs ) {
        if (!lhs.state && !rhs.state) return true;
        return false;
    }
    friend bool operator!=( generator_iterator const& lhs, generator_iterator const& rhs ) {
        return !(lhs==rhs);
    }
    // We implicitly construct from a std::function with the right signature:
    generator_iterator( std::function< std::optional<T>() > f ):operation(std::move(f))
    {
        if (operation)
            state = operation();
    }
    // default all special member functions:
    generator_iterator( generator_iterator && ) =default;
    generator_iterator( generator_iterator const& ) =default;
    generator_iterator& operator=( generator_iterator && ) =default;
```

```
generator_iterator& operator=( generator_iterator const& ) =default;  
generator_iterator() =default;  
};
```

живой пример .

Мы сохраняем сгенерированный элемент раньше, чтобы мы могли легче обнаружить, если мы уже в конце.

Поскольку функция итератора конечного генератора никогда не используется, мы можем создать ряд итераторов генератора, только скопировав `std::function` один раз. По умолчанию построенный итератор генератора сравнивается с равным самому себе и всем другим итераторам конечных генераторов.

Прочитайте итераторы онлайн: <https://riptutorial.com/ru/cplusplus/topic/473/итераторы>

глава 39: итерация

Examples

перерыв

Выскакивает из ближайшего замкнутого цикла или оператора `switch`.

```
// print the numbers to a file, one per line
for (const int num : num_list) {
    errno = 0;
    fprintf(file, "%d\n", num);
    if (errno == ENOSPC) {
        fprintf(stderr, "no space left on device; output will be truncated\n");
        break;
    }
}
```

Продолжить

Переход к концу самого маленького замкнутого контура.

```
int sum = 0;
for (int i = 0; i < N; i++) {
    int x;
    std::cin >> x;
    if (x < 0) continue;
    sum += x;
    // equivalent to: if (x >= 0) sum += x;
}
```

делать

Представляет [цикл do-while](#).

```
// Gets the next non-whitespace character from standard input
char read_char() {
    char c;
    do {
        c = getchar();
    } while (isspace(c));
    return c;
}
```

за

Представляет [цикл for](#) или, в C ++ 11 и более поздних, [цикл, основанный на диапазоне](#).

```
// print 10 asterisks
for (int i = 0; i < 10; i++) {
    putchar('*');
}
```

в то время как

Вводит [цикл while](#) .

```
int i = 0;
// print 10 asterisks
while (i < 10) {
    putchar('*');
    i++;
}
```

диапазон для цикла

```
std::vector<int> primes = {2, 3, 5, 7, 11, 13};

for(auto prime : primes) {
    std::cout << prime << std::endl;
}
```

Прочитайте итерация онлайн: <https://riptutorial.com/ru/cplusplus/topic/7841/итерация>

глава 40: Категории ценностей

Examples

Значение Категория Значение

Выражениям на C ++ присваивается определенная категория значений, основанная на результате этих выражений. Категории значений для выражений могут влиять на перегрузку функции C ++.

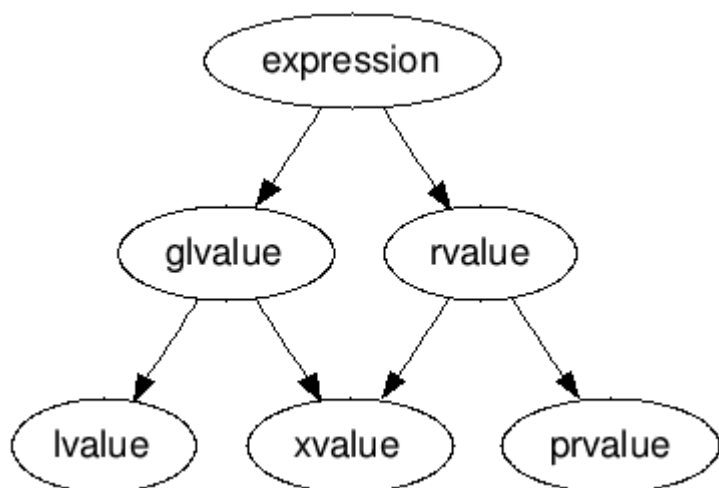
Категории значений определяют два важных, но отдельных свойства выражения. Одним из свойств является то, имеет ли выражение тождество. Выражение имеет личность, если оно относится к объекту с именем переменной. Имя переменной не может быть задействовано в выражении, но объект все еще может иметь его.

Другое свойство заключается в том, является ли законным неявное перемещение из значения выражения. Или, более конкретно, будет ли выражение при использовании в качестве параметра функции связываться с параметрами параметров r-значения или нет.

C ++ определяет 3 категории значений, которые представляют полезную комбинацию этих свойств: lvalue (выражения с идентичностью, но не движимые), xvalue (выражения с идентификацией, которые могут быть перемещены), и rvalue (выражения без идентификатора, которые можно перемещать). C ++ не имеет выражений, которые не имеют идентификатора и не могут быть перемещены.

C ++ определяет две другие категории значений, каждая из которых основана исключительно на одном из этих свойств: glvalue (выражения с идентификатором) и rvalue (выражения, которые можно перенести из). Они действуют как полезные группировки предыдущих категорий.

Этот график служит иллюстрацией:



prvalue

Выражение prvalue (pure-rvalue) - это выражение, которое не имеет идентичности, оценка которого обычно используется для инициализации объекта и из которого можно неявно перемещать. К ним относятся, но не ограничиваются:

- Выражения, представляющие временные объекты, такие как `std::string("123")` .
- Выражение вызова функции, которое не возвращает ссылку
- Литерал (*кроме* строкового литерала - это lvalues), например, `1` , `true` , `0.5f` или `'a'`
- Лямбда-выражение

Встроенный адрес оператора (`&`) не может применяться к этим выражениям.

xvalue

Выражение xvalue (eXpiring value) - это выражение, которое имеет идентификатор и представляет объект, из которого можно неявно перемещать. Общая идея с выражениями xvalue заключается в том, что объект, который они представляют, скоро будет уничтожен (следовательно, часть «eXpiring») и, следовательно, неявно перемещается от них в порядке.

Дано:

```
struct X { int n; };
extern X x;

4;                // prvalue: does not have an identity
x;                // lvalue
x.n;              // lvalue
std::move(x);     // xvalue
std::forward<X&>(x); // lvalue
X{4};             // prvalue: does not have an identity
X{4}.n;           // xvalue: does have an identity and denotes resources
                  // that can be reused
```

именующий

Выражение lvalue является выражением, которое имеет идентичность, но не может быть неявным образом перемещено. Среди них - выражения, которые состоят из имени переменной, имени функции, выражений, которые используются встроенным оператором разыменования и выражений, которые ссылаются на ссылки lvalue.

Типичная lvalue - это просто имя, но lvalues могут появляться и в других вкусах:

```
struct X { ... };

X x;           // x is an lvalue
X* px = &x;    // px is an lvalue
*px = X{};     // *px is also an lvalue, X{} is a prvalue
```



```
X* foo_ptr(); // foo_ptr() is a prvalue
X& foo_ref(); // foo_ref() is an lvalue
```

Кроме того, в то время как большинство литералов (например, `4`, `'x'` и т. Д.) Являются `prvalues`, строковые литералы являются `lvalues`.

glvalue

Выражение `glvalue` («обобщенное значение `lvalue`») представляет собой любое выражение, имеющее идентичность, независимо от того, может ли оно быть перемещено или нет. Эта категория включает `lvalues` (выражения, которые имеют идентичность, но не могут быть перемещены) и `xvalues` (выражения, которые имеют идентификатор и могут быть перемещены из), но исключают `prvalues` (выражения без идентификатора).

Если выражение имеет *имя*, это значение `glvalue`:

```
struct X { int n; };
X foo();

X x;
x; // has a name, so it's a glvalue
std::move(x); // has a name (we're moving from "x"), so it's a glvalue
               // can be moved from, so it's an xvalue not an lvalue

foo(); // has no name, so is a prvalue, not a glvalue
X{};   // temporary has no name, so is a prvalue, not a glvalue
X{}.n; // HAS a name, so is a glvalue. can be moved from, so it's an xvalue
```

Rvalue

Выражение `rvalue` - любое выражение, которое может быть неявно перемещено, независимо от того, имеет ли он идентификатор.

Более точно, выражения `rvalue` могут использоваться как аргумент функции, которая принимает параметр типа `T &&` (где `T` - тип `expr`). В качестве аргументов для таких параметров функции могут указываться *только* выражения `rvalue`; если используется выражение `non-rvalue`, то разрешение перегрузки будет выбирать любую функцию, которая не использует параметр ссылки `rvalue`. И если они не существуют, вы получите сообщение об ошибке.

Категория выражений `rvalue` включает все выражения `xvalue` и `prvalue` и только те выражения.

Стандартная функция библиотеки `std::move` существует, чтобы явно преобразовать выражение `non-rvalue` в `r`-значение. Более конкретно, оно превращает выражение в значение `x`, поскольку, даже если раньше оно было выражением без знака, передавая его как параметр для `std::move`, оно получает идентификатор (имя параметра функции) и

становится значением `xvalue`.

Рассмотрим следующее:

```
std::string str("init");           //1
std::string test1(str);             //2
std::string test2(std::move(str));  //3

str = std::string("new value");     //4
std::string &&str_ref = std::move(str); //5
std::string test3(str_ref);         //6
```

`std::string` имеет конструктор, который принимает единственный параметр типа `std::string&&`, обычно называемый «конструктором перемещения». Однако категория значений выражения `str` не является `rvalue` (в частности, это `lvalue`), поэтому она не может вызвать эту перегрузку конструктора. Вместо этого он вызывает `const std::string&` overload, конструктор копирования.

Строка 3 меняет ситуацию. Возвращаемое значение `std::move` - это `T&&`, где `T` - базовый тип переданного параметра. Таким образом `std::move(str)` возвращает `std::string&&`. Вызов функции, возвращающее значение, является ссылкой `rvalue`, является выражением `rvalue` (в частности, значением `xvalue`), поэтому он может вызывать конструктор перемещения `std::string`. После строки 3 `str` была перенесена с (содержимое пользователя теперь не определено).

Строка 4 передает временный оператор присваивания `std::string`. Это имеет перегрузку, которая принимает `std::string&&`. Выражение `std::string("new value")` является выражением `rvalue` (в частности, значением `prvalue`), поэтому оно может вызвать перегрузку. Таким образом, временное перемещается в `str`, заменяя неопределенное содержимое определенным содержимым.

Строка 5 создает ссылку с именем `rvalue`, называемую `str_ref` которая ссылается на `str`. Вот почему категории ценностей запутываются.

См., В то время как `str_ref` является ссылкой `rvalue` на `std::string`, категория значений выражения `str_ref` не является значением `rvalue`. Это выражение `lvalue`. Да, действительно. Из-за этого нельзя вызвать конструктор перемещения `std::string` с выражением `str_ref`. Поэтому строка 6 копирует значение `str` в `test3`.

Чтобы переместить его, нам пришлось бы использовать `std::move` again.

Прочитайте Категории ценностей онлайн: <https://riptutorial.com/ru/cplusplus/topic/763/категории-ценностей>

глава 41: Классы / Структуры

Синтаксис

- `variable.member_var = константа;`
- `variable.member_function ();`
- `variable_pointer-> member_var = constant;`
- `variable_pointer-> member_function ();`

замечания

Обратите внимание, что **единственная** разница между ключевыми словами `struct` и `class` заключается в том, что по умолчанию переменные-члены, функции-члены и базовые классы `struct` являются `public`, а в `class` они являются `private`. Программисты на C++, как правило, называют это классом, если у него есть конструкторы и деструкторы, а также возможность принудительно применять свои собственные инварианты; или `struct`, если это просто простой набор значений, но сам язык C++ не делает различий.

Examples

Основы классов

Класс является определяемым пользователем типом. Класс вводится с ключевым словом `class`, `struct` или `union`. В разговорной речи термин «класс» обычно относится только к непрофсоюзным классам.

Класс - это набор *членов класса*, который может быть:

- переменные-члены (также называемые «полями»)
- функции-члены (также называемые «методы»)
- типы членов или typedefs (например, «вложенные классы»)
- шаблоны членов (любого типа: шаблон переменной, функции, класса или псевдонима)

Ключевые слова `class` и `struct`, называемые *ключами классов*, в значительной степени взаимозаменяемы, за исключением того, что спецификатор доступа по умолчанию для членов и баз является «закрытым» для класса, объявленного с ключом `class` и «общедоступным» для класса, объявленного с помощью ключа `struct` или `union` (см. [модификаторы доступа](#)).

Например, следующие фрагменты кода идентичны:

```
struct Vector
{
```

```

    int x;
    int y;
    int z;
};
// are equivalent to
class Vector
{
public:
    int x;
    int y;
    int z;
};

```

Объявляя класс, в вашу программу добавляется новый тип, и можно создавать объекты этого класса посредством

```
Vector my_vector;
```

Доступ к членам класса осуществляется с помощью точечного синтаксиса.

```

my_vector.x = 10;
my_vector.y = my_vector.x + 1; // my_vector.y = 11;
my_vector.z = my_vector.y - 4; // my_vector.z = 7;

```

Спецификаторы доступа

В качестве **спецификаторов доступа** действуют три **ключевых слова**. Они ограничивают доступ к членам класса после спецификатора, пока другой спецификатор снова не изменит уровень доступа:

Ключевое слово	Описание
public	У каждого есть доступ
protected	Только сам класс, производные классы и друзья имеют доступ
private	Только у самого класса и друзей есть доступ

Когда тип определяется с помощью ключевого слова `class`, спецификатор доступа по умолчанию является `private`, но если тип определяется с помощью ключевого слова `struct`, спецификатор доступа по умолчанию является `public`:

```

struct MyStruct { int x; };
class MyClass { int x; };

MyStruct s;
s.x = 9; // well formed, because x is public

MyClass c;
c.x = 9; // ill-formed, because x is private

```

Спецификаторы доступа в основном используются для ограничения доступа к внутренним полям и методам и заставляют программиста использовать определенный интерфейс, например, чтобы принудительно использовать геттеры и сеттеры вместо прямого обращения к переменной:

```
class MyClass {  
  
    public: /* Methods: */  
  
        int x() const noexcept { return m_x; }  
        void setX(int const x) noexcept { m_x = x; }  
  
    private: /* Fields: */  
  
        int m_x;  
  
};
```

Использование `protected` полезно для того, чтобы определенные функциональные возможности этого типа были доступны только для производных классов, например, в следующем коде, метод `calculateValue()` доступен только для классов, полученных из базового класса `Plus2Base`, например `FortyTwo`:

```
struct Plus2Base {  
    int value() noexcept { return calculateValue() + 2; }  
protected: /* Methods: */  
    virtual int calculateValue() noexcept = 0;  
};  
struct FortyTwo: Plus2Base {  
protected: /* Methods: */  
    int calculateValue() noexcept final override { return 40; }  
};
```

Обратите внимание, что ключевое слово `friend` может использоваться для добавления исключений доступа к функциям или типам для доступа к защищенным и закрытым членам.

`public`, `protected` и `private` ключевые слова также могут использоваться для предоставления или ограничения доступа к подобъектам базового класса. См. Пример [Inheritance](#).

наследование

Классы / структуры могут иметь отношения наследования.

Если класс / структура `B` наследуется от класса / структуры `A`, это означает, что `B` имеет родительский `A`. Мы говорим, что `B` - производный класс / struct из `A`, а `A` - базовый класс / struct.

```
struct A  
{  
    public:
```

```

    int p1;
protected:
    int p2;
private:
    int p3;
};

//Make B inherit publicly (default) from A
struct B : A
{
};

```

Существует 3 формы наследования для класса / структуры:

- public
- private
- protected

Обратите внимание, что наследование по умолчанию такое же, как видимость членов по умолчанию: `public` если вы используете ключевое слово `struct`, и `private` для ключевого слова `class`.

Можно даже получить `class` из `struct` (или наоборот). В этом случае наследование по умолчанию контролируется дочерним элементом, поэтому `struct` которая выводится из `class`, по умолчанию будет иметь наследование по умолчанию, а `class`, полученный из `struct`, по умолчанию будет иметь частное наследование.

`public` наследование:

```

struct B : public A // or just `struct B : A`
{
    void foo()
    {
        p1 = 0; //well formed, p1 is public in B
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //well formed, p1 is public
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible

```

`private` наследство:

```

struct B : private A
{
    void foo()
    {
        p1 = 0; //well formed, p1 is private in B
        p2 = 0; //well formed, p2 is private in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

```

```

B b;
b.p1 = 1; //ill formed, p1 is private
b.p2 = 1; //ill formed, p2 is private
b.p3 = 1; //ill formed, p3 is inaccessible

```

protected наследование:

```

struct B : protected A
{
    void foo()
    {
        p1 = 0; //well formed, p1 is protected in B
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //ill formed, p1 is protected
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible

```

Обратите внимание, что хотя `protected` наследование разрешено, фактическое его использование встречается редко. Одним из примеров того, как `protected` наследование используется в приложении, является частная специализация базового класса (обычно называемая «контролируемый полиморфизм»).

Когда ООП был относительно новым, (публичное) наследование часто говорилось о моделировании отношения «IS-A». То есть, наследование является правильным, только если экземпляр производного класса *также является* экземпляром базового класса.

Это позже было уточнено в [Принципе замещения Лискова](#) : публичное наследование должно использоваться только тогда, когда / если экземпляр производного класса может быть заменен экземпляром базового класса при любых возможных обстоятельствах (и по-прежнему имеет смысл).

Как правило, частное наследство воплощает совершенно разные отношения: «реализуется в терминах» (иногда называемое отношением «HAS-A»). Например, класс `Stack` может наследовать конфиденциально из класса `Vector` . Частное наследование имеет гораздо большее сходство с агрегированием, чем государственное наследование.

Защищенное наследование почти никогда не используется, и нет общего согласия относительно того, какие отношения он воплощает.

Виртуальное наследование

При использовании наследования вы можете указать ключевое слово `virtual` :

```

struct A{};

```

```
struct B: public virtual A{;
```

Когда класс **B** имеет виртуальную базу **A** это означает, что **A** **будет находиться в большинстве производных классов** дерева наследования и, следовательно, большинство производных классов также отвечает за инициализацию этой виртуальной базы:

```
struct A
{
    int member;
    A(int param)
    {
        member = param;
    }
};

struct B: virtual A
{
    B(): A(5){}
};

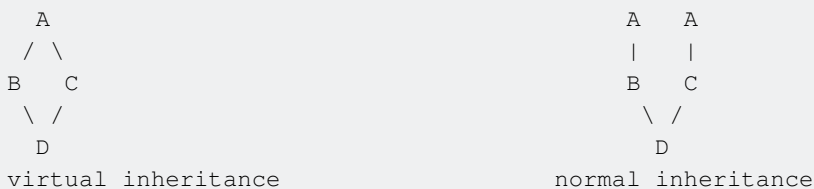
struct C: B
{
    C(): /*A(88)*/ {}
};

void f()
{
    C object; //error since C is not initializing it's indirect virtual base `A`
}
```

Если мы не будем комментировать `/*A(88)*/` мы не получим никакой ошибки, так как **C** теперь инициализирует ее косвенную виртуальную базу **A**

Также обратите внимание, что когда мы создаем переменный `object`, большинство производных классов **C**, поэтому **C** отвечает за создание (вызывающий конструктор) **A** и, следовательно, значение `A::member` равно `88`, а не `5` (как и было бы, если бы мы были создавая объект типа **B**).

Это полезно при решении [проблемы алмаза](#) :



B и **C**, как наследовать от **A** и **D** наследует от **B** и **C**, так **что 2 экземпляра A в D** ! Это приводит к двусмысленности при доступе к члену от **A** до **D**, поскольку компилятор не знает, из какого класса вы хотите получить доступ к этому члену (тот, который **B** наследует, или тот, который наследуется **C** ?),

Виртуальное наследование решает эту проблему: поскольку виртуальная база находится только в большинстве производных объектов, будет только один экземпляр `A` в `D`.

```
struct A
{
    void foo() {}
};

struct B : public /*virtual*/ A {};
struct C : public /*virtual*/ A {};

struct D : public B, public C
{
    void bar()
    {
        foo(); //Error, which foo? B::foo() or C::foo()? - Ambiguous
    }
};
```

Удаление комментариев устраняет двусмысленность.

Многократное наследование

Помимо одиночного наследования:

```
class A {};  
class B : public A {};
```

Вы также можете иметь множественное наследование:

```
class A {};  
class B {};  
class C : public A, public B {};
```

Теперь `C` будет наследовать от `A` и `B` одновременно.

Примечание. Это может привести к двусмысленности, если одни и те же имена используются в нескольких унаследованных *class* *s* или *struct* *s*. Быть осторожен!

Неоднозначность в множественном наследовании

Множественное наследование может быть полезным в некоторых случаях, но иногда нечетным видом проблем при использовании множественного наследования.

Например: у двух базовых классов есть функции с тем же именем, которые не переопределены в производном классе, и если вы пишете код для доступа к этой функции с использованием объекта производного класса, компилятор показывает ошибку, потому что он не может определить, какую функцию вызывать. Вот код для этого типа двусмысленности в множественном наследовании.

```

class base1
{
    public:
        void function( )
        { //code for base1 function }
};
class base2
{
    void function( )
    { // code for base2 function }
};

class derived : public base1, public base2
{

};

int main()
{
    derived obj;

    // Error because compiler can't figure out which function to call
    //either function( ) of base1 or base2 .
    obj.function( )
}

```

Но эту проблему можно решить с помощью функции разрешения разрешающей способности, чтобы указать, какая функция относится к классу base1 или base2:

```

int main()
{
    obj.base1::function( ); // Function of class base1 is called.
    obj.base2::function( ); // Function of class base2 is called.
}

```

Доступ к членам класса

Чтобы получить доступ к переменным - членам и функциям - членам объекта класса, то . используется оператор:

```

struct SomeStruct {
    int a;
    int b;
    void foo() {}
};

SomeStruct var;
// Accessing member variable a in var.
std::cout << var.a << std::endl;
// Assigning member variable b in var.
var.b = 1;
// Calling a member function.
var.foo();

```

При доступе к членам класса с помощью указателя обычно используется оператор -> .

Альтернативно, экземпляр может быть разыменован и . оператора, хотя это реже:

```
struct SomeStruct {
    int a;
    int b;
    void foo() {}
};

SomeStruct var;
SomeStruct *p = &var;
// Accessing member variable a in var via pointer.
std::cout << p->a << std::endl;
std::cout << (*p).a << std::endl;
// Assigning member variable b in var via pointer.
p->b = 1;
(*p).b = 1;
// Calling a member function via a pointer.
p->foo();
(*p).foo();
```

При доступе к статическим членам класса используется оператор :: , но не имя класса вместо экземпляра его. Альтернативно, статический член может быть доступен из экземпляра или указателя на экземпляр с помощью . или -> , соответственно, с тем же синтаксисом, что и доступ к нестационарным членам.

```
struct SomeStruct {
    int a;
    int b;
    void foo() {}

    static int c;
    static void bar() {}
};
int SomeStruct::c;

SomeStruct var;
SomeStruct* p = &var;
// Assigning static member variable c in struct SomeStruct.
SomeStruct::c = 5;
// Accessing static member variable c in struct SomeStruct, through var and p.
var.a = var.c;
var.b = p->c;
// Calling a static member function.
SomeStruct::bar();
var.bar();
p->bar();
```

Фон

Оператор -> необходим, потому что оператор доступа к члену . имеет приоритет над оператором разыменования * .

Можно было бы ожидать, что *p_a будет разыменовывать p (в результате ссылается на объект p , указывающий на), а затем обращается к его члену a . Но на самом деле он

пытается получить доступ к члену `a` из `p` а затем разыменовать его. `le *pa` эквивалентно `*(pa)`. В приведенном выше примере это приведет к ошибке компилятора из-за двух фактов: во-первых, `p` является указателем и не имеет члена `a`. Во-вторых, `a` является целым числом и, следовательно, не может быть разыменован.

Необычно используемым решением этой проблемы было бы прямое управление приоритетом: `(*p).a`

Вместо этого оператор- `->` почти всегда используется. Это короткая рука для первого разыменования указателя, а затем доступа к нему. `le (*p).a` точно такое же, как `p->a`.

Оператор `::` оператор области видимости, используемый таким же образом, как и доступ к члену пространства имен. Это связано с тем, что статический член класса рассматривается как область видимости этого класса, но не считается членом экземпляров этого класса. Использование нормального `.` и `->` также допускается для статических членов, несмотря на то, что они не являются членами экземпляра по историческим причинам; это полезно для написания генерического кода в шаблонах, поскольку вызывающему абоненту не нужно беспокоиться о том, является ли данная функция-член статичной или нестатической.

Частное наследование: ограничение интерфейса базового класса

Частное наследование полезно, когда требуется ограничить публичный интерфейс класса:

```
class A {
public:
    int move();
    int turn();
};

class B : private A {
public:
    using A::turn;
};

B b;
b.move(); // compile error
b.turn(); // OK
```

Этот подход эффективно предотвращает доступ к общедоступным методам `A` путем отбрасывания указателя `A` или ссылки:

```
B b;
A& a = static_cast<A&>(b); // compile error
```

В случае публичного наследования такое литье предоставит доступ ко всем общедоступным методам `A`, несмотря на альтернативные способы предотвращения этого в производном `B`, например, скрытие:

```
class B : public A {
private:
    int move();
};
```

или частное использование:

```
class B : public A {
private:
    using A::move;
};
```

то для обоих случаев это возможно:

```
B b;
A& a = static_cast<A&>(b); // OK for public inheritance
a.move(); // OK
```

Заключительные классы и структуры

C ++ 11

Вывод класса может быть запрещен с `final` спецификатором. Объявим окончательный класс:

```
class A final {
};
```

Теперь любая попытка подкласса вызовет ошибку компиляции:

```
// Compilation error: cannot derive from final class:
class B : public A {
};
```

Конечный класс может отображаться в любом месте иерархии классов:

```
class A {
};

// OK.
class B final : public A {
};

// Compilation error: cannot derive from final class B.
class C : public B {
};
```

дружба

Ключевое слово `friend` используется для предоставления другим классам и функциям

доступа к закрытым и защищенным членам класса, даже если они определены вне рамок класса.

```
class Animal{
private:
    double weight;
    double height;
public:
    friend void printWeight(Animal animal);
    friend class AnimalPrinter;
    // A common use for a friend function is to overload the operator<< for streaming.
    friend std::ostream& operator<<(std::ostream& os, Animal animal);
};

void printWeight(Animal animal)
{
    std::cout << animal.weight << "\n";
}

class AnimalPrinter
{
public:
    void print(const Animal& animal)
    {
        // Because of the `friend class AnimalPrinter;" declaration, we are
        // allowed to access private members here.
        std::cout << animal.weight << ", " << animal.height << std::endl;
    }
}

std::ostream& operator<<(std::ostream& os, Animal animal)
{
    os << "Animal height: " << animal.height << "\n";
    return os;
}

int main() {
    Animal animal = {10, 5};
    printWeight(animal);

    AnimalPrinter aPrinter;
    aPrinter.print(animal);

    std::cout << animal;
}
```

```
10
10, 5
Animal height: 5
```

Вложенные классы / структуры

`class` или `struct` также могут содержать другое определение `class` / `struct` внутри себя, которое называется «вложенным классом»; в этой ситуации содержащий класс называется «охватывающим классом». Вложенное определение класса считается членом входящего

класса, но в противном случае оно является отдельным.

```
struct Outer {
    struct Inner { };
};
```

За пределами закрывающего класса доступ к вложенным классам осуществляется с помощью оператора области видимости. Однако изнутри закрывающего класса вложенные классы могут использоваться без квалификаторов:

```
struct Outer {
    struct Inner { };

    Inner in;
};

// ...

Outer o;
Outer::Inner i = o.in;
```

Как и в случае не-вложенного `class / struct`, функции-члены и статические переменные могут быть определены либо внутри вложенного класса, либо в охватывающем пространстве имен. Тем не менее, они не могут быть определены внутри охватывающего класса, поскольку он считается другим классом, чем вложенный класс.

```
// Bad.
struct Outer {
    struct Inner {
        void do_something();
    };

    void Inner::do_something() {}
};

// Good.
struct Outer {
    struct Inner {
        void do_something();
    };
};

void Outer::Inner::do_something() {}
```

Как и в случае не-вложенных классов, вложенные классы могут быть объявлены и определены позже, если они определены до их непосредственного использования.

```
class Outer {
    class Inner1;
    class Inner2;

    class Inner1 {};
```

```

    Inner1 in1;
    Inner2* in2p;

public:
    Outer();
    ~Outer();
};

class Outer::Inner2 {};

Outer::Outer() : in1(Inner1()), in2p(new Inner2) {}
Outer::~~Outer() {
    if (in2p) { delete in2p; }
}

```

C ++ 11

До C ++ 11 вложенные классы имели доступ только к именам типов, `static` членам и счетчикам из охватывающего класса; все остальные члены, определенные в охватывающем классе, были недоступны.

C ++ 11

Начиная с C ++ 11, вложенные классы и их члены обрабатываются так, как если бы они были `friend s` входящего класса и могли обращаться ко всем своим членам в соответствии с обычными правилами доступа; если члены вложенного класса требуют возможности оценить один или несколько нестатических членов охватывающего класса, они должны, таким образом, передать экземпляр:

```

class Outer {
    struct Inner {
        int get_sizeof_x() {
            return sizeof(x); // Legal (C++11): x is unevaluated, so no instance is required.
        }

        int get_x() {
            return x; // Illegal: Can't access non-static member without an instance.
        }

        int get_x(Outer& o) {
            return o.x; // Legal (C++11): As a member of Outer, Inner can access private
members.
        }
    };

    int x;
};

```

И наоборот, класс-оболочка *не* рассматривается как друг вложенного класса и, следовательно, не может получить доступ к его частным членам без явного разрешения.

```

class Outer {
    class Inner {

```



```

        // friend class Outer;

        int x;
    };

    Inner in;

public:
    int get_x() {
        return in.x; // Error: int Outer::Inner::x is private.
        // Uncomment "friend" line above to fix.
    }
};

```

Друзья вложенного класса автоматически не считаются друзьями окружающего класса; если они также должны быть друзьями окружающего класса, это должно быть объявлено отдельно. И наоборот, поскольку закрывающий класс автоматически не считается другом вложенного класса, ни друзья окружающего класса не будут считаться друзьями вложенного класса.

```

class Outer {
    friend void barge_out(Outer& out, Inner& in);

    class Inner {
        friend void barge_in(Outer& out, Inner& in);

        int i;
    };

    int o;
};

void barge_in(Outer& out, Outer::Inner& in) {
    int i = in.i; // Good.
    int o = out.o; // Error: int Outer::o is private.
}

void barge_out(Outer& out, Outer::Inner& in) {
    int i = in.i; // Error: int Outer::Inner::i is private.
    int o = out.o; // Good.
}

```

Как и все другие члены класса, вложенные классы могут быть названы только вне класса, если они имеют открытый доступ. Тем не менее, вам разрешен доступ к ним независимо от модификатора доступа до тех пор, пока вы не укажете их явно.

```

class Outer {
    struct Inner {
        void func() { std::cout << "I have no private taboo.\n"; }
    };

public:
    static Inner make_Inner() { return Inner(); }
};

```

```
// ...

Outer::Inner oi; // Error: Outer::Inner is private.

auto oi = Outer::make_Inner(); // Good.
oi.func();                     // Good.
Outer::make_Inner().func();    // Good.
```

Вы также можете создать псевдоним типа для вложенного класса. Если псевдоним типа содержится в охватывающем классе, вложенный тип и псевдоним типа могут иметь разные модификаторы доступа. Если тип псевдоним находится вне его класса, он требует, чтобы либо вложенный класс, или `typedef` их, быть публичным.

```
class Outer {
    class Inner_ {};

public:
    typedef Inner_ Inner;
};

typedef Outer::Inner ImOut; // Good.
typedef Outer::Inner_ ImBad; // Error.

// ...

Outer::Inner oi; // Good.
Outer::Inner_ oi; // Error.
ImOut oi; // Good.
```

Как и в случае с другими классами, вложенные классы могут быть получены из или из других классов.

```
struct Base {};

struct Outer {
    struct Inner : Base {};
};

struct Derived : Outer::Inner {};
```

Это может быть полезно в ситуациях, когда охватывающий класс получен из другого класса, позволяя программисту обновлять вложенный класс по мере необходимости. Это можно объединить с `typedef`, чтобы обеспечить согласованное имя для каждого вложенного класса вложенного класса:

```
class BaseOuter {
    struct BaseInner_ {
        virtual void do_something() {}
        virtual void do_something_else();
    } b_in;

public:
    typedef BaseInner_ Inner;
```

```

    virtual ~BaseOuter() = default;

    virtual Inner& getInner() { return b_in; }
};

void BaseOuter::BaseInner_::do_something_else() {}

// ---

class DerivedOuter : public BaseOuter {
    // Note the use of the qualified typedef; BaseOuter::BaseInner_ is private.
    struct DerivedInner_ : BaseOuter::Inner {
        void do_something() override {}
        void do_something_else() override;
    } d_in;

public:
    typedef DerivedInner_ Inner;

    BaseOuter::Inner& getInner() override { return d_in; }
};

void DerivedOuter::DerivedInner_::do_something_else() {}

// ...

// Calls BaseOuter::BaseInner_::do_something();
BaseOuter* b = new BaseOuter;
BaseOuter::Inner& bin = b->getInner();
bin.do_something();
b->getInner().do_something();

// Calls DerivedOuter::DerivedInner_::do_something();
BaseOuter* d = new DerivedOuter;
BaseOuter::Inner& din = d->getInner();
din.do_something();
d->getInner().do_something();

```

В приведенном выше случае, как `BaseOuter` и `DerivedOuter` поставляют тип члена `Inner`, как `BaseInner_` и `DerivedInner_`, соответственно. Это позволяет выводить вложенные типы без нарушения интерфейса охватывающего класса и позволяет использовать вложенный тип в полиморфном виде.

Типы участников и псевдонимы

`class` или `struct` также могут определять псевдонимы типа члена, которые являются псевдонимами типа, содержащимися внутри и рассматриваемыми как члены самого класса.

```

struct IHaveATypedef {
    typedef int MyTypedef;
};

struct IHaveATemplateTypedef {
    template<typename T>
    using MyTemplateTypedef = std::vector<T>;
};

```

Как и статические члены, эти typedefs доступны с помощью оператора области видимости :: .

```
IAHaveATypedef::MyTypedef i = 5; // i is an int.

IAHaveATemplateTypedef::MyTemplateTypedef<int> v; // v is a std::vector<int>.
```

Как и в случае с обычными псевдонимами типа, каждому псевдониmu типа члена разрешается ссылаться на любой тип, определенный или сглаженный раньше, но не после его определения. Аналогично, typedef вне определения класса может ссылаться на любые доступные typedef в определении класса, если он приходит после определения класса.

```
template<typename T>
struct Helper {
    T get() const { return static_cast<T>(42); }
};

struct IHaveTypedefs {
    // typedef MyTypedef NonLinearTypedef; // Error if uncommented.
    typedef int MyTypedef;
    typedef Helper<MyTypedef> MyTypedefHelper;
};

IAHaveTypedefs::MyTypedef i; // x_i is an int.
IAHaveTypedefs::MyTypedefHelper hi; // x_hi is a Helper<int>.

typedef IHaveTypedefs::MyTypedef TypedefBeFree;
TypedefBeFree ii; // ii is an int.
```

Алиасы типа пользователя могут быть объявлены с любым уровнем доступа и будут уважать соответствующий модификатор доступа.

```
class TypedefAccessLevels {
    typedef int PrvInt;

protected:
    typedef int ProInt;

public:
    typedef int PubInt;
};

TypedefAccessLevels::PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
TypedefAccessLevels::ProInt pro_i; // Error: TypedefAccessLevels::ProInt is protected.
TypedefAccessLevels::PubInt pub_i; // Good.

class Derived : public TypedefAccessLevels {
    PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
    ProInt pro_i; // Good.
    PubInt pub_i; // Good.
};
```

Это можно использовать для обеспечения уровня абстракции, позволяя дизайнеру класса изменять свои внутренние работы без нарушения кода, который опирается на него.

```

class Something {
    friend class SomeComplexType;

    short s;
    // ...

public:
    typedef SomeComplexType MyHelper;

    MyHelper get_helper() const { return MyHelper(8, s, 19.5, "shoe", false); }

    // ...
};

// ...

Something s;
Something::MyHelper hlp = s.get_helper();

```

В этой ситуации, если класс-помощник изменен с `SomeComplexType` на какой-либо другой тип, нужно будет изменить только объявление `typedef` и `friend`; поскольку класс-помощник предоставляет те же функции, любой код, который использует его как `Something::MyHelper` вместо указания его по имени, как правило, будет работать без каких-либо изменений. Таким образом, мы минимизируем количество кода, которое необходимо изменить при изменении базовой реализации, так что имя типа необходимо изменить только в одном месте.

Это также можно сочетать с `decltype`, если этого так `decltype`.

```

class SomethingElse {
    AnotherComplexType<bool, int, SomeThirdClass> helper;

public:
    typedef decltype(helper) MyHelper;

private:
    InternalVariable<MyHelper> ivh;

    // ...

public:
    MyHelper& get_helper() const { return helper; }

    // ...
};

```

В этой ситуации изменение реализации `SomethingElse::helper` автоматически изменит `typedef` для нас из-за `decltype`. Это минимизирует количество необходимых изменений, когда мы хотим изменить `helper`, который минимизирует риск человеческой ошибки.

Однако, как и во всем, это можно занять слишком далеко. Если имя типа используется только один или два раза внутренне и нулевое время извне, например, нет необходимости предоставлять псевдоним для него. Если он используется сотни или тысячи раз во всем

проекте, или если у него есть достаточно длинное имя, тогда может быть полезно предоставить его как typedef, а не всегда использовать его в абсолютных выражениях. Нужно балансировать вперед совместимость и удобство с количеством создаваемого ненужного шума.

Это также можно использовать с классами шаблонов, чтобы обеспечить доступ к параметрам шаблона вне класса.

```
template<typename T>
class SomeClass {
    // ...

public:
    typedef T MyParam;
    MyParam getParam() { return static_cast<T>(42); }
};

template<typename T>
typename T::MyParam some_func(T& t) {
    return t.getParam();
}

SomeClass<int> si;
int i = some_func(si);
```

Обычно это используется с контейнерами, которые обычно предоставляют свой тип элемента и другие вспомогательные типы в качестве псевдонимов типа члена. Например, большинство контейнеров в стандартной библиотеке C++ предоставляют следующие 12 вспомогательных типов, а также любые другие специальные типы, которые могут потребоваться.

```
template<typename T>
class SomeContainer {
    // ...

public:
    // Let's provide the same helper types as most standard containers.
    typedef T value_type;
    typedef std::allocator<value_type> allocator_type;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef MyIterator<value_type> iterator;
    typedef MyConstIterator<value_type> const_iterator;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
};
```

До C++ 11 он также обычно использовался для создания «шаблона typedef», поскольку эта функция еще не была доступна; они стали немного менее распространены с введением

шаблонов псевдонимов, но по-прежнему полезны в некоторых ситуациях (и сочетаются с шаблонами псевдонимов в других ситуациях, что может быть очень полезно для получения отдельных компонентов сложного типа, таких как указатель на функцию). Они обычно используют `type` имени для своего псевдонима типа.

```
template<typename T>
struct TemplateTypedef {
    typedef T type;
}

TemplateTypedef<int>::type i; // i is an int.
```

Это часто использовалось с типами с несколькими параметрами шаблона, чтобы обеспечить псевдоним, который определяет один или несколько параметров.

```
template<typename T, size_t SZ, size_t D>
class Array { /* ... */ };

template<typename T, size_t SZ>
struct OneDArray {
    typedef Array<T, SZ, 1> type;
};

template<typename T, size_t SZ>
struct TwoDArray {
    typedef Array<T, SZ, 2> type;
};

template<typename T>
struct MonoDisplayLine {
    typedef Array<T, 80, 1> type;
};

OneDArray<int, 3>::type    arr1i; // arr1i is an Array<int, 3, 1>.
TwoDArray<short, 5>::type arr2s; // arr2s is an Array<short, 5, 2>.
MonoDisplayLine<char>::type arr3c; // arr3c is an Array<char, 80, 1>.
```

Статические члены класса

Класс также может иметь `static` элементы, которые могут быть либо переменными, либо функциями. Они считаются находящимися в классе, но не рассматриваются как обычные члены; они имеют статическую продолжительность хранения (они существуют с самого начала программы до конца), не привязаны к конкретному экземпляру класса, и для всего класса существует только одна копия.

```
class Example {
    static int num_instances; // Static data member (static member variable).
    int i; // Non-static member variable.

public:
    static std::string static_str; // Static data member (static member variable).
    static int static_func(); // Static member function.
```

```

    // Non-static member functions can modify static member variables.
    Example() { ++num_instances; }
    void set_str(const std::string& str);
};

int         Example::num_instances;
std::string Example::static_str = "Hello.";

// ...

Example one, two, three;
// Each Example has its own "i", such that:
// (&one.i != &two.i)
// (&one.i != &three.i)
// (&two.i != &three.i).
// All three Examples share "num_instances", such that:
// (&one.num_instances == &two.num_instances)
// (&one.num_instances == &three.num_instances)
// (&two.num_instances == &three.num_instances)

```

Статические переменные-члены не считаются определенными внутри класса, только объявлены и, следовательно, имеют свое определение вне определения класса; программисту разрешено, но не требуется, инициализировать статические переменные в их определении. При определении переменных-членов ключевое слово `static` опущено.

```

class Example {
    static int num_instances;           // Declaration.

public:
    static std::string static_str;     // Declaration.

    // ...
};

int         Example::num_instances;    // Definition.  Zero-initialised.
std::string Example::static_str = "Hello."; // Definition.

```

В связи с этим статические переменные могут быть неполными (кроме `void`), если они позже определены как полный тип.

```

struct ForwardDeclared;

class ExIncomplete {
    static ForwardDeclared fd;
    static ExIncomplete  i_contain_myself;
    static int           an_array[];
};

struct ForwardDeclared {};

ForwardDeclared ExIncomplete::fd;
ExIncomplete    ExIncomplete::i_contain_myself;
int             ExIncomplete::an_array[5];

```

Функции статического члена могут быть определены внутри или вне определения класса, как и для обычных функций-членов. Как и для статических переменных-членов, ключевое

слово `static` опущено при определении статических функций-членов вне определения класса.

```
// For Example above, either...
class Example {
    // ...

public:
    static int static_func() { return num_instances; }

    // ...

    void set_str(const std::string& str) { static_str = str; }
};

// Or...

class Example { /* ... */ };

int Example::static_func() { return num_instances; }
void Example::set_str(const std::string& str) { static_str = str; }
```

Если статическая переменная-член объявляется `const` но не является `volatile` и имеет тип интеграла или перечисления, ее можно инициализировать при объявлении внутри определения класса.

```
enum E { VAL = 5 };

struct ExConst {
    const static int ci = 5;           // Good.
    static const E ce = VAL;          // Good.
    const static double cd = 5;       // Error.
    static const volatile int cvi = 5; // Error.

    const static double good_cd;
    static const volatile int good_cvi;
};

const double ExConst::good_cd = 5;    // Good.
const volatile int ExConst::good_cvi = 5; // Good.
```

C ++ 11

Начиная с C ++ 11, статические переменные- `LiteralType` типов `LiteralType` (типы, которые могут быть `constexpr` во время компиляции, в соответствии с правилами `constexpr`) также могут быть объявлены как `constexpr` ; если это так, они должны быть инициализированы в определении класса.

```
struct ExConstexpr {
    constexpr static int ci = 5;           // Good.
    static constexpr double cd = 5;       // Good.
    constexpr static int carr[] = { 1, 1, 2 }; // Good.
    static constexpr ConstructibleClass c{}; // Good.
    constexpr static int bad_ci;          // Error.
```

```
};

constexpr int ExConstexpr::bad_ci = 5; // Still an error.
```

Если `const` или `constexpr` статическая переменная член *УСО-используется* (неофициально, если он имеет свой адрес Предпринятые или назначается в качестве ссылки), то он должен еще отдельное определение, вне определения класса. Это определение не позволяет содержать инициализатор.

```
struct ExODR {
    static const int odr_used = 5;
};

// const int ExODR::odr_used;

const int* odr_user = & ExODR::odr_used; // Error; uncomment above line to resolve.
```

Поскольку статические члены не привязаны к данному экземпляру, к ним можно получить доступ с помощью оператора области видимости `::`.

```
std::string str = Example::static_str;
```

Их также можно получить, как если бы они были обычными, нестационарными членами. Это имеет историческое значение, но используется реже, чем оператор области, чтобы предотвратить путаницу в отношении того, является ли элемент статическим или нестационарным.

```
Example ex;
std::string rts = ex.static_str;
```

Члены класса могут получить доступ к статическим членам без квалификации своей области, как и для нестатических членов класса.

```
class ExTwo {
    static int num_instances;
    int my_num;

public:
    ExTwo() : my_num(num_instances++) {}

    static int get_total_instances() { return num_instances; }
    int get_instance_number() const { return my_num; }
};

int ExTwo::num_instances;
```

Они не могут быть `mutable` и не должны быть такими; поскольку они не привязаны к какому-либо конкретному экземпляру, независимо от того, является ли экземпляр или не является константой, не затрагивает статические члены.

```

struct ExDontNeedMutable {
    int immuta;
    mutable int muta;

    static int i;

    ExDontNeedMutable() : immuta(-5), muta(-5) {}
};
int ExDontNeedMutable::i;

// ...

const ExDontNeedMutable dnm;
dnm.immuta = 5; // Error: Can't modify read-only object.
dnm.muta = 5;   // Good.  Mutable fields of const objects can be written.
dnm.i = 5;      // Good.  Static members can be written regardless of an instance's const-
ness.

```

Статические члены рассматривают модификаторы доступа, как и нестатические элементы.

```

class ExAccess {
    static int prv_int;

protected:
    static int pro_int;

public:
    static int pub_int;
};

int ExAccess::prv_int;
int ExAccess::pro_int;
int ExAccess::pub_int;

// ...

int x1 = ExAccess::prv_int; // Error: int ExAccess::prv_int is private.
int x2 = ExAccess::pro_int; // Error: int ExAccess::pro_int is protected.
int x3 = ExAccess::pub_int; // Good.

```

Поскольку они не привязаны к данному экземпляру, статические функции-члены не имеют `this` указателя; из-за этого они не могут получить доступ к нестационарным переменным-членам, если не передал экземпляр.

```

class ExInstanceRequired {
    int i;

public:
    ExInstanceRequired() : i(0) {}

    static void bad_mutate() { ++i *= 5; } // Error.
    static void good_mutate(ExInstanceRequired& e) { ++e.i *= 5; } // Good.
};

```

Из-за отсутствия `this` указателя их адреса не могут храниться в указателях-членах-

функциях и вместо этого сохраняются в обычных указателях-функциях.

```
struct ExPointer {
    void nsfunc() {}
    static void sfunc() {}
};

typedef void (ExPointer::* mem_f_ptr)();
typedef void (*f_ptr)();

mem_f_ptr p_sf = &ExPointer::sfunc; // Error.
f_ptr p_sf = &ExPointer::sfunc; // Good.
```

Из-за не имея `this` указатель, они также не могут быть `const` или `volatile`, они не могут иметь реф-классификаторы. Они также не могут быть виртуальными.

```
struct ExCVQualifiersAndVirtual {
    static void func() {} // Good.
    static void cfunc() const {} // Error.
    static void vfunc() volatile {} // Error.
    static void cvfunc() const volatile {} // Error.
    static void rfunc() & {} // Error.
    static void rvfunc() && {} // Error.

    virtual static void vsfunc() {} // Error.
    static virtual void svfunc() {} // Error.
};
```

Поскольку они не привязаны к данному экземпляру, статические переменные-члены эффективно рассматриваются как специальные глобальные переменные; они создаются при запуске программы и уничтожаются при ее выходе, независимо от того, существуют ли какие-либо экземпляры класса. Существует только одна копия каждой статической переменной-члена (если только переменная не объявлена `thread_local` (C++ 11 или новее), в этом случае есть один экземпляр для потока).

Статические переменные-члены имеют ту же связь, что и класс, независимо от того, имеет ли класс внешнюю или внутреннюю связь. Локальным классам и неназванным классам не разрешено иметь статические члены.

Нестатические функции-члены

Класс может иметь **нестатические функции-члены**, которые работают с отдельными экземплярами класса.

```
class CL {
public:
    void member_function() {}
};
```

Эти функции вызываются в экземпляре класса, например:

```
CL instance;
instance.member_function();
```

Они могут быть определены как внутри, так и вне определения класса; если они определены снаружи, они указываются как входящие в область класса.

```
struct ST {
    void defined_inside() {}
    void defined_outside();
};
void ST::defined_outside() {}
```

Они могут быть **квалифицированными** и / или **квалифицированными квалифицированными специалистами**, влияющими на то, как они видят экземпляр, к которому они обращаются; функция увидит экземпляр как имеющий указанный cv-квалификатор (ы), если он есть. Какая версия называется, будет основана на cv-квалификаторах экземпляра. Если в качестве экземпляра нет версии с теми же cv-квалификаторами, то будет выпущена версия с более высокой квалификацией, если она доступна.

```
struct CVQualifiers {
    void func() {} // 1: Instance is non-cv-qualified.
    void func() const {} // 2: Instance is const.

    void cv_only() const volatile {}
};

CVQualifiers non_cv_instance;
const CVQualifiers c_instance;

non_cv_instance.func(); // Calls #1.
c_instance.func(); // Calls #2.

non_cv_instance.cv_only(); // Calls const volatile version.
c_instance.cv_only(); // Calls const volatile version.
```

C ++ 11

Функция ref-qualifiers функции-члена указывает, должна ли функция быть вызвана для экземпляров rvalue, и использовать тот же синтаксис, что и функция cv-qualifiers.

```
struct RefQualifiers {
    void func() & {} // 1: Called on normal instances.
    void func() && {} // 2: Called on rvalue (temporary) instances.
};

RefQualifiers rf;
rf.func(); // Calls #1.
RefQualifiers{}.func(); // Calls #2.
```

CV-квалификаторы и ref-квалификаторы также могут быть объединены, если необходимо.

```
struct BothCVAndRef {
```

```
void func() const& {} // Called on normal instances.  Sees instance as const.
void func() &&      {} // Called on temporary instances.

};
```

Они также могут быть **виртуальными** ; это имеет основополагающее значение для полиморфизма и позволяет дочернему классу (классам) предоставлять тот же интерфейс, что и родительский класс, и при этом предоставлять свои собственные функции.

```
struct Base {
    virtual void func() {}
};
struct Derived {
    virtual void func() {}
};

Base* bp = new Base;
Base* dp = new Derived;
bp->func(); // Calls Base::func().
dp->func(); // Calls Derived::func().
```

Для получения дополнительной информации см. [Здесь](#) .

Без названия struct / class

Доступна struct (тип не имеет имени)

```
void foo()
{
    struct /* No name */ {
        float x;
        float y;
    } point;

    point.x = 42;
}
```

или же

```
struct Circle
{
    struct /* No name */ {
        float x;
        float y;
    } center; // but a member name
    float radius;
};
```

и позже

```
Circle circle;
circle.center.x = 42.f;
```

но НЕ *анонимная struct* (неназванный тип и неназванный объект)

```
struct InvalidCircle
{
    struct /* No name */ {
        float centerX;
        float centerY;
    }; // No member either.
    float radius;
};
```

Примечание. Некоторые компиляторы разрешают *анонимную struct* как расширение .

C ++ 11

- *lambda* можно рассматривать как специальную *неназванную struct* .
- `decltype` позволяет получить тип *неназванной struct* :

```
decltype(circle.point) otherPoint;
```

- *unnamed struct* instance может быть параметром метода шаблона:

```
void print_square_coordinates()
{
    const struct {float x; float y;} points[] = {
        {-1, -1}, {-1, 1}, {1, -1}, {1, 1}
    };

    // for range relies on `template <class T, std::size_t N> std::begin(T (&)[N])`
    for (const auto& point : points) {
        std::cout << "{" << point.x << ", " << point.y << "}\n";
    }

    decltype(points[0]) topRightCorner{1, 1};
    auto it = std::find(points, points + 4, topRightCorner);
    std::cout << "top right corner is the "
        << 1 + std::distance(points, it) << "th\n";
}
```

Прочитайте Классы / Структуры онлайн: <https://riptutorial.com/ru/cplusplus/topic/508/классы---структуры>

глава 42: ключевое слово const

Синтаксис

- `const Тип myVariable = initial; //` Объявляет константную переменную; не может быть изменено
- `const Тип & myReference = myVariable; //` Объявляет ссылку на константную переменную
- `const Тип * myPointer = & myVariable; //` Объявляет указатель на константу. Указатель может меняться, но основной элемент данных не может быть изменен с помощью указателя
- `Тип * const myPointer = & myVariable; //` Объявляет указатель `const`. Указатель не может быть переназначен, чтобы указать на что-то еще, но базовый элемент данных можно изменить
- `const Тип * const myPointer = & myVariable; //` Объявляет `const`-указатель-`const`-`const`.

замечания

Переменная помечается как `const` не может быть изменен ¹. Попытка вызвать любые неконстантные операции над ним приведет к ошибке компилятора.

1: Ну, его можно изменить через `const_cast`, но вы почти никогда не должны использовать это

Examples

Константные локальные переменные

Декларация и использование.

```
// a is const int, so it can't be changed
const int a = 15;
a = 12;           // Error: can't assign new value to const variable
a += 1;           // Error: can't assign new value to const variable
```

Связывание ссылок и указателей

```
int &b = a;        // Error: can't bind non-const reference to const variable
const int &c = a;  // OK; c is a const reference

int *d = &a;       // Error: can't bind pointer-to-non-const to const variable
const int *e = &a  // OK; e is a pointer-to-const

int f = 0;
e = &f;            // OK; e is a non-const pointer-to-const,
                  // which means that it can be rebound to new int* or const int*
```



```
*e = 1           // Error: e is a pointer-to-const which means that
                  // the value it points to can't be changed through dereferencing e

int *g = &f;
*g = 1;          // OK; this value still can be changed through dereferencing
                  // a pointer-not-to-const
```

Указатели констант

```
int a = 0, b = 2;

const int* pA = &a; // pointer-to-const. `a` can't be changed through this
int* const pB = &a; // const pointer. `a` can be changed, but this pointer can't.
const int* const pC = &a; // const pointer-to-const.

//Error: Cannot assign to a const reference
*pA = b;

pA = &b;

*pB = b;

//Error: Cannot assign to const pointer
pB = &b;

//Error: Cannot assign to a const reference
*pC = b;

//Error: Cannot assign to const pointer
pC = &b;
```

Функции члена-члена

Функции-члены класса могут быть объявлены `const`, который сообщает компилятору и будущим читателям, что эта функция не будет изменять объект:

```
class MyClass
{
private:
    int myInt_;
public:
    int myInt() const { return myInt_; }
    void setMyInt(int myInt) { myInt_ = myInt; }
};
```

В функции-члене `const this` указатель эффективно представляет собой `const MyClass *` вместо `MyClass *`. Это означает, что вы не можете изменять какие-либо переменные-члены внутри функции; компилятор выдаст предупреждение. Поэтому `setMyInt` не может быть объявлен `const`.

Вы должны почти всегда отмечать функции-члены как `const` когда это возможно. На `const MyClass` можно вызвать только функции `const` члена.

`static` методы не могут быть объявлены как `const`. Это связано с тем, что статический метод принадлежит классу и не вызывается на объект; поэтому он никогда не может изменять внутренние переменные объекта. Поэтому объявление `static` методов как `const` будет излишним.

Избегание дублирования кода в методах `const` и `non-const` getter.

В C++ методы, которые отличаются только `const` могут быть перегружены. Иногда может потребоваться две версии getter, которые возвращают ссылку на некоторую часть.

Пусть `Foo` - класс, который имеет два метода, которые выполняют идентичные операции и возвращает ссылку на объект типа `Bar`:

```
class Foo
{
public:
    Bar& GetBar(/* some arguments */)
    {
        /* some calculations */
        return bar;
    }

    const Bar& GetBar(/* some arguments */) const
    {
        /* some calculations */
        return bar;
    }

    // ...
};
```

Единственная разница между ними заключается в том, что один метод является неконстантным и возвращает неконстантную ссылку (которую можно использовать для изменения объекта), а вторая является константой и возвращает константную ссылку.

Чтобы избежать дублирования кода, возникает соблазн вызвать один метод из другого. Однако мы не можем вызывать метод `non-const` из константы. Но мы можем вызвать метод `const` из неконстантного. Это потребует использования «`const_cast`» для удаления спецификатора `const`.

Решение:

```
struct Foo
{
    Bar& GetBar(/*arguments*/)
    {
        return const_cast<Bar&>(const_cast<const Foo*>(this)->GetBar(/*arguments*/));
    }

    const Bar& GetBar(/*arguments*/) const
    {
        /* some calculations */
    }
};
```

```

        return foo;
    }
};

```

В приведенном выше коде мы вызываем const-версию `GetBar` из не-const `GetBar`, отбрасывая это на тип `const`: `const_cast<const Foo*>(this)`. Поскольку мы вызываем метод `const` из не-const, сам объект не является константой, и исключение `const` допускается.

Изучите следующий более полный пример:

```

#include <iostream>

class Student
{
public:
    char& GetScore(bool midterm)
    {
        return const_cast<char&>(const_cast<const Student*>(this)->GetScore(midterm));
    }

    const char& GetScore(bool midterm) const
    {
        if (midterm)
        {
            return midtermScore;
        }
        else
        {
            return finalScore;
        }
    }

private:
    char midtermScore;
    char finalScore;
};

int main()
{
    // non-const object
    Student a;
    // We can assign to the reference. Non-const version of GetScore is called
    a.GetScore(true) = 'B';
    a.GetScore(false) = 'A';

    // const object
    const Student b(a);
    // We still can call GetScore method of const object,
    // because we have overloaded const version of GetScore
    std::cout << b.GetScore(true) << b.GetScore(false) << '\n';
}

```

Прочитайте ключевое слово `const` онлайн: <https://riptutorial.com/ru/cplusplus/topic/2386/ключевое-слово-const>

глава 43: Ключевое слово Friend

Вступление

Хорошо продуманные классы инкапсулируют их функциональность, скрывая их реализацию, обеспечивая при этом чистый документированный интерфейс. Это позволяет перепроектировать или изменить, если интерфейс не изменился.

В более сложном сценарии может потребоваться несколько классов, которые полагаются на детали реализации друг друга. Классы и функции друзей позволяют этим одноранговым узлам получить доступ к деталям друг друга без ущерба для инкапсуляции и скрывания информации документированного интерфейса.

Examples

Функция друга

Класс или структура могут объявлять любую функцию, которая является ее другом. Если функция является другом класса, она может получить доступ ко всем ее защищенным и закрытым членам:

```
// Forward declaration of functions.
void friend_function();
void non_friend_function();

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
private:
    int private_value;
    // Declare one of the function as a friend.
    friend void friend_function();
};

void non_friend_function() {
    PrivateHolder ph(10);
    // Compilation error: private_value is private.
    std::cout << ph.private_value << std::endl;
}

void friend_function() {
    // OK: friends may access private values.
    PrivateHolder ph(10);
    std::cout << ph.private_value << std::endl;
}
```

Модификаторы доступа не изменяют семантику друзей. Публичные, защищенные и частные объявления друга эквивалентны.

Объявления друзей не наследуются. Например, если мы подклассифицируем `PrivateHolder` :

```
class PrivateHolderDerived : public PrivateHolder {
public:
    PrivateHolderDerived(int val) : PrivateHolder(val) {}
private:
    int derived_private_value = 0;
};
```

и попытайтесь получить доступ к его членам, мы получим следующее:

```
void friend_function() {
    PrivateHolderDerived pd(20);
    // OK.
    std::cout << pd.private_value << std::endl;
    // Compilation error: derived_private_value is private.
    std::cout << pd.derived_private_value << std::endl;
}
```

Обратите внимание, что `PrivateHolderDerived` члена `PrivateHolderDerived` не может получить доступ к `PrivateHolder::private_value` , в то время как функция друга может это сделать.

Метод Friend

Методы могут объявляться как друзья, так и функции:

```
class Accesser {
public:
    void private_accesser();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend void Accesser::private_accesser();
private:
    int private_value;
};

void Accesser::private_accesser() {
    PrivateHolder ph(10);
    // OK: this method is declares as friend.
    std::cout << ph.private_value << std::endl;
}
```

Класс друга

Целый класс может быть объявлен другом. Объявление класса друга означает, что любой член друга может получить доступ к закрытым и защищенным членам объявляющего класса:

```
class Accesser {
```

```

public:
    void private_accesser1();
    void private_accesser2();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend class Accesser;
private:
    int private_value;
};

void Accesser::private_accesser1() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value << std::endl;
}

void Accesser::private_accesser2() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value + 1 << std::endl;
}

```

Объявление класса друга не является рефлексивным. Если классы нуждаются в частном доступе в обоих направлениях, им нужны объявления друзей.

```

class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
private:
    int private_value = 0;
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    // Accesser is a friend of PrivateHolder
    friend class Accesser;
    void reverse_accesse() {
        // but PrivateHolder cannot access Accesser's members.
        Accesser a;
        std::cout << a.private_value;
    }
private:
    int private_value;
};

```

Прочитайте Ключевое слово Friend онлайн: <https://riptutorial.com/ru/cplusplus/topic/3275/ключевое-слово-friend>

глава 44: ключевое слово mutable

Examples

нестатический модификатор члена класса

`mutable` модификатор в этом контексте используется для указания того, что поле данных объекта `const` может быть изменено без влияния на внешнее видимое состояние объекта.

Если вы думаете о кэшировании результата дорогостоящих вычислений, вероятно, вы должны использовать это ключевое слово.

Если у вас есть поле данных блокировки (например, `std::unique_lock`), которое заблокировано и разблокировано внутри метода `const`, это ключевое слово также вы можете использовать.

Вы не должны использовать это ключевое слово для нарушения логической константы объекта.

Пример с кэшированием:

```
class pi_calculator {
public:
    double get_pi() const {
        if (pi_calculated) {
            return pi;
        } else {
            double new_pi = 0;
            for (int i = 0; i < 1000000000; ++i) {
                // some calculation to refine new_pi
            }
            // note: if pi and pi_calculated were not mutable, we would get an error from a
            compiler
            // because in a const method we can not change a non-mutable field
            pi = new_pi;
            pi_calculated = true;
            return pi;
        }
    }
private:
    mutable bool pi_calculated = false;
    mutable double pi = 0;
};
```

изменчивый лямбда

По умолчанию неявный `operator()` лямбда является `const`. Это запрещает выполнение `const` операций на лямбда. Чтобы разрешить модификацию членов, лямбда может быть помечена как `mutable`, что делает неявный `operator()` не `const`:

```
int a = 0;

auto bad_counter = [a] {
    return a++;    // error: operator() is const
                  // cannot modify members
};

auto good_counter = [a]() mutable {
    return a++;    // OK
}

good_counter(); // 0
good_counter(); // 1
good_counter(); // 2
```

Прочитайте ключевое слово `mutable` онлайн: <https://riptutorial.com/ru/cplusplus/topic/2705/ключевое-слово-mutable>

глава 45: Ключевые слова

Вступление

Ключевые слова имеют фиксированное значение, определенное стандартом C ++ и не могут использоваться как идентификаторы. Неправильно переопределять ключевые слова с использованием препроцессора в любой единицы перевода, которая включает стандартный заголовок библиотеки. Однако ключевые слова теряют свое особое значение внутри атрибутов.

Синтаксис

- `asm ( string-literal );`
- `поехсепт ( выражение ) // значение 1`
- `поехсепт ( константное выражение ) // значение 2`
- `поехсепт // значение 2`
- `размерное унитарное выражение`
- `sizeof ( type-id )`
- `sizeof ... ( идентификатор ) // с C ++ 11`
- `Идентификатор имени-имени-имени-имени типа // Значение 1`
- `typename шаблон вложенного имени-спецификатора ( opt ) simple-template-id // значение 1`
- `Идентификатор typename ( opt ) // Значение 2`
- `typename ... identifier ( opt ) // значение 2; поскольку C ++ 11`
- `typename identifier ( opt ) = type-id // значение 2`
- `template < template-parameter-list > typename ... ( opt ) identifier ( opt ) // значение 3`
- `template < template-parameter-list > идентификатор typename ( opt ) = id-expression // значение 3`

замечания

Полный список ключевых слов выглядит следующим образом:

- `alignas` (поскольку C ++ 11)
- `alignof` (поскольку C ++ 11)
- `asm`
- `auto` : `c C ++ 11` , `до C ++ 11`
- `bool`
- `break`
- `case`
- `catch`
- `char`

- `char16_t` (начиная с C ++ 11)
- `char32_t` (начиная с C ++ 11)
- `class`
- `const`
- `constexpr` (поскольку C ++ 11)
- `const_cast`
- `continue`
- `decltype` (начиная с C ++ 11)
- `default`
- `delete` для управления памятью , для функций (начиная с C ++ 11)
- `do`
- `double`
- `dynamic_cast`
- `else`
- `enum`
- `explicit`
- `export`
- `extern` как спецификатор объявления , в спецификации привязки , для шаблонов
- `false`
- `float`
- `for`
- `friend`
- `goto`
- `if`
- `inline` для функций , для пространств имен (начиная с C ++ 11), для переменных (начиная с C ++ 17)
- `int`
- `long`
- `mutable`
- `namespace`
- `new`
- `noexcept` (поскольку C ++ 11)
- `nullptr` (поскольку C ++ 11)
- `operator`
- `private`
- `protected`
- `public`
- `register`
- `reinterpret_cast`
- `return`
- `short`
- `signed`
- `sizeof`
- `static`
- `static_assert` (с C ++ 11)
- `static_cast`
- `struct`
- `switch`
- `template`
- `this`
- `thread_local` (с C ++ 11)
- `throw`

- `true`
- `try`
- `typedef`
- `typeid`
- `typename`
- `union`
- `unsigned`
- `using` для переопределения имени , для псевдонима пространства имен , для псевдонима типа
- `virtual` для функций , для базовых классов
- `void`
- `volatile`
- `wchar_t`
- `while`

`final` и `override` токенов не являются ключевыми словами. Они могут использоваться как идентификаторы и имеют особое значение только в определенных контекстах.

Маркеры `and` , `and_eq` , `bitand` , `bitor` , `compl` , `not` , `not_eq` , `or` , `or_eq` , `xor` и `xor_eq` альтернативные варианты написания `&&` , `&=` , `&` , `|` , `~!` , `!=` , `||` , `|=` , `^` и `^=` , соответственно. Стандарт не рассматривает их как ключевые слова, но они являются ключевыми словами для всех целей и целей, поскольку их невозможно переопределить или использовать для обозначения чего-либо другого, кроме операторов, которые они представляют.

Следующие темы содержат подробные объяснения многих ключевых слов на C ++, которые служат фундаментальным целям, таким как именование основных типов или контроль потока выполнения.

- [Основные ключевые слова](#)
- [Управление потоком](#)
- [итерация](#)
- [Буквенные ключевые слова](#)
- [Тип Ключевые слова](#)
- [Ключевые слова с переменной переменной](#)
- [Классы / Структуры](#)
- [Спецификаторы класса хранения](#)

Examples

как м

Ключевое слово `asm` использует один операнд, который должен быть строковым литералом.

Он имеет значение, определенное реализацией, но обычно передается ассемблеру реализации, а выход ассемблера включен в блок перевода.

Оператор `asm` является *определением*, а не *выражением*, поэтому он может отображаться как в области блока, так и в пространстве имен (включая глобальную область). Однако, поскольку встроенная сборка не может быть ограничена правилами языка C++, `asm` не может появляться внутри функции `constexpr`.

Пример:

```
[[noreturn]] void halt_system() {
    asm("hlt");
}
```

ЯВНЫЙ

1. При применении к конструктору с одним аргументом предотвращает использование конструктора для выполнения неявных преобразований.

```
class MyVector {
public:
    explicit MyVector(uint64_t size);
};
MyVector v1(100); // ok
uint64_t len1 = 100;
MyVector v2{len1}; // ok, len1 is uint64_t
int len2 = 100;
MyVector v3{len2}; // ill-formed, implicit conversion from int to uint64_t
```

Поскольку C++ 11 представил списки инициализаторов, в C++ 11 и более поздних версиях `explicit` может быть применено к конструктору с любым количеством аргументов с тем же значением, что и в случае с одним аргументом.

```
struct S {
    explicit S(int x, int y);
};
S f() {
    return {12, 34}; // ill-formed
    return S{12, 34}; // ok
}
```

C++ 11

2. При применении к функции преобразования предотвращает использование функции преобразования для выполнения неявных преобразований.

```
class C {
    const int x;
public:
    C(int x) : x(x) {}
}
```

```

    explicit operator int() { return x; }
};
C c(42);
int x = c; // ill-formed
int y = static_cast<int>(c); // ok; explicit conversion

```

noexcept

C++ 11

1. Унарный оператор, который определяет, может ли оценка его операнда распространять исключение. Заметим, что тела вызываемых функций не рассматриваются, поэтому `noexcept` может давать ложные отрицания. Операнд не оценивается.

```

#include <iostream>
#include <stdexcept>
void foo() { throw std::runtime_error("oops"); }
void bar() {}
struct S {};
int main() {
    std::cout << noexcept(foo()) << '\n'; // prints 0
    std::cout << noexcept(bar()) << '\n'; // prints 0
    std::cout << noexcept(1 + 1) << '\n'; // prints 1
    std::cout << noexcept(S()) << '\n'; // prints 1
}

```

В этом примере, хотя `bar()` никогда не может генерировать исключение, `noexcept(bar())` все равно `false`, поскольку тот факт, что `bar()` не может распространять исключение, явно не указан.

2. При объявлении функции указывает, может ли функция распространять исключение. Один, он заявляет, что функция не может распространять исключение. С аргументом в скобках указано, что функция может или не может распространять исключение в зависимости от значения истины аргумента.

```

void f1() { throw std::runtime_error("oops"); }
void f2() noexcept(false) { throw std::runtime_error("oops"); }
void f3() {}
void f4() noexcept {}
void f5() noexcept(true) {}
void f6() noexcept {
    try {
        f1();
    } catch (const std::runtime_error&) {}
}

```

В этом примере мы объявили, что `f4`, `f5` и `f6` не могут распространять исключения. (Хотя исключение может быть выбрано во время выполнения `f6`, оно поймано и не допускается к распространению из функции.) Мы объявили, что `f2` может распространять исключение. Когда спецификатор `noexcept` опущен, он эквивалентен

`noexcept(false)` , поэтому мы неявно объявили, что `f1` и `f3` могут распространять исключения, даже если исключения не могут быть фактически вызваны во время выполнения `f3` .

C++ 17

Независимо от того, является или нет функция `noexcept` является частью типа функции: то есть в приведенном выше примере `f1` , `f2` и `f3` имеют разные типы из `f4` , `f5` и `f6` . Поэтому `noexcept` также значим в указателях функций, аргументах шаблона и т. Д.

```
void g1() {}
void g2() noexcept {}
void (*p1)() noexcept = &g1; // ill-formed, since g1 is not noexcept
void (*p2)() noexcept = &g2; // ok; types match
void (*p3)() = &g1;           // ok; types match
void (*p4)() = &g2;           // ok; implicit conversion
```

имяТипа

1. Когда за ним следует квалифицированное имя, `typename` указывает, что это имя типа. Это часто требуется в шаблонах, в частности, когда вложенный спецификатор имен является зависимым типом, отличным от текущего экземпляра. В этом примере `std::decay<T>` зависит от параметра шаблона `T` , поэтому для того, чтобы назвать тип вложенного `type` , нам нужен префикс всего квалифицированного имени с помощью имени `typename` . Дополнительные сведения см. В разделе « [Где и почему я должен поставить ключевые слова «template» и «typename»?](#) »

```
template <class T>
auto decay_copy(T&& r) -> typename std::decay<T>::type;
```

2. Представляет параметр типа в объявлении [шаблона](#) . В этом контексте он является взаимозаменяемым с `class` .

```
template <typename T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

C++ 17

3. `typename` также может использоваться при объявлении [параметра шаблона шаблона](#) , предшествующего имени параметра, точно так же, как `class` .

```
template <template <class T> typename U>
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

размер

Унарный оператор, который дает размер в байтах своего операнда, который может быть либо выражением, либо типом. Если операнд является выражением, он не оценивается. Размер представляет собой постоянное выражение типа `std::size_t`.

Если операнд является типом, он должен быть заключен в скобки.

- `sizeof` применять `sizeof` к типу функции.
- `sizeof` применять `sizeof` к неполному типу, включая `void`.
- Если `sizeof` применяется к ссылочному типу `T&` или `T&&`, он эквивалентен `sizeof(T)`.
- Когда `sizeof` применяется к типу класса, он дает количество байтов в полном объекте этого типа, включая любые прописные байты в середине или в конце. Поэтому выражение `sizeof` никогда не может иметь значение 0. Подробнее см. В [описании типов объектов](#).
- `char`, `signed char` и `unsigned char` имеют размер 1. Наоборот, байт определяется как объем памяти, необходимый для хранения объекта `char`. Это не обязательно означает 8 бит, так как некоторые системы имеют объекты `char` более 8 бит.

Если *выражение* `expr` является выражением, `sizeof( expr )` эквивалентно `sizeof(T)` где `T` - тип *выражения* `expr`.

```
int a[100];
std::cout << "The number of bytes in `a` is: " << sizeof a;
memset(a, 0, sizeof a); // zeroes out the array
```

C ++ 11

Оператор `sizeof...` дает количество элементов в пакете параметров.

```
template <class... T>
void f(T&&...) {
    std::cout << "f was called with " << sizeof...(T) << " arguments\n";
}
```

Различные ключевые слова

void C ++

1. Когда используется как возвращаемый тип функции, ключевое слово `void` указывает, что функция не возвращает значение. При использовании для списка параметров функции `void` указывает, что функция не принимает никаких параметров. При использовании в объявлении указателя `void` указывает, что указатель является «универсальным».
2. Если тип указателя равен `void *`, указатель может указывать на любую переменную, которая не объявляется с ключевым словом `const` или `volatile`. Указатель `void` не

может быть разыменован, если он не будет передан другому типу. Указатель void может быть преобразован в любой другой тип указателя данных.

3. Указатель void может указывать на функцию, но не на член класса в C ++.

```
void vobject;    // C2182
void *pv;       // okay
int *pint; int i;
int main() {
    pv = &i;
    // Cast optional in C required in C++
    pint = (int *)pv;
```

Неустойчивый C ++

1. Определитель типа, который можно использовать для объявления того, что объект может быть изменен в программе аппаратным обеспечением.

```
volatile declarator ;
```

виртуальный C ++

1. Ключевое слово virtual объявляет виртуальную функцию или виртуальный базовый класс.

```
virtual [type-specifiers] member-function-declarator
virtual [access-specifier] base-class-name
```

параметры

1. **Тип-спецификаторы** Указывает тип возвращаемого значения виртуальной функции-члена.
2. **member-function-declarator** Объявляет функцию-член.
3. **access-specifier** Определяет уровень доступа к базовому классу, public, protected или private. Может отображаться до или после ключевого слова virtual.
4. **base-class-name** Идентифицирует ранее объявленный тип класса

этот указатель

1. Этот указатель является указателем, доступным только внутри нестатических функций-членов класса, структуры или типа объединения. Он указывает на объект, для которого вызывается функция-член. У статических функций-членов нет указателя.

```
this->member-identifier
```


Указатель объекта не является частью самого объекта; он не отражается в результате инструкции `sizeof` для объекта. Вместо этого, когда для объекта вызывается нестатическая функция-член, адрес объекта передается компилятором в качестве скрытого аргумента функции. Например, вызов следующей функции:

```
myDate.setMonth( 3 );
```

can be interpreted this way:

```
setMonth( &myDate, 3 );
```

The object's address is available from within the member function as the `this` pointer. Most uses of this are implicit. It is legal, though unnecessary, to explicitly use `this` when referring to members of the class. For example:

```
void Date::setMonth( int mn )
{
    month = mn;           // These three statements
    this->month = mn;      // are equivalent
    (*this).month = mn;
}
```

The expression `*this` is commonly used to return the current object from a member function:

```
return *this;
```

The `this` pointer is also used to guard against self-reference:

```
if (&Object != this) {
    // do not execute in cases of self-reference
}
```

try, throw и catch Statement (C ++)

1. Чтобы реализовать обработку исключений в C ++, вы используете выражения `try`, `throw` и `catch`.
2. Во-первых, используйте блок `try`, чтобы заключить одно или несколько операторов, которые могут вызывать исключение.
3. Выражение выражает сигнал о том, что исключительное условие - часто, ошибка произошла в блоке `try`. Вы можете использовать объект любого типа в качестве операнда выражения `throw`. Как правило, этот объект используется для передачи информации об ошибке. В большинстве случаев мы рекомендуем использовать класс `std :: exception` или один из производных классов, определенных в стандартной библиотеке. Если один из них не подходит, мы рекомендуем вам получить свой собственный класс исключения из `std :: exception`.
4. Чтобы обрабатывать исключения, которые могут быть выбраны, реализовать один или несколько блоков `catch` сразу после блока `try`. Каждый блок `catch` определяет тип исключения, с которым он может обращаться.

```

    MyData md;
try {
    // Code that could throw an exception
    md = GetNetworkResource();
}
catch (const networkIOException& e) {
    // Code that executes when an exception of type
    // networkIOException is thrown in the try block
    // ...
    // Log error message in the exception object
    cerr << e.what();
}
catch (const myDataFormatException& e) {
    // Code that handles another exception type
    // ...
    cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
{
    // ...
    if (IOSuccess == false)
        throw networkIOException("Unable to connect");
    // ...
    if (readError)
        throw myDataFormatException("Format error");
    // ...
}

```

Код после предложения try является защищенной секцией кода. Выражение броска бросает - то есть повышает - исключение. Блок кода после предложения catch является обработчиком исключений. Это обработчик, который ловит исключение, которое выбрасывается, если типы в выражениях throw и catch **совместимы**.

```

    try {
        throw CSomeOtherException();
    }
    catch(...) {
        // Catch all exceptions - dangerous!!!
        // Respond (perhaps only partially) to the exception, then
        // re-throw to pass the exception to some other handler
        // ...
        throw;
    }

```

друг (C ++)

1. В некоторых случаях удобнее предоставлять доступ на уровне членов к функциям, которые не являются членами класса или ко всем членам в отдельном классе. Только разработчик класса может объявить, кто его друзья. Функция или класс не могут объявить себя другом любого класса. В определении класса используйте ключевое слово friend и имя функции, не являющейся членом, или другого класса, чтобы предоставить ему доступ к закрытым и защищенным членам вашего класса. В

определении шаблона параметр типа может быть объявлен как друг.

2. Если вы объявляете функцию друга, которая ранее не была объявлена, эта функция экспортируется в охватываемую область неклассификации.

```
class friend F
friend F;
class ForwardDeclared; // Class name is known.
class HasFriends
{
    friend int ForwardDeclared::IsAFriend(); // C2039 error expected
};
```

функции друга

1. Функция друга - это функция, которая не является членом класса, но имеет доступ к закрытым и защищенным членам класса. Дружеские функции не считаются членами класса; они являются нормальными внешними функциями, которым предоставляются особые привилегии доступа.
2. Друзья не входят в сферу действия класса, и они не вызывается с использованием операторов выбора элементов (. И ->), если они не являются членами другого класса.
3. Функция друга объявляется классом, предоставляющим доступ. Объявление друга может быть помещено в любом месте объявления класса. К ним не относятся ключевые слова управления доступом.

```
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
    int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
    Point sPoint;
    sPoint.PrintPrivate();
    ChangePrivate(sPoint);
    sPoint.PrintPrivate();
    // Output: 0
    1
}
```

Члены класса в качестве друзей

```

class B;

class A {
public:
    int Func1( B& b );

private:
    int Func2( B& b );
};

class B {
private:
    int _b;

    // A::Func1 is a friend function to class B
    // so A::Func1 has access to all members of B
    friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }    // OK
int A::Func2( B& b ) { return b._b; }    // C2248

```

Прочитайте Ключевые слова онлайн: <https://riptutorial.com/ru/cplusplus/topic/4891/ключевые-слова>

глава 46: Ключевые слова с переменной переменной

Examples

Const

Спецификатор типа; когда применяется к типу, выдает версию типа `const`. См. [Ключевое слово `const`](#) для получения подробной информации о значении `const`.

```
const int x = 123;
x = 456; // error
int& r = x; // error

struct S {
    void f();
    void g() const;
};
const S s;
s.f(); // error
s.g(); // OK
```

decltype

C++ 11

Устанавливает тип своего операнда, который не оценивается.

- Если операнд `e` является именем без каких-либо дополнительных круглых скобок, то `decltype(e)` является *объявленным типом* `e`.

```
int x = 42;
std::vector<decltype(x)> v(100, x); // v is a vector<int>
```

- Если операндом `e` является доступ к члену класса без каких-либо дополнительных круглых скобок, тогда `decltype(e)` является *объявленным типом* доступного элемента.

```
struct S {
    int x = 42;
};
const S s;
decltype(s.x) y; // y has type int, even though s.x is const
```

- Во всех других случаях `decltype(e)` дает как тип, так и [категорию](#) значений выражения `e` следующим образом:

- Если `e` является значением типа `T`, то `decltype(e)` есть `T&`.

- Если `e` - значение `x` типа `T` , то `decltype(e)` - `T&&` .
- Если `e` является `rvalue` типа `T` , то `decltype(e)` есть `T`

Это включает случай с посторонними скобками.

```
int f() { return 42; }
int& g() { static int x = 42; return x; }
int x = 42;
decltype(f()) a = f(); // a has type int
decltype(g()) b = g(); // b has type int&
decltype((x)) c = x;    // c has type int&, since x is an lvalue
```

C ++ 14

Специальная форма `decltype(auto)` выводит тип переменной из ее инициализатора или возвращаемого типа функции из операторов `return` в ее определении, используя правила вывода типа `decltype` а не `auto` .

```
const int x = 123;
auto y = x;           // y has type int
decltype(auto) z = x; // z has type const int, the declared type of x
```

подписанный

Ключевое слово, которое является частью некоторых имен целых типов.

- При использовании в одиночку подразумевается `int` , так что `signed` , `signed int` и `int` имеют один и тот же тип.
- В сочетании с `char` выдает тип `signed char` , который отличается от `char` , даже если `char` также подписан. `signed char` имеет диапазон, который включает, по меньшей мере, от -127 до +127 включительно.
- В сочетании с `short` , `long` или `long long` , он избыточен, поскольку эти типы уже подписаны.
- `signed` не может быть объединен с `bool` , `wchar_t` , `char16_t` или `char32_t` .

Пример:

```
signed char celsius_temperature;
std::cin >> celsius_temperature;
if (celsius_temperature < -35) {
    std::cout << "cold day, eh?\n";
}
```

неподписанный

Спецификатор типа, который запрашивает неподписанную версию целочисленного типа.

- При использовании в одиночку подразумевается `int` , поэтому `unsigned` имеет тот же

тип, что и `unsigned int` .

- Тип `unsigned char` отличается от `char` типа, даже если `char` без знака. Он может содержать целые числа не менее 255.
- `unsigned` также может сочетаться с `short` , `long` или `long long` . Его нельзя комбинировать с `bool` , `wchar_t` , `char16_t` или `char32_t` .

Пример:

```
char invert_case_table[256] = { ..., 'a', 'b', 'c', ..., 'A', 'B', 'C', ... };
char invert_case(char c) {
    unsigned char index = c;
    return invert_case_table[index];
    // note: returning invert_case_table[c] directly does the
    // wrong thing on implementations where char is a signed type
}
```

летучий

Тип классификатора; когда применяется к типу, создается версия с изменчивой квалификацией. Неустойчивая квалификация играет ту же роль, что и `const` квалификация в системе типов, но `volatile` не препятствует модификации объектов; вместо этого он вынуждает компилятор обрабатывать все обращения к таким объектам, как побочные эффекты.

В приведенном ниже примере, если `memory_mapped_port` не был `volatile`, компилятор мог оптимизировать функцию так, чтобы она выполняла только окончательную запись, которая была бы неправильной, если `sizeof(int)` больше 1. `volatile` квалификация заставляет ее обрабатывать все `sizeof(int)` записывается как разные побочные эффекты и, следовательно, выполняет все их (в порядке).

```
extern volatile char memory_mapped_port;
void write_to_device(int x) {
    const char* p = reinterpret_cast<const char*>(&x);
    for (int i = 0; i < sizeof(int); i++) {
        memory_mapped_port = p[i];
    }
}
```

Прочитайте Ключевые слова с переменной переменной онлайн:

<https://riptutorial.com/ru/cplusplus/topic/7840/ключевые-слова-с-переменной-переменной>

глава 47: Ковариантность типа возврата

замечания

Ковариация параметра или возвращаемого значения для виртуальной функции-члена m , где его тип T становится более специфичным в переопределении производного класса m . Тип T затем изменяется (*дисперсия*) по специфичности таким же образом (*co*), что и классы, обеспечивающие m . C++ обеспечивает языковую поддержку для ковариантных типов возвращаемых данных, которые являются необработанными указателями или исходными ссылками - ковариация предназначена для типа адресата или референта.

Поддержка C++ ограничена возвращаемыми типами, потому что возвращаемые значения функции являются единственными чистыми **out-arguments** в C++, а ковариация - только тип `safe` для чистого аргумента. В противном случае вызывающий код может предоставить объект менее определенного типа, чем ожидает код приема. Профессор Массачусетского технологического института Барбара Лисков изучила эту проблему и связала ее с вопросами безопасности, и теперь она известна как Принцип замещения Лискова или **LSP**.

Ковариантная поддержка существенно помогает избежать опускания и динамического контроля типов.

Поскольку интеллектуальные указатели типа класса не может использовать встроенную поддержку для ковариации непосредственно для смарт результатов указателей, но можно определить по-видимому, *общековариантную* `Непро-virtual` смарт - функцию результата указателя оболочки для ковариантной `virtual` функции, которая производит сырье указателей.

Examples

1. Базовый пример без ковариантных возвратов, показывает, почему они желательны

```
// 1. Base example not using language support for covariance, dynamic type checking.

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;          // Necessary for `delete` via Top*.
};

class D : public Top
{
public:
```



```

        Top* clone() const override
        { return new D( *this ); }
};

class DD : public D
{
private:
    int answer_ = 42;

public:
    int answer() const
    { return answer_; }

    Top* clone() const override
    { return new DD( *this ); }
};

#include <assert.h>
#include <iostream>
#include <typeinfo>
using namespace std;

int main()
{
    cout << boolalpha;

    DD* p1 = new DD();
    Top* p2 = p1->clone();
    bool const correct_dynamic_type = (typeid( *p2 ) == typeid( DD ));
    cout << correct_dynamic_type << endl;           // "true"

    assert( correct_dynamic_type ); // This is essentially dynamic type checking. :(
    auto p2_most_derived = static_cast<DD*>( p2 );
    cout << p2_most_derived->answer() << endl;      // "42"
    delete p2;
    delete p1;
}

```

2. Ковариантная версия результата базового примера, проверка статического типа.

```

// 2. Covariant result version of the base example, static type checking.

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;           // Necessary for `delete` via Top*.
};

class D : public Top
{
public:
    D* /* ← Covariant return */ clone() const override
    { return new D( *this ); }
};

class DD : public D
{

```

```

private:
    int answer_ = 42;

public:
    int answer() const
    { return answer_; }

    DD* /* ← Covariant return */ clone() const override
    { return new DD( *this ); }
};

#include <iostream>
using namespace std;

int main()
{
    DD* p1 = new DD();
    DD* p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;           // "42"
    delete p2;
    delete p1;
}

```

3. Результат ковариантного умного указателя (автоматическая очистка).

```

// 3. Covariant smart pointer result (automated cleanup).

#include <memory>
using std::unique_ptr;

template< class Type >
auto up( Type* p ) { return unique_ptr<Type>( p ); }

class Top
{
private:
    virtual Top* virtual_clone() const = 0;

public:
    unique_ptr<Top> clone() const
    { return up( virtual_clone() ); }

    virtual ~Top() = default;           // Necessary for `delete` via Top*.
};

class D : public Top
{
private:
    D* /* ← Covariant return */ virtual_clone() const override
    { return new D( *this ); }

public:
    unique_ptr<D> /* ← Apparent covariant return */ clone() const
    { return up( virtual_clone() ); }
};

class DD : public D

```

```

{
private:
    int answer_ = 42;

    DD* /* ← Covariant return */ virtual_clone() const override
    { return new DD( *this ); }

public:
    int answer() const
    { return answer_; }

    unique_ptr<DD> /* ← Apparent covariant return */ clone() const
    { return up( virtual_clone() ); }
};

#include <iostream>
using namespace std;

int main()
{
    auto p1 = unique_ptr<DD>(new DD());
    auto p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;          // "42"
    // Cleanup is automated via unique_ptr.
}

```

Прочитайте Ковариантность типа возврата онлайн:

<https://riptutorial.com/ru/cplusplus/topic/5411/ковариантность-типа-возврата>

глава 48: Компиляция и строительство

Вступление

Программы, написанные на C ++, должны быть скомпилированы до их запуска. В зависимости от вашей операционной системы доступно большое количество компиляторов.

замечания

Большинство операционных систем поставляются без компилятора, и их необходимо установить позже. Некоторые общие варианты компиляторов:

- GCC, сборник компиляторов GNU g ++
- clang: интерфейс семейства языков C для LLVM clang ++
- MSVC, Microsoft Visual C ++ (входит в состав Visual Studio) visual-c ++
- C ++ Builder, Embarcadero C ++ Builder (входит в RAD Studio) c ++ builder

Пожалуйста, обратитесь к соответствующему руководству по компиляции, о том, как скомпилировать программу на C ++.

Другой вариант использования конкретного компилятора с его собственной конкретной системой сборки, можно позволить родовым [системам сборки](#) настраивать проект для конкретного компилятора или для установленного по умолчанию.

Examples

Компиляция с помощью GCC

Предполагая, что один исходный файл с именем `main.cpp`, команда для компиляции и ссылки на не оптимизированный исполняемый файл выглядит следующим образом (компиляция без оптимизации полезна для начальной разработки и отладки, хотя `-Og` официально рекомендуется для более новых версий GCC).

```
g++ -o app -Wall main.cpp -O0
```

Чтобы создать оптимизированный исполняемый файл для использования в производстве, используйте один из опций `-O` (см. : `-O1`, `-O2`, `-O3`, `-Os`, `-Ofast`):

```
g++ -o app -Wall -O2 main.cpp
```

Если опция `-O` опущена, `-O0`, что означает отсутствие оптимизаций, используется по умолчанию (указание `-O` без номера разрешается на `-O1`).

Альтернативно, используйте флаги оптимизации из групп `-O` (или более экспериментальных оптимизаций) напрямую. В следующем примере построена оптимизация `-O2` плюс один флаг с уровня оптимизации `-O3` :

```
g++ -o app -Wall -O2 -ftree-partial-pre main.cpp
```

Чтобы создать оптимизированный исполняемый файл для конкретной платформы (для использования на производстве с той же архитектурой), используйте:

```
g++ -o app -Wall -O2 -march=native main.cpp
```

Любой из вышеперечисленных файлов создаст двоичный файл, который можно запустить с `.\app.exe` в Windows и `./app` в Linux, Mac OS и т. Д.

Флаг `-o` также может быть пропущен. В этом случае GCC создаст исполняемый файл `a.exe` по умолчанию `a.exe` Windows и `a.out` в Unix-подобных системах. Чтобы скомпилировать файл без его связывания, используйте параметр `-c` :

```
g++ -o file.o -Wall -c file.cpp
```

Это создает объектный файл с именем `file.o` который впоследствии может быть связан с другими файлами для создания двоичного файла:

```
g++ -o app file.o otherfile.o
```

Подробнее о вариантах оптимизации можно узнать на gcc.gnu.org . Особо следует отметить `-Og` (оптимизация с упором на опыт отладки - рекомендуется для стандартного цикла редактирования-компиляции-отладки) и `-Ofast` (все оптимизации, включая те, которые не соответствуют строгому соблюдению стандартов).

Флаг `-Wall` позволяет предупреждать многие распространенные ошибки и всегда должен использоваться. Для улучшения качества кода часто рекомендуется использовать `-Wextra` и другие предупреждающие флаги, которые автоматически не активируются `-Wall` и `-Wextra` .

Если код ожидает определенный стандарт C ++, укажите, какой стандарт использовать, включая флаг `-std=` . Поддерживаемые значения соответствуют году завершения для каждой версии стандарта ISO C ++. Начиная с GCC 6.1.0, допустимыми значениями для `std=` flag являются `c++98` / `c++03` , `c++11` , `c++14` и `c++17` / `c++1z` . Значения, разделенные косой чертой, эквивалентны.

```
g++ -std=c++11 <file>
```

GCC включает некоторые расширения, специфичные для компилятора, которые отключены, когда они конфликтуют со стандартом, заданным флагом `-std=` . Для компиляции с включенными всеми расширениями может использоваться значение `gnu++XX` ,

где `xx` - любой из лет, используемых значениями `c++` перечисленными выше.

Стандарт по умолчанию будет использоваться, если ни один не указан. Для версий GCC до 6.1.0 по умолчанию используется значение `-std=gnu++03` ; в GCC 6.1.0 и выше по умолчанию используется `-std=gnu++14` .

Обратите внимание, что из-за ошибок в GCC флаг `-pthread` должен присутствовать при компиляции и связывании для GCC для поддержки стандартных функций потоковой `-pthread` C ++, представленных с C ++ 11, таких как `std::thread` и `std::wait_for` . Опускание при использовании функций потоковой передачи может не приводить к **появлению предупреждений, но недействительных результатов** на некоторых платформах.

Связывание с библиотеками:

Используйте опцию `-l` для передачи имени библиотеки:

```
g++ main.cpp -lpcr2-8
#pcr2-8 is the PCRE2 library for 8bit code units (UTF-8)
```

Если библиотека не находится в стандартном пути библиотеки, добавьте путь с параметром `-L` :

```
g++ main.cpp -L/my/custom/path/ -lmylib
```

Несколько библиотек могут быть связаны между собой:

```
g++ main.cpp -lmylib1 -lmylib2 -lmylib3
```

Если одна библиотека зависит от другой, поместите зависимую библиотеку **перед** независимой библиотекой:

```
g++ main.cpp -lchild-lib -lbase-lib
```

Или дайте компоновщику определить само упорядочение через `--start-group` и `--end-group` (обратите внимание: это имеет значительную производительность):

```
g++ main.cpp -Wl,--start-group -lbase-lib -lchild-lib -Wl,--end-group
```

Компиляция с помощью Visual C ++ (командной строки)

Для программистов, поступающих из GCC или Clang в Visual Studio, или программистов, более удобных с командной строкой в целом, вы можете использовать компилятор Visual C ++ из командной строки, а также из среды IDE.

Если вы хотите скомпилировать свой код из командной строки в Visual Studio, сначала необходимо настроить среду командной строки. Это можно сделать, открыв [Visual Studio Command Prompt / Developer Command Prompt / x86 Native Tools Command Prompt / x64 Native Tools Command Prompt](#) или [аналогичная](#) (как предоставляется вашей версией Visual Studio) или в командной строке, путем перехода к подкаталог `vc` каталога установки компилятора (обычно `\Program Files (x86)\Microsoft Visual Studio x\VC`, где `x` - номер версии (например, 10.0 для 2010 или 14.0 для 2015) и запуск командного файла `VCVARSALL` с помощью параметр командной строки, указанный [здесь](#).

Обратите внимание, что в отличие от GCC, Visual Studio не предоставляет интерфейс для компоновщика (`link.exe`) через компилятор (`cl.exe`), но вместо этого предоставляет компоновщик как отдельную программу, которую компилятор вызывает при выходе. `cl.exe` и `link.exe` могут использоваться отдельно с разными файлами и параметрами, иначе `cl` может передаваться файлы и параметры для `link` если обе задачи выполняются вместе. Любые параметры привязки, указанные для `cl` будут переведены в параметры `link`, и любые файлы, не обработанные `cl` будут переданы непосредственно в `link`. Поскольку это в основном простое руководство по компиляции с командной строкой Visual Studio, аргументы для `link` не будут описаны в настоящее время; если вам нужен список, см. [здесь](#).

Обратите внимание, что аргументы для `cl` чувствительны к регистру, а аргументы для `link` - нет.

[Следует иметь в виду, что в некоторых из следующих примеров при указании абсолютных имен пути используется переменная «текущий каталог» оболочки Windows, `%cd%`. Для тех, кто не знаком с этой переменной, он расширяется до текущего рабочего каталога. Из командной строки это будет каталог, в котором вы были, когда вы запускаете `cl`, и указывается в командной строке по умолчанию (если ваша командная строка - `C:\src>`, например, то `%cd%` - `C:\src\`).]

Предполагая, что в текущей папке имеется один исходный файл с именем `main.cpp`, команда для компиляции и ссылки на неоптимизированный исполняемый файл (полезный для начальной разработки и отладки) является (используйте любой из следующих):

```
cl main.cpp
// Generates object file "main.obj".
// Performs linking with "main.obj".
// Generates executable "main.exe".

cl /Od main.cpp
// Same as above.
// "/Od" is the "Optimisation: disabled" option, and is the default when no /O is specified.
```

Предположим, что дополнительный файл-источник «`nam.cpp`» в том же каталоге использует следующее:

```
cl main.cpp niam.cpp
// Generates object files "main.obj" and "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

Вы также можете использовать подстановочные знаки, как и следовало ожидать:

```
cl main.cpp src\*.cpp
// Generates object file "main.obj", plus one object file for each ".cpp" file in folder
// "%cd%\src".
// Performs linking with "main.obj", and every additional object file generated.
// All object files will be in the current folder.
// Generates executable "main.exe".
```

Чтобы переименовать или переместить исполняемый файл, выполните одно из следующих действий:

```
cl /o name main.cpp
// Generates executable named "name.exe".

cl /o folder\ main.cpp
// Generates executable named "main.exe", in folder "%cd%\folder".

cl /o folder\name main.cpp
// Generates executable named "name.exe", in folder "%cd%\folder".

cl /Fename main.cpp
// Same as "/o name".

cl /Fefolder\ main.cpp
// Same as "/o folder\".

cl /Fefolder\name main.cpp
// Same as "/o folder\name".
```

Оба параметра `/o` и `/Fe` передают свой параметр (назовем его `o-param`), чтобы `link` как `/OUT:o-param`, добавив соответствующее расширение (обычно `.exe` или `.dll`) в «имя» `o-param` по мере необходимости. Хотя оба `/o` и `/Fe`, насколько мне известно, идентичны по функциональности, последний предпочтительнее для Visual Studio. `/o` отмечен как устаревший и, как представляется, в основном предоставляется программистам, более знакомым с GCC или Clang.

Обратите внимание, что, хотя пространство между `/o` и указанной папкой и `/` или именем является необязательным, между `/Fe` и указанным папок и `/` или именем не может быть пробела.

Аналогично, для создания оптимизированного исполняемого файла (для использования в производстве) используйте:

```
cl /O1 main.cpp
// Optimise for executable size. Produces small programs, at the possible expense of slower
```



```
// execution.

cl /O2 main.cpp
// Optimise for execution speed.  Produces fast programs, at the possible expense of larger
// file size.

cl /GL main.cpp other.cpp
// Generates special object files used for whole-program optimisation, which allows CL to
// take every module (translation unit) into consideration during optimisation.
// Passes the option "/LTCG" (Link-Time Code Generation) to LINK, telling it to call CL during
// the linking phase to perform additional optimisations.  If linking is not performed at
// this
// time, the generated object files should be linked with "/LTCG".
// Can be used with other CL optimisation options.
```

Наконец, чтобы создать оптимизированный исполняемый файл для конкретной платформы (для использования на производстве на машине с указанной архитектурой), выберите соответствующую командную строку или параметр `VCVARSALL` для целевой платформы. `link` должна обнаружить желаемую платформу из объектных файлов; если нет, используйте параметр `/MACHINE` чтобы явно указать целевую платформу.

```
// If compiling for x64, and LINK doesn't automatically detect target platform:
cl main.cpp /link /machine:X64
```

Любое из вышеперечисленного даст исполняемый файл с именем, указанным `/o` или `/Fe`, или если ни один из них не указан, с именем, идентичным первому источнику или объекту, указанному компилятору.

```
cl a.cpp b.cpp c.cpp
// Generates "a.exe".

cl d.obj a.cpp q.cpp
// Generates "d.exe".

cl y.lib n.cpp o.obj
// Generates "n.exe".

cl /o yo zp.obj pz.cpp
// Generates "yo.exe".
```

Чтобы скомпилировать файл (ы) без ссылки, используйте:

```
cl /c main.cpp
// Generates object file "main.obj".
```

Это говорит `cl` для выхода без вызова `link` и создает объектный файл, который позже может быть связан с другими файлами для создания двоичного файла.

```
cl main.obj niam.cpp
// Generates object file "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
```

```
// Generates executable "main.exe".

link main.obj niam.obj
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

Существуют и другие важные параметры командной строки, которые было бы очень полезно для пользователей:

```
cl /EHsc main.cpp
// "/EHsc" specifies that only standard C++ ("synchronous") exceptions will be caught,
// and `extern "C"` functions will not throw exceptions.
// This is recommended when writing portable, platform-independent code.

cl /clr main.cpp
// "/clr" specifies that the code should be compiled to use the common language runtime,
// the .NET Framework's virtual machine.
// Enables the use of Microsoft's C++/CLI language in addition to standard ("native") C++,
// and creates an executable that requires .NET to run.

cl /Za main.cpp
// "/Za" specifies that Microsoft extensions should be disabled, and code should be
// compiled strictly according to ISO C++ specifications.
// This is recommended for guaranteeing portability.

cl /Zi main.cpp
// "/Zi" generates a program database (PDB) file for use when debugging a program, without
// affecting optimisation specifications, and passes the option "/DEBUG" to LINK.

cl /LD dll.cpp
// "/LD" tells CL to configure LINK to generate a DLL instead of an executable.
// LINK will output a DLL, in addition to an LIB and EXP file for use when linking.
// To use the DLL in other programs, pass its associated LIB to CL or LINK when compiling
those
// programs.

cl main.cpp /link /LINKER_OPTION
// "/link" passes everything following it directly to LINK, without parsing it in any way.
// Replace "/LINKER_OPTION" with any desired LINK option(s).
```

Для тех, кто более знаком с системами *nix и / или GCC / Clang, `cl`, `link` и другими инструментами командной строки Visual Studio, могут принимать параметры, указанные с помощью дефиса (например, `-c`), вместо косой черты (например, `/c`). Кроме того, Windows распознает либо косую черту, либо обратную косую черту как допустимый разделитель путей, поэтому можно использовать также пути в стиле *nix. Это упрощает преобразование простых команд компилятора из `g++` или `clang++` в `cl`, или наоборот, с минимальными изменениями.

```
g++ -o app src/main.cpp
cl -o app src/main.cpp
```

Конечно, при переносе командных строк, которые используют более сложные параметры

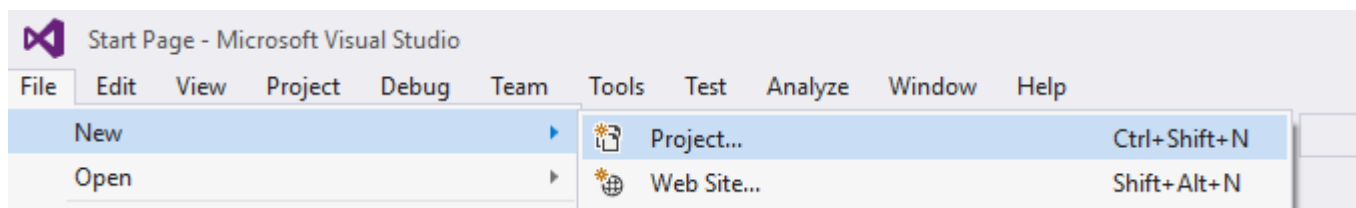
g++ или clang++ , вам нужно искать эквивалентные команды в применимых документах компилятора и / или на ресурсных сайтах, но это облегчает работу с минимальным временем обучения новые компиляторы.

Если вам нужны специальные языковые функции для вашего кода, требуется определенная версия MSVC. Из [Visual C ++ 2015 Update 3](#) можно выбрать версию стандарта для компиляции с помощью флага /std . Возможные значения: /std:c++14 и /std:c++latest (/std:c++17 скоро появится).

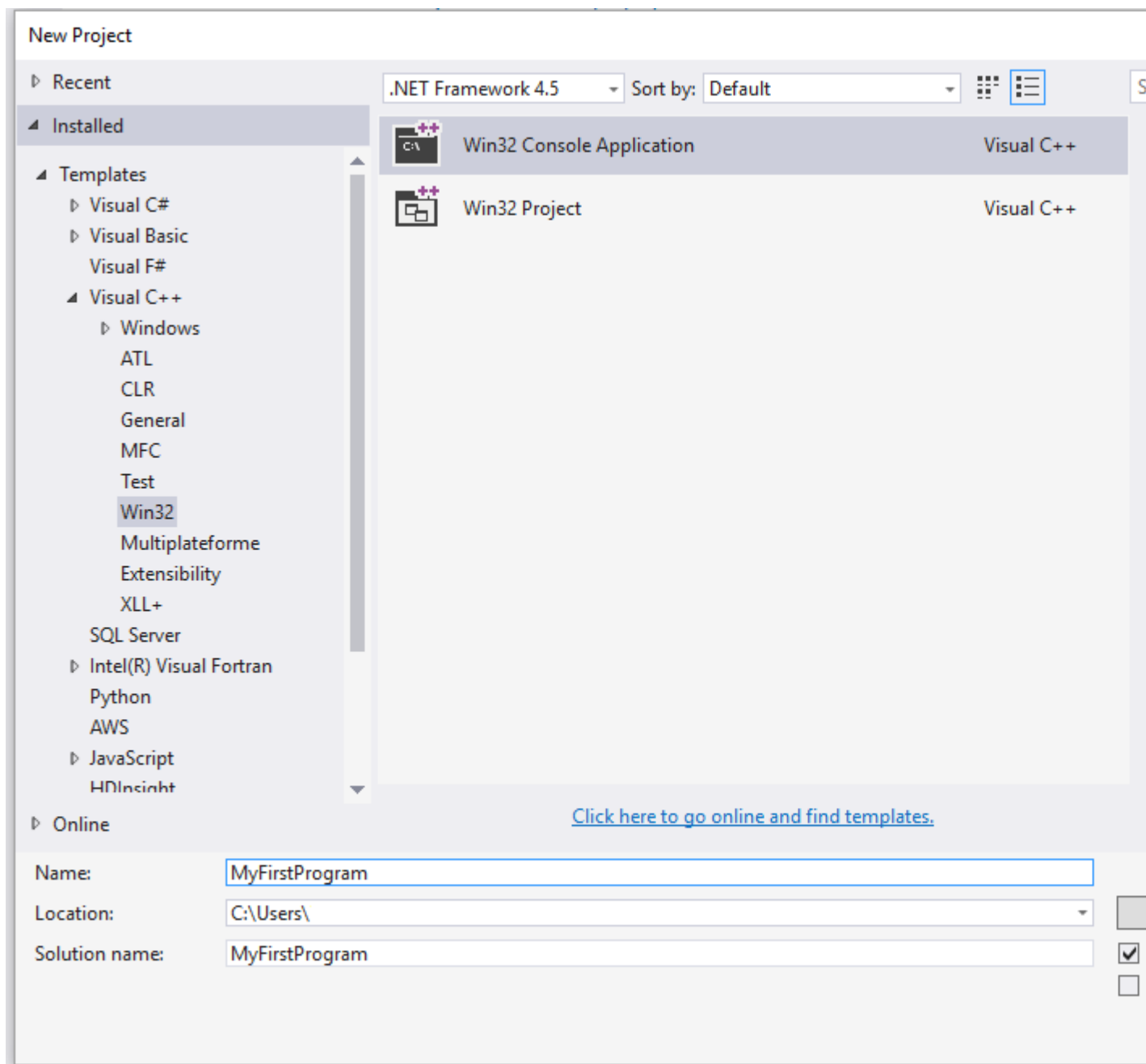
Примечание. В более старых версиях этого компилятора были доступны специальные флаги функций, однако в основном это использовалось для предварительного просмотра новых функций.

Компиляция с помощью Visual Studio (графический интерфейс) - Hello World

1. Загрузите и установите [Visual Studio Community 2015](#)
2. Открыть сообщество Visual Studio
3. Нажмите Файл -> Создать -> Проект

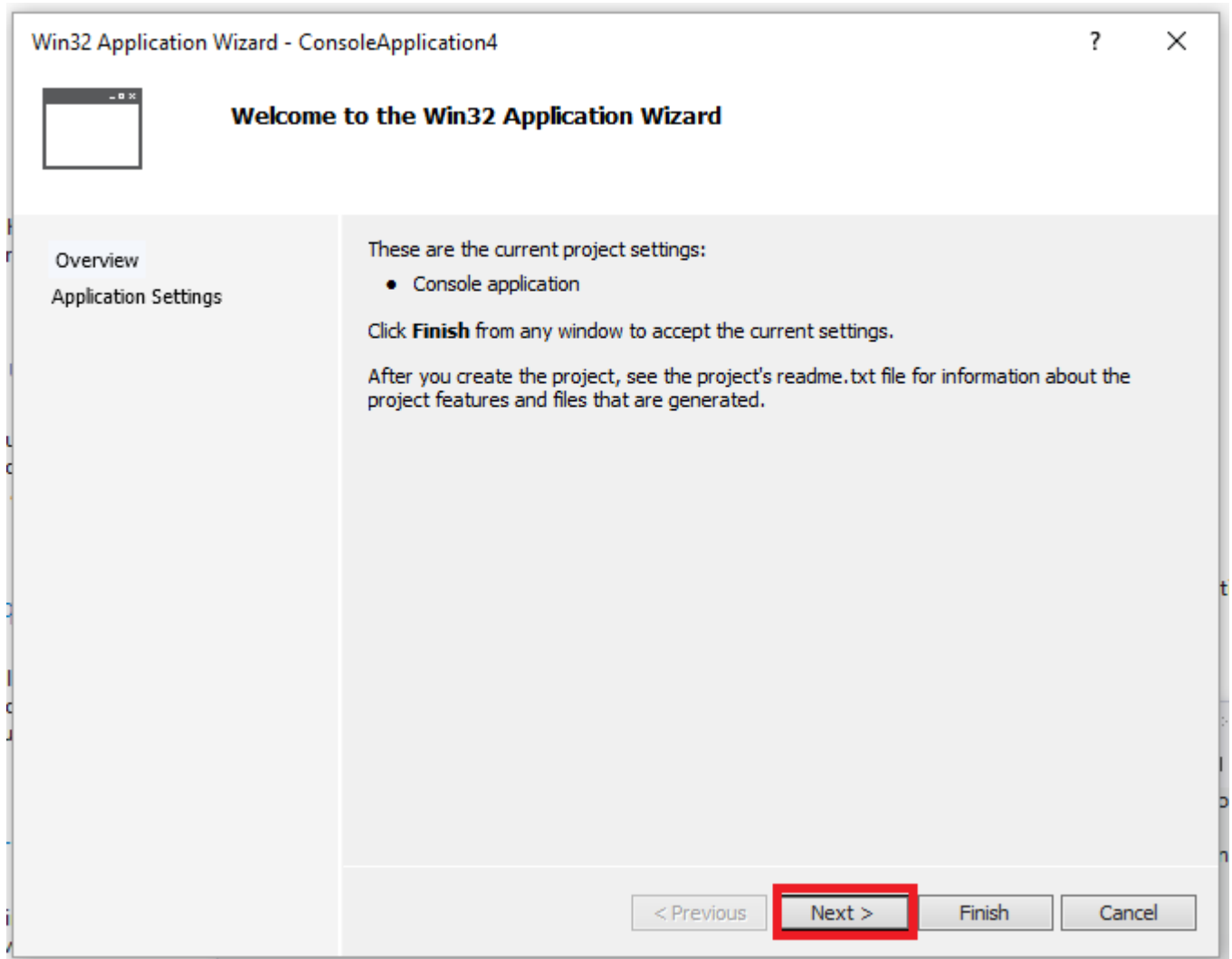


4. Нажмите Шаблоны -> Visual C ++ -> Консольное приложение Win32, а затем назовите проект **MyFirstProgram** .

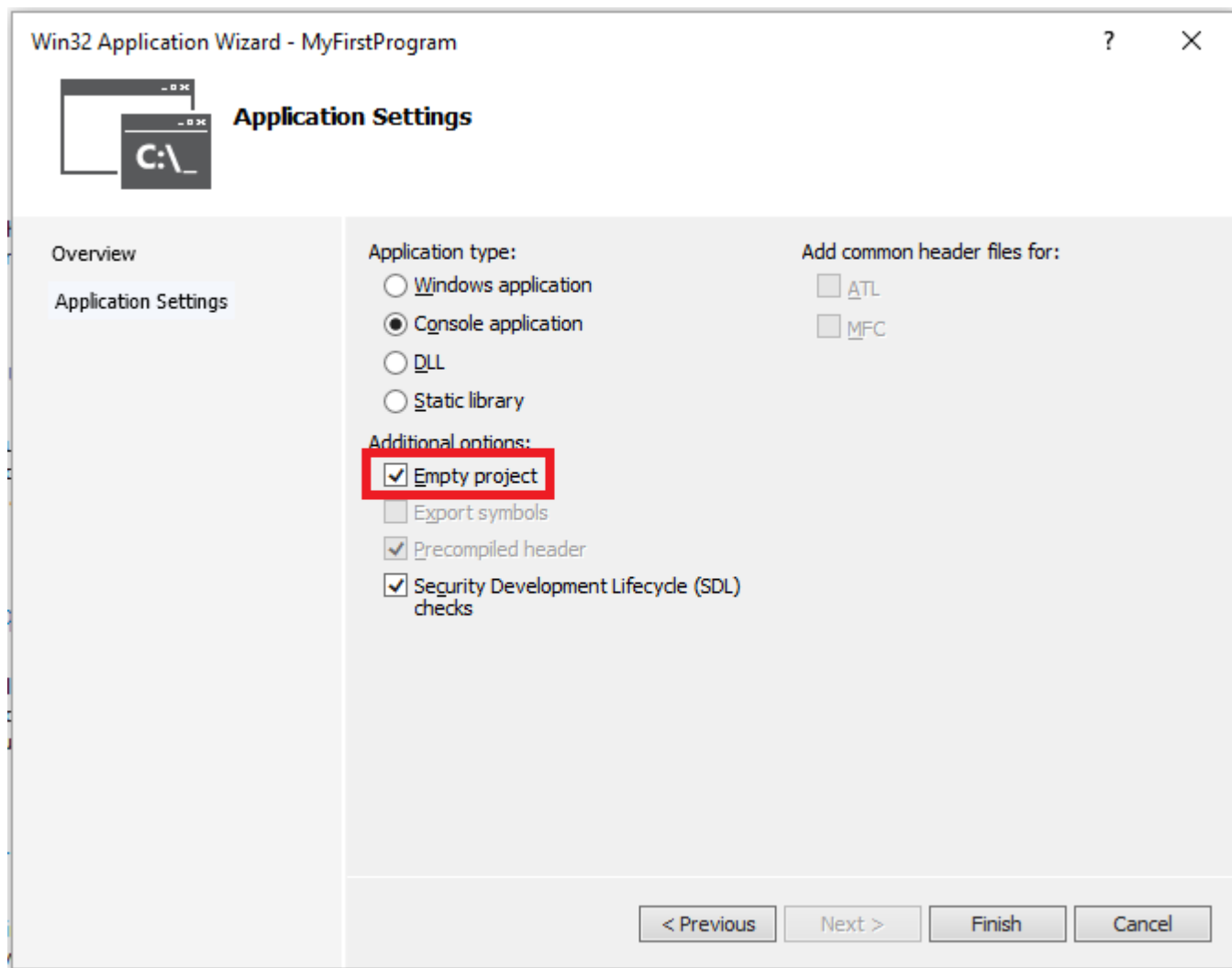


5. Нажмите OK.

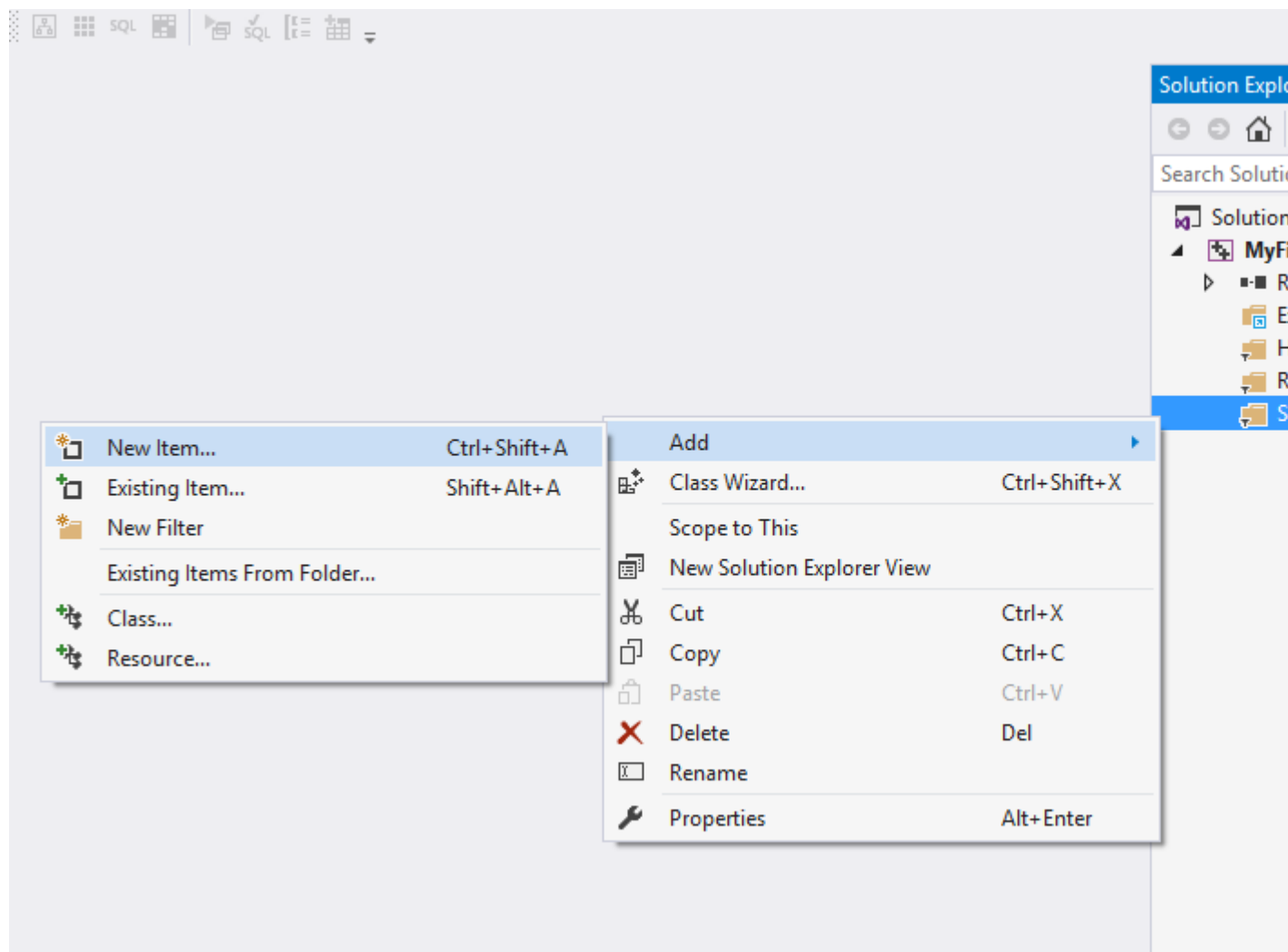
6. Нажмите «Далее» в следующем окне.



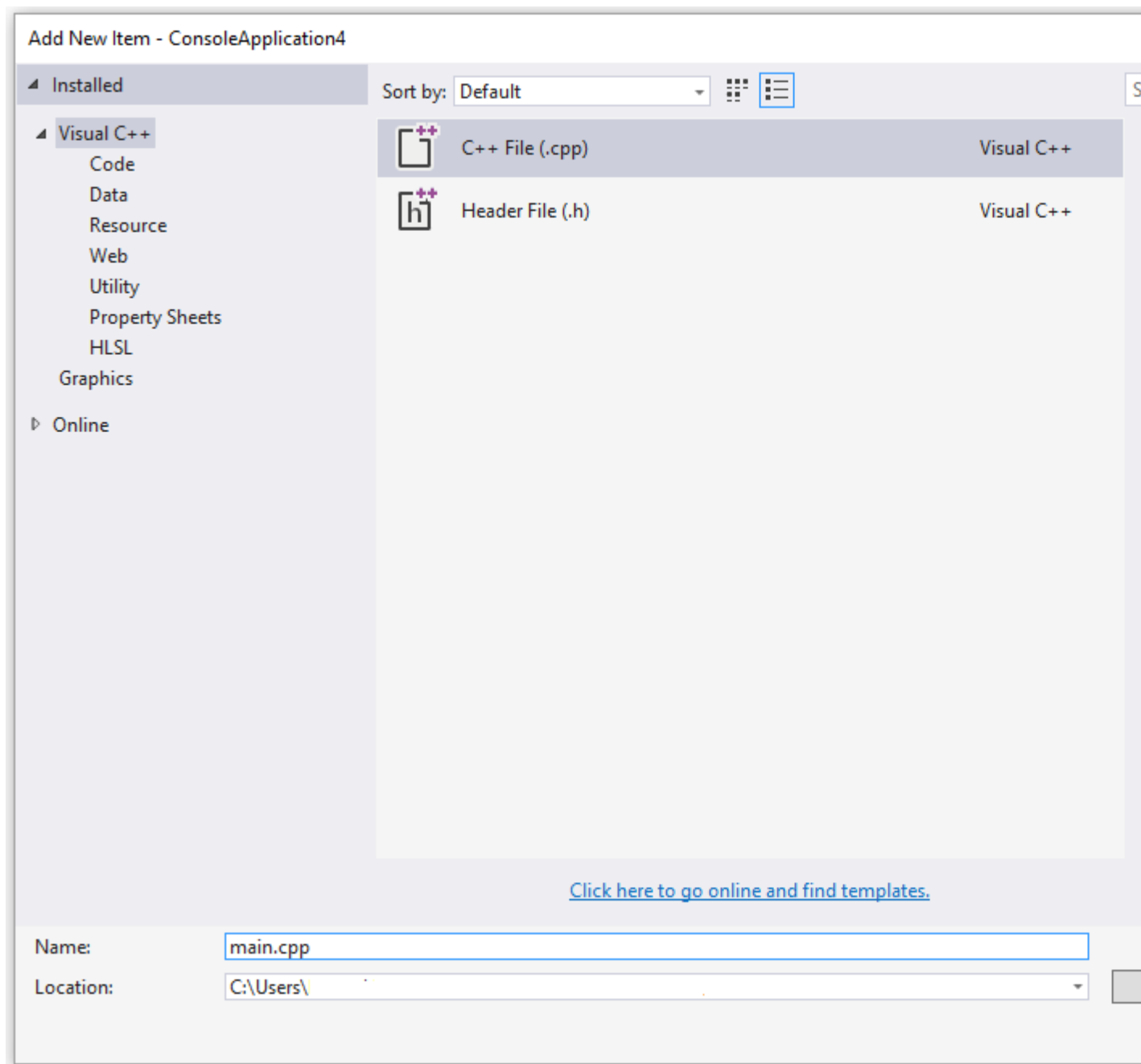
7. Проверьте флажок « Empty project » и нажмите «Готово»:



8. Щелкните правой кнопкой мыши папку Source Source затем -> Добавить -> Новый элемент:



9. Выберите файл C ++ и назовите файл main.cpp, затем нажмите «Добавить»:

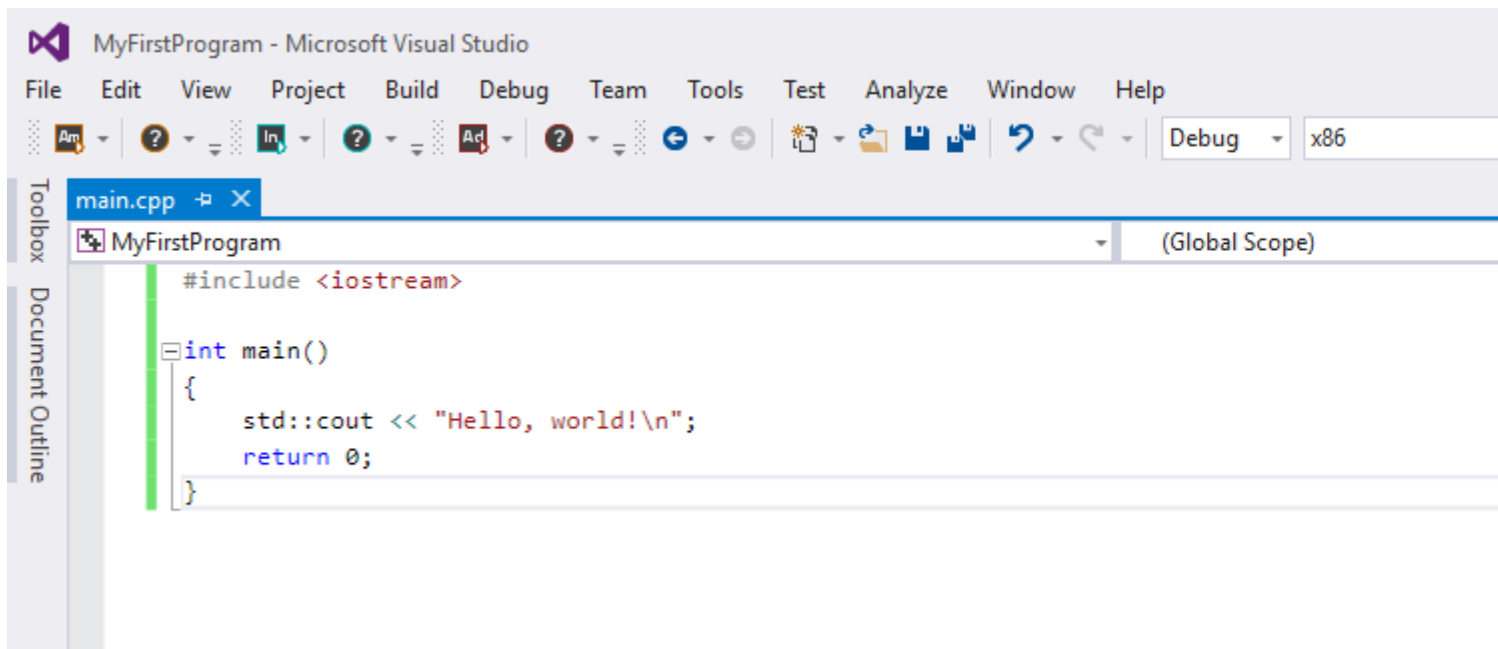


10: Скопируйте и вставьте следующий код в новый файл main.cpp:

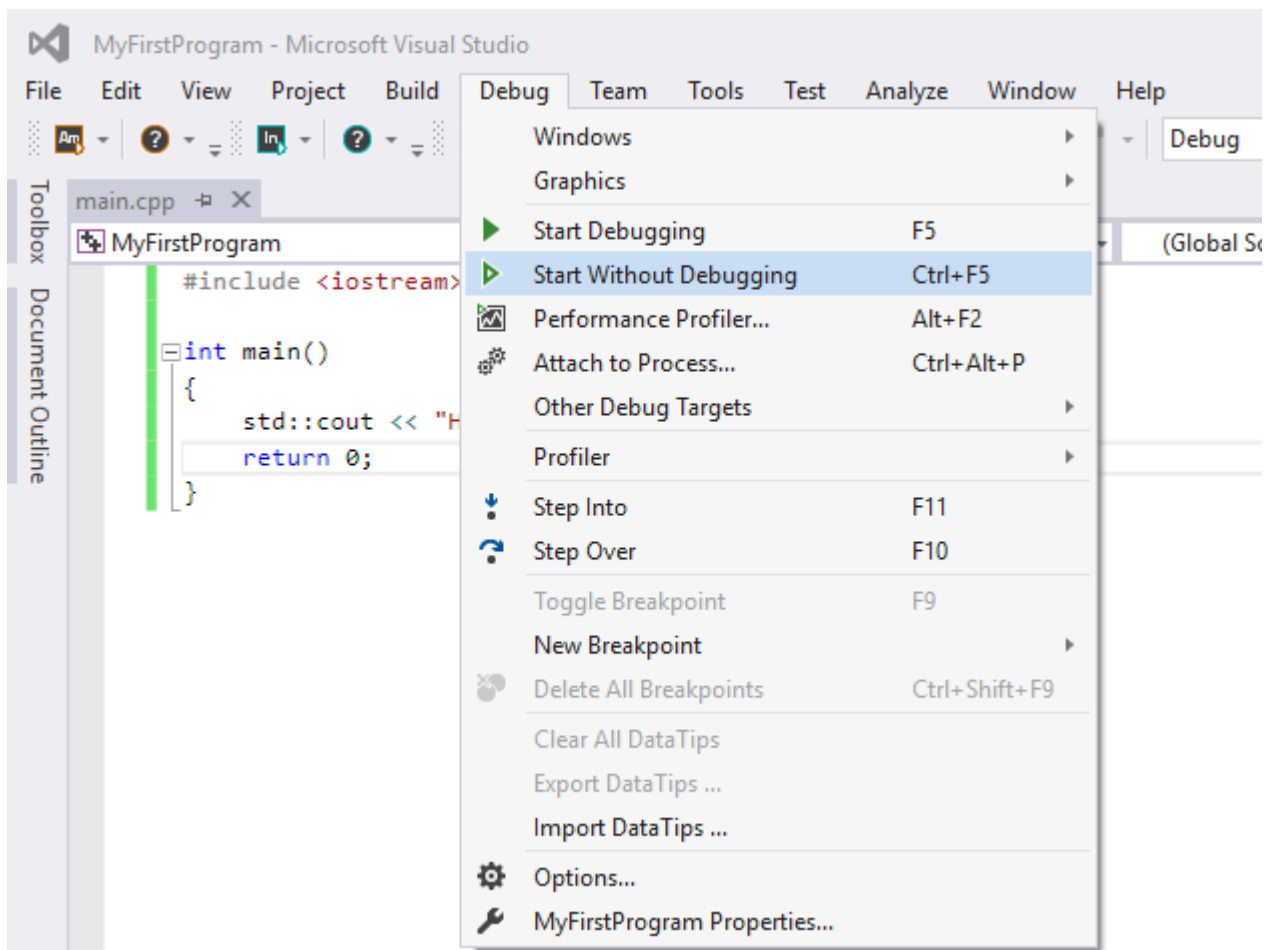
```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

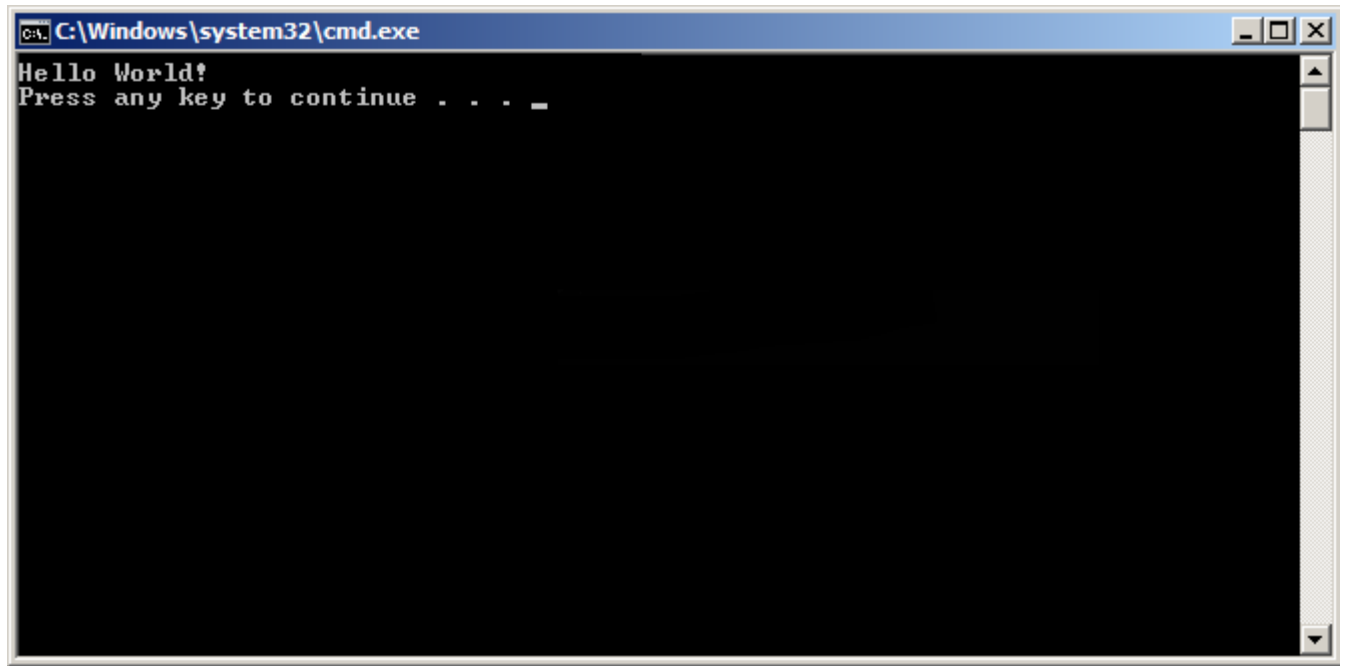
Вы должны выглядеть так:



11. Нажмите «Отладка» -> «Старт без отладки» (или нажмите ctrl + F5):



12. Готово. Вы должны получить следующий вывод консоли:



Компиляция с Clang

Поскольку front-end [Clang](#) предназначен для совместимости с GCC, большинство программ, которые могут быть скомпилированы через [GCC](#), будут скомпилированы, когда вы замените `g++` на `clang++` в сценариях сборки. Если по `-std=version` задана, будет использоваться `gnu11`.

Пользователи Windows, которые используются для [MSVC](#), могут поменять `cl.exe` на `clang-cl.exe`. По умолчанию `clang` пытается быть совместимым с самой высокой версией MSVC, которая была установлена.

В случае компиляции с визуальной студией `clang-cl` можно использовать, изменив `Platform toolset` в свойствах проекта.

В обоих случаях `clang` совместим только с его интерфейсом, хотя он также пытается генерировать двоичные файлы совместимых объектов. Пользователи `clang-cl` должны заметить, что [совместимость с MSVC еще не завершена](#).

Чтобы использовать `clang` или `clang-cl`, можно использовать установку по умолчанию в некоторых дистрибутивах Linux или в комплекте с IDE (например, XCode на Mac). Для других версий этого компилятора или на платформах, для которых это не установлено, это можно загрузить с [официальной страницы загрузки](#).

Если вы используете CMake для создания кода, вы обычно можете переключать компилятор, задав переменные окружения `CC` и `CXX` следующим образом:

```
mkdir build
cd build
CC=clang CXX=clang++ cmake ..
cmake --build .
```

См. Также [введение в Stake](#) .

Компиляторы онлайн

Различные веб-сайты предоставляют онлайн-доступ к компиляторам C ++. Набор функций онлайн-компилятора значительно варьируется от сайта к сайту, но обычно они позволяют сделать следующее:

- Вставьте свой код в веб-форму в браузере.
- Выберите некоторые параметры компилятора и скомпилируйте код.
- Собрать компилятор и / или выпуск программы.

Онлайн-поведение веб-компилятора обычно является довольно ограничительным, поскольку они позволяют кому-либо запускать компиляторы и выполнять произвольный код на своей стороне сервера, тогда как обычно удаленное выполнение произвольного кода считается уязвимостью.

Компиляторы онлайн могут быть полезны для следующих целей:

- Запустите небольшой фрагмент кода с компьютера, на котором отсутствует компилятор C ++ (смартфоны, планшеты и т. Д.).
- Убедитесь, что код успешно компилируется с разными компиляторами и работает одинаково, независимо от компилятора, с которым он был скомпилирован.
- Изучайте или преподавайте основы C ++.
- Изучите современные возможности C ++ (C ++ 14 и C ++ 17 в ближайшем будущем), когда современный компилятор C ++ недоступен на локальной машине.
- Найдите ошибку в своем компиляторе по сравнению с большим набором других компиляторов. Проверьте, исправлена ли ошибка компилятора в будущих версиях, которые недоступны на вашем компьютере.
- Решите проблемы онлайн-судьи.

Какие онлайн-компиляторы **не** должны использоваться для:

- Разработка полнофункциональных (даже небольших) приложений с использованием C ++. Обычно онлайн-компиляторы не позволяют связываться со сторонними библиотеками или загружать артефакты сборки.
- Выполнять интенсивные вычисления. Ресурсы на стороне Sever ограничены, поэтому любая пользовательская программа будет убита через несколько секунд после ее выполнения. Допустимое время выполнения обычно достаточно для тестирования и обучения.
- Сам сервер компилятора атаки или сторонние хосты в сети.

Примеры:

Отказ от ответственности: автор (ы) документации не связаны ни с какими

ресурсами, перечисленными ниже. Веб-сайты перечислены в алфавитном порядке.

- <http://codepad.org/> Онлайн-компилятор с совместным использованием кода. Редактирование кода после компиляции с предупреждением или ошибкой исходного кода работает не так хорошо.
- <http://coliru.stacked-crooked.com/> Онлайн-компилятор, для которого вы указываете командную строку. Предоставляет компиляторы GCC и Clang для использования.
- <http://cpp.sh/> - Онлайн-компилятор с поддержкой C ++ 14. Не позволяет редактировать командную строку компилятора, но некоторые параметры доступны через элементы управления графическим интерфейсом.
- <https://gcc.godbolt.org/> - Предоставляет широкий список версий, архитектуры и разборки компилятора. Очень полезно, когда вам нужно проверить, что ваш код компилируется разными компиляторами. GCC, Clang, MSVC (`CL`), компилятор Intel (`icc`), ELLCC и Zарсс, причем один или несколько из этих компиляторов доступны для ARM, ARMv8 (как ARM64), Atmel AVR, MIPS, MIPS64, MSP430, PowerPC , x86 и x64 architectures. Аргументы командной строки компилятора могут быть отредактированы.
- <https://ideone.com/> - Широко используется в сети для иллюстрации поведения фрагмента кода. Предоставляет GCC и Clang для использования, но не позволяет редактировать командную строку компилятора.
- <http://melpon.org/wandbox> - Поддерживает многочисленные версии компилятора Clang и GNU / GCC.
- <http://onlinegdb.com/> - крайне минималистичная среда разработки, включающая редактор, компилятор (gcc) и отладчик (gdb).
- <http://rextester.com/> - Предоставляет компиляторы Clang, GCC и Visual Studio для C и C ++ (наряду с компиляторами для других языков), с доступной для использования библиотекой Boost.
- http://tutorialspoint.com/compile_cpp11_online.php - полнофункциональная оболочка UNIX с GCC и удобный для пользователя проект.
- <http://webcompiler.cloudapp.net/> - Компилятор Online Visual Studio 2015, предоставленный Microsoft в составе RiSE4fun.

Процесс компиляции C ++

Когда вы разрабатываете программу на C ++, следующим шагом является компиляция программы перед ее запуском. Компиляция - это процесс, который преобразует программу, написанную на удобочитаемом языке, такую ​​как C, C ++ и т. Д., В машинный код, который прямо понимается Центральным процессором. Например, если у вас есть исходный код C ++ с именем `prog.cpp`, и вы выполняете команду компиляции,

```
g++ -Wall -ansi -o prog prog.cpp
```

Существует четыре основных этапа, связанных с созданием исполняемого файла из

исходного файла.

1. Препроцессор C ++ принимает файл исходного кода C ++ и обрабатывает заголовки (`#include`), макросы (`#define`) и другие директивы препроцессора.
2. Расширенный исходный код C ++, созданный препроцессором C ++, скомпилирован на язык ассемблера для платформы.
3. Код ассемблера, сгенерированный компилятором, собран в код объекта для платформы.
4. Файл объектного кода, созданный ассемблером, связан вместе с объектными кодами для любых библиотечных функций, используемых для создания библиотеки или исполняемого файла.

предварительная обработка

Препроцессор обрабатывает директивы препроцессора, такие как `#include` и `#define`. Это агностик синтаксиса C ++, поэтому его следует использовать с осторожностью.

Он работает в одном исходном файле C ++ за раз, заменяя директивы `#include` содержимым соответствующих файлов (обычно это просто объявления), выполняет замену макросов (`#define`) и выбирает разные части текста в зависимости от `#if`, `#ifdef` и `#ifndef`.

Препроцессор работает над потоком токенов предварительной обработки. Макрозамена определяется как замена токенов другими токенами (оператор `##` позволяет слить два токена, когда это имеет смысл).

После этого препроцессор создает один вывод, который представляет собой поток токенов, полученный в результате описанных выше преобразований. Он также добавляет некоторые специальные маркеры, которые сообщают компилятору, откуда появилась каждая строка, чтобы он мог использовать их для создания разумных сообщений об ошибках.

На этом этапе могут быть сделаны некоторые ошибки с умным использованием директив `#if` и `#error`.

Используя ниже флаг компилятора, мы можем остановить процесс на этапе предварительной обработки.

```
g++ -E prog.cpp
```

компиляция

Шаг компиляции выполняется на каждом выходе препроцессора. Компилятор анализирует чистый исходный код C ++ (теперь без каких-либо препроцессорных директив) и преобразует его в код сборки. Затем вызывает базовый сервер (ассемблер в `toolchain`),

который собирает этот код в машинный код, производящий фактический двоичный файл в некотором формате (ELF, COFF, a.out, ...). Этот объектный файл содержит скомпилированный код (в двоичной форме) символов, определенных на входе. Символы в объектных файлах называются по имени.

Файлы объектов могут ссылаться на символы, которые не определены. Это тот случай, когда вы используете декларацию и не предоставляете определение для нее. Компилятор не возражает против этого и будет с удовольствием создавать объектный файл до тех пор, пока исходный код будет правильно сформирован.

Компиляторы обычно позволяют прекратить компиляцию на этом этапе. Это очень полезно, потому что с ним вы можете скомпилировать каждый исходный код отдельно. Преимущество этого заключается в том, что вам не нужно перекомпилировать все, если вы меняете только один файл.

Созданные объектные файлы могут быть помещены в специальные архивы, называемые статическими библиотеками, для более легкого повторного использования позже.

На этом этапе сообщается о «регулярных» ошибках компилятора, таких как синтаксические ошибки или ошибки с ошибками при перегрузке.

Чтобы остановить процесс после этапа компиляции, мы можем использовать параметр -S:

```
g++ -Wall -ansi -S prog.cpp
```

монтаж

Ассемблер создает объектный код. В системе UNIX вы можете видеть файлы с суффиксом .o (.OBJ на MSDOS), чтобы указать файлы объектных кодов. На этом этапе ассемблер преобразует эти объектные файлы из кода сборки в инструкции машинного уровня, а созданный файл является перемещаемым объектным кодом. Следовательно, на этапе компиляции создается перемещаемая объектная программа, и эта программа может использоваться в разных местах без повторной компиляции.

Чтобы остановить процесс после шага сборки, вы можете использовать опцию -c:

```
g++ -Wall -ansi -c prog.cpp
```

соединение

Линкером является то, что дает окончательный результат компиляции из файлов объектов, созданных ассемблером. Этот вывод может быть либо общей (или динамической) библиотекой (и, хотя имя похоже, они не имеют много общего со статическими библиотеками, упомянутыми ранее), либо исполняемый файл.

Он связывает все объектные файлы, заменяя ссылки на неопределенные символы

правильными адресами. Каждый из этих символов может быть определен в других объектных файлах или в библиотеках. Если они определены в библиотеках, отличных от стандартной библиотеки, вам нужно сообщить об этом компоновщику.

На этом этапе наиболее распространенными ошибками являются отсутствующие определения или дублирующие определения. Первое означает, что либо определения не существуют (т.е. они не написаны), либо что объектные файлы или библиотеки, в которых они находятся, не были предоставлены компоновщику. Последнее очевидно: один и тот же символ был определен в двух разных объектных файлах или библиотеках.

Компиляция с помощью кода :: Блоки (графический интерфейс)

1. Загрузите и установите Code :: Blocks [здесь](#) . Если вы находитесь в Windows, будьте осторожны, чтобы выбрать файл, для которого имя содержит `mingw` , другие файлы не устанавливают компилятор.
2. Откройте код :: Блоки и нажмите «Создать новый проект»:

Start here - Code::Blocks 16.01

File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plug



Management x

Projects Symbols

Workspace

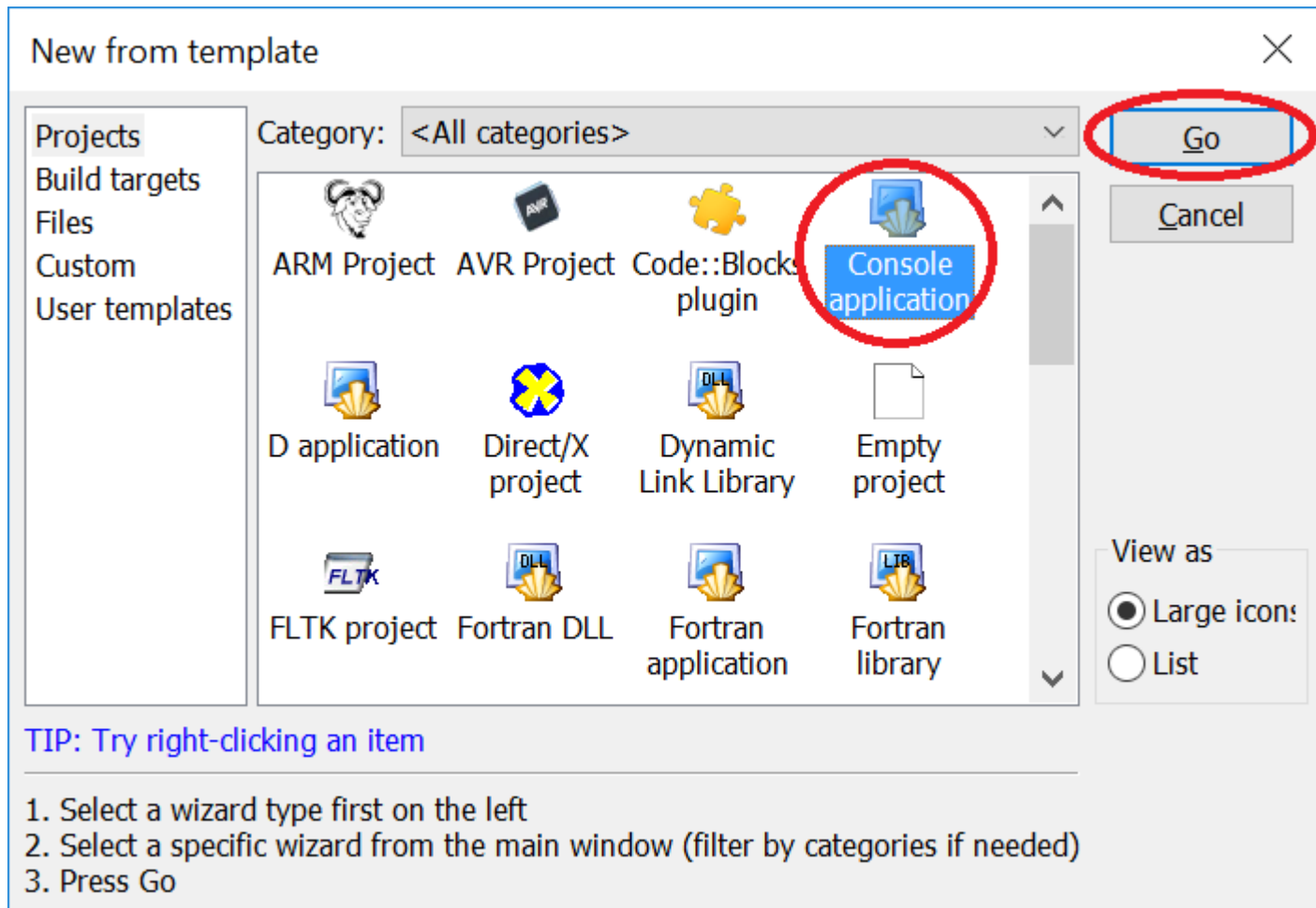
Start here x

Logs & others

Code::Blocks x Search results x Cccc x Build

Start here

3. Выберите «Консольное приложение» и нажмите «Перейти»:



4. Нажмите «Далее», выберите «С ++», нажмите «Далее», выберите имя для своего проекта и выберите папку для ее сохранения, нажмите «Далее» и затем нажмите «Готово».

нажмите «Далее», выберите имя для своего проекта и выберите папку для ее сохранения, нажмите «Далее» и затем нажмите «Готово».

5. Теперь вы можете редактировать и компилировать свой код. Код по умолчанию, который печатает "Hello world!" в консоли уже есть. Чтобы скомпилировать и / или запустить вашу программу, нажмите одну из трех кнопок компиляции / запуска на панели инструментов:

main.cpp [df] - Code::Blocks 16.01

File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plug



Management

Projects Symbols

Workspace
df
Sources
main.cpp

main.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      cout << "Hello world!" << endl;
8      return 0;
9  }
10
```

Logs & others

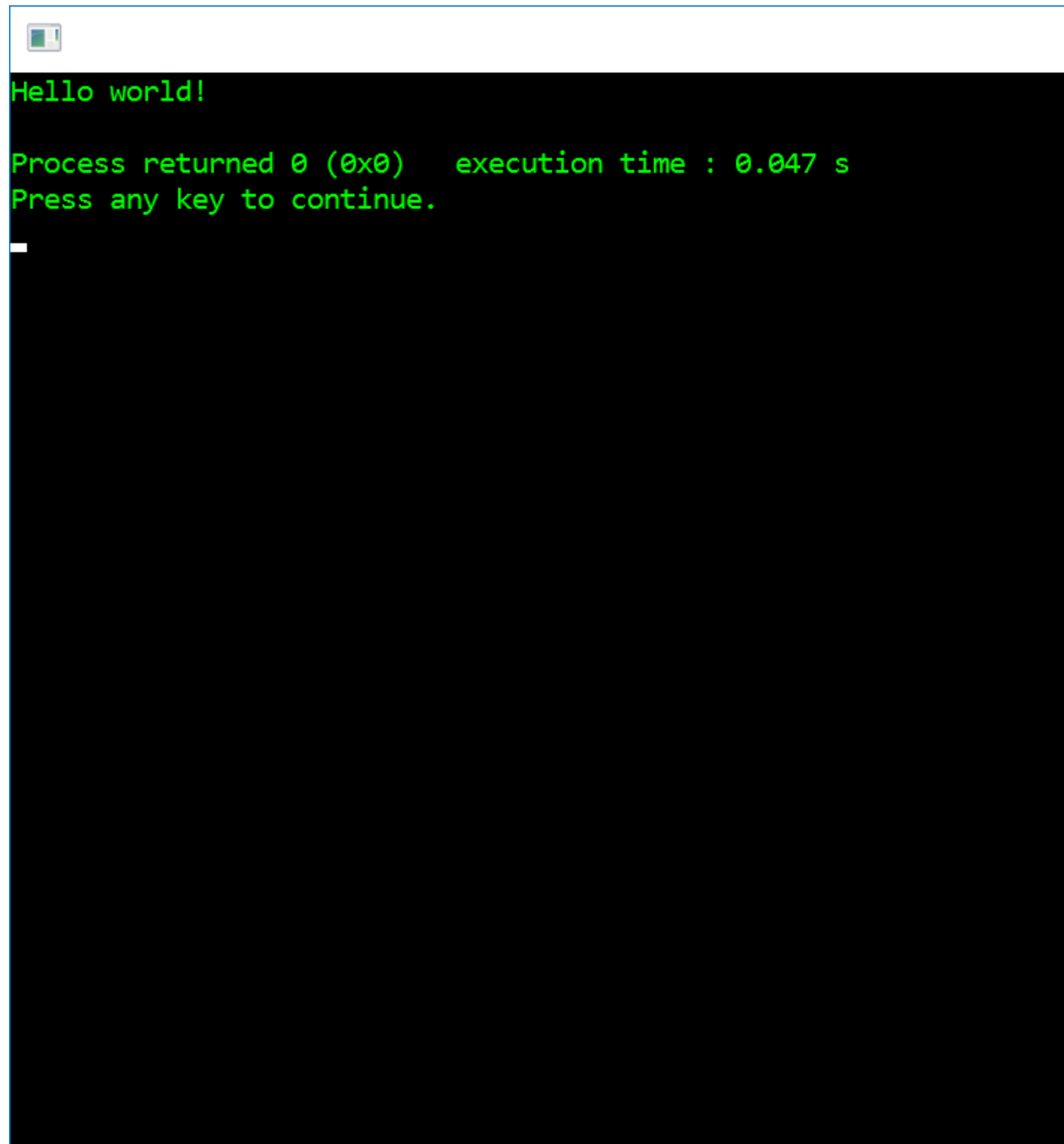
Code::Blocks Search results Cccc Build

C:\Users\glind\Desktop\df\main.cpp

Для компиляции без запуска нажмите  , для запуска без компиляции снова нажмите  и компилировать, а затем запускать, нажмите  ,

для запуска без компиляции снова нажмите  и компилировать, а затем запускать, нажмите  ,

Компиляция и запуск по умолчанию «Hello world!» код дает следующий результат:



```
Hello world!  
  
Process returned 0 (0x0)   execution time : 0.047 s  
Press any key to continue.  
_
```

Прочитайте [Компиляция и строительство онлайн:](#)

<https://riptutorial.com/ru/cplusplus/topic/4708/компиляция-и-строительство>

глава 49: Константные функции члена класса

замечания

Что действительно означает «функции-члены-члены» класса. Простое определение, похоже, состоит в том, что функция-член `const` не может изменить объект. Но то, что «не может изменить», на самом деле означает здесь. Это просто означает, что вы не можете выполнять назначение для членов данных класса.

Однако вы можете выполнять другие косвенные операции, такие как вставка записи в карту, как показано в примере. Разрешить это может выглядеть так, что эта функция `const` изменяет объект (да, это в некотором смысле), но это разрешено.

Итак, реальное значение заключается в том, что функция-член `const` не может выполнять назначение для переменных данных класса. Но он может делать другие вещи, как описано в этом примере.

Examples

постоянная функция-член

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

class A {
public:
    map<string, string> * mapOfStrings;
public:
    A() {
        mapOfStrings = new map<string, string>();
    }

    void insertEntry(string const & key, string const & value) const {
        (*mapOfStrings)[key] = value;           // This works? Yes it does.
        delete mapOfStrings;                     // This also works
        mapOfStrings = new map<string, string>(); // This * does * not work
    }

    void refresh() {
        delete mapOfStrings;
        mapOfStrings = new map<string, string>(); // Works as refresh is non const function
    }

    void getEntry(string const & key) const {
```

```
        cout << mapOfStrings->at(key);  
    }  
};  
  
int main(int argc, char* argv[]) {  
  
    A var;  
    var.insertEntry("abc", "abcValue");  
    var.getEntry("abc");  
    getchar();  
    return 0;  
}
```

Прочитайте Константные функции члена класса онлайн:

<https://riptutorial.com/ru/cplusplus/topic/7120/константные-функции-члена-класса>

глава 50: Контейнеры C ++

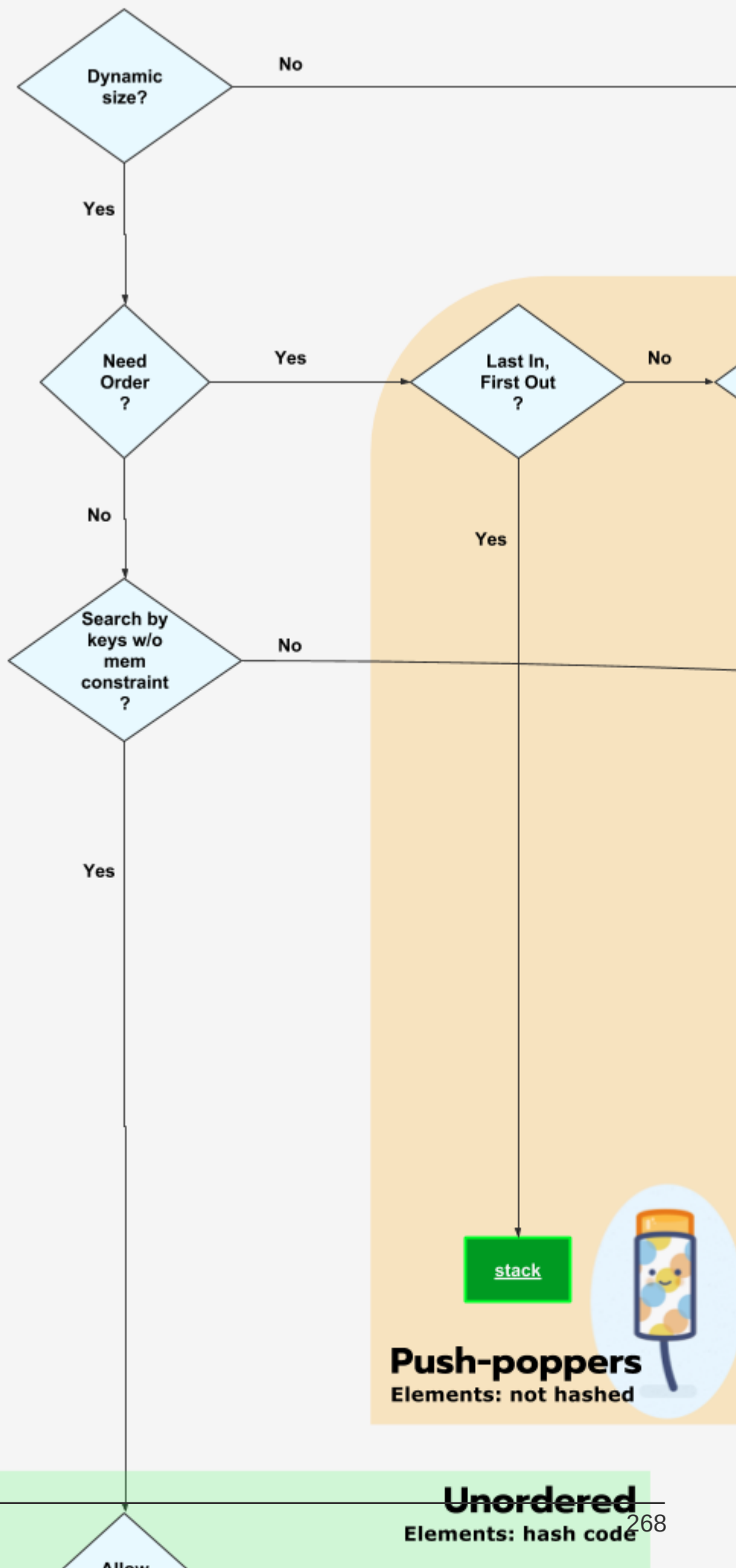
Вступление

Контейнеры C ++ хранят коллекцию элементов. Контейнеры включают в себя векторы, списки, карты и т. Д. Используя шаблоны, контейнеры C ++ содержат коллекции примитивов (например, `int`) или пользовательские классы (например, `MyClass`).

Examples

Блок-схема контейнеров C ++

Выбор того, какой C ++ Container использовать, может быть сложным, так что вот простая блок-схема, которая поможет решить, какой контейнер подходит для работы.



Эта небольшая графика в блок-схеме от [Меган Хопкинс](#)

Прочитайте Контейнеры C ++ онлайн: <https://riptutorial.com/ru/cplusplus/topic/10848/контейнеры-c-plusplus>

глава 51: Копирование Elision

Examples

Цель копирования

В стандарте есть места, где объект копируется или перемещается для инициализации объекта. Копирование elision (иногда называемое оптимизацией возвращаемого значения) - это оптимизация, при которой при определенных обстоятельствах компилятору разрешается избегать копирования или перемещения, даже если стандарт говорит, что это должно произойти.

Рассмотрим следующую функцию:

```
std::string get_string()
{
    return std::string("I am a string.");
}
```

Согласно строгой формулировке стандарта, эта функция инициализирует временную `std::string`, затем скопирует / переместит ее в объект возвращаемого значения, а затем уничтожит временную. В стандарте очень ясно, что так интерпретируется код.

Копирование elision - это правило, которое позволяет компилятору C++ *игнорировать* создание временного и последующего копирования / уничтожения. То есть, компилятор может взять инициализирующее выражение для временного и инициализировать возвращаемое значение функции из него напрямую. Это, очевидно, экономит производительность.

Однако он имеет два видимых эффекта для пользователя:

1. Тип должен иметь конструктор `copy` / `move`, который был бы вызван. Даже если компилятор возвращает копию / перемещение, тип все равно должен быть скопирован / перемещен.
2. Побочные эффекты конструкторов копирования / перемещения не гарантируются в обстоятельствах, когда может возникнуть конфликт. Рассмотрим следующее:

C++ 11

```
struct my_type
{
    my_type() = default;
    my_type(const my_type &) {std::cout <<"Copying\n";}
    my_type(my_type &&) {std::cout <<"Moving\n";}
};
```

```
my_type func()
{
    return my_type();
}
```

Что вызовет `func` ? Ну, он никогда не будет печатать «Копирование», так как временный - это `rvalue`, а `my_type` - подвижный. Так будет ли он печатать «Перемещение»?

Без правила `elise` для копирования это всегда должно было печатать «Moving». Но поскольку существует правило элиминации копирования, конструктор перемещения может быть вызван или не может быть вызван; это зависит от реализации.

И поэтому вы не можете зависеть от вызова конструкторов `copy` / `move` в контекстах, где возможна элиция.

Поскольку `elision` - это оптимизация, ваш компилятор может не поддерживать `elision` во всех случаях. И независимо от того, будет ли компилятор выдавать конкретный случай или нет, тип все равно должен поддерживать отмененную операцию. Поэтому, если построена копия, тип все равно должен иметь конструктор копирования, даже если он не будет вызываться.

Гарантированное копирование

C ++ 17

Обычно элиция - это оптимизация. Несмотря на то, что практически каждая копия поддержки поддерживает копирование в простейшем случае, наличие элиции по-прежнему создает особую нагрузку на пользователей. А именно, тип, который копирует / перемещается, *должен* все же иметь операцию копирования / перемещения, которая была отменена.

Например:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
    return std::lock_guard<std::mutex>(a_mutex);
}
```

Это может быть полезно в случаях, когда `a_mutex` - это мьютекс, который находится в частной собственности какой-либо системы, но внешний пользователь может захотеть заблокировать его.

Это также не является законным, потому что `std::lock_guard` не может быть скопирован или перемещен. Даже несмотря на то, что практически каждый компилятор C ++ будет удалять копию / перемещение, стандарт по-прежнему *требует*, чтобы тип имел эту операцию.

До C ++ 17.

C ++ 17 разрешает элицию путем эффективного переопределения самого значения определенных выражений, чтобы не происходило копирование / перемещение. Рассмотрим приведенный выше код.

В соответствии с формулировкой pre-C ++ 17 этот код говорит о создании временного и затем использует временное копирование / перемещение в возвращаемое значение, но временную копию можно удалить. В соответствии с формулировкой C ++ 17, которая не создает временного вообще.

В C ++ 17 любое **выражение prvalue** , когда используется для инициализации объекта того же типа, что и выражение, не создает временного. Выражение непосредственно инициализирует этот объект. Если вы вернете prvalue того же типа, что и возвращаемое значение, тогда тип не должен иметь конструктор copy / move. И поэтому в соответствии с правилами C ++ 17 приведенный выше код может работать.

Формулировка C ++ 17 работает в тех случаях, когда тип prvalue соответствует инициализированному типу. Поэтому, учитывая `get_lock` выше, это также не потребует копирования / перемещения:

```
std::lock_guard the_lock = get_lock();
```

Поскольку результатом `get_lock` является выражение prvalue, используемое для инициализации объекта того же типа, копирование или перемещение не произойдет. Это выражение никогда не создает временного; он используется для непосредственной инициализации `the_lock` . Элис не существует, потому что копия / перемещение не будет устранено.

Поэтому термин «гарантированное копирование» является чем-то неправильным, но **это название функции, как это предлагается для стандартизации C ++ 17** . Это не гарантирует никакого решения; он полностью *исключает* копирование / перемещение, переопределяя C ++, чтобы никогда не удалялось копирование / перемещение.

Эта функция работает только в случаях, связанных с выражением prvalue. Таким образом, это использует обычные правила:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
    std::lock_guard<std::mutex> my_lock(a_mutex);
    //Do stuff
    return my_lock;
}
```

Несмотря на то, что это правильный пример для копирования, правила C ++ 17 не *исключают* копирование / перемещение в этом случае. Таким образом, тип должен иметь

конструктор копирования / перемещения, который будет использоваться для инициализации возвращаемого значения. И поскольку `lock_guard` этого не делает, это все еще ошибка компиляции. Реализациям разрешено отказывать в выдаче копий при передаче или возврате объекта с возможностью тривиально-копируемого типа. Это позволяет перемещать такие объекты в регистры, которые некоторые ABI могут выполнять в своих соглашениях о вызовах.

```
struct trivially_copyable {
    int a;
};

void foo (trivially_copyable a) {}

foo(trivially_copyable{}); //copy elision not mandated
```

Возвращаемое значение elision

Если вы возвращаете **выражение prvalue** из функции, а выражение prvalue имеет тот же тип, что и возвращаемый тип функции, тогда копия из временного значения prvalue может быть удалена:

```
std::string func()
{
    return std::string("foo");
}
```

Практически все компиляторы в этом случае будут исключать временную конструкцию.

Параметр elision

Когда вы передаете аргумент функции, а аргумент является **выражением prvalue** типа параметра функции, и этот тип не является ссылкой, то конструкцию prvalue можно удалить.

```
void func(std::string str) { ... }

func(std::string("foo"));
```

Это говорит о создании временной `string`, а затем переместить ее в параметр функции `str`. Копирование elision позволяет этому выражению напрямую создавать объект в `str`, а не использовать временный + перемещение.

Это полезная оптимизация для случаев, когда конструктор объявляется `explicit`. Например, мы могли бы написать выше как `func("foo")`, но только потому, что `string` имеет неявный конструктор, который преобразует из `const char*` в `string`. Если этот конструктор был `explicit`, мы были бы вынуждены использовать временный вызов `explicit` конструктора. Копирование elision избавляет нас от необходимости делать ненужную

копию / перемещение.

Именованное возвращаемое значение elision

Если вы возвращаете **выражение lvalue** из функции, и это lvalue:

- представляет собой автоматическую переменную, локальную для этой функции, которая будет уничтожена после `return`
- автоматическая переменная не является параметром функции
- и тип переменной такой же, как тип возвращаемого значения функции

Если все это так, то копирование / перемещение из lvalue можно отбросить:

```
std::string func()
{
    std::string str("foo");
    //Do stuff
    return str;
}
```

Более сложные случаи имеют право на элицию, но чем более сложный случай, тем меньше вероятность того, что компилятор действительно сможет его ликвидировать:

```
std::string func()
{
    std::string ret("foo");
    if(some_condition)
    {
        return "bar";
    }
    return ret;
}
```

Компилятор все еще может игнорировать `ret`, но шансы на их делают так идти вниз.

Как отмечалось ранее, для значений *параметров* не разрешено исключение.

```
std::string func(std::string str)
{
    str.assign("foo");
    //Do stuff
    return str; //No elision possible
}
```

Копирование инициализации

Если вы используете **выражение prvalue** для копирования инициализации переменной, и эта переменная имеет тот же тип, что и выражение prvalue, тогда копирование может быть отменено.

```
std::string str = std::string("foo");
```

Инициализация кода эффективно преобразует это в `std::string str("foo");` (есть незначительные различия).

Это также работает с возвращаемыми значениями:

```
std::string func()
{
    return std::string("foo");
}

std::string str = func();
```

Без копирования elision это вызовет 2 обращения к конструктору перемещения `std::string`. Копирование elision позволяет это вызвать конструктор перемещения 1 или нулевое время, и большинство компиляторов выберет последнее.

Прочитайте Копирование Elision онлайн: <https://riptutorial.com/ru/cplusplus/topic/2489/копирование-elision>

глава 52: Копирование и назначение

Синтаксис

- **Копировать конструктор**
- `MyClass (const MyClass & other);`
- `MyClass (MyClass & other);`
- `MyClass (volatile const MyClass & other);`
- `MyClass (volatile MyClass & other);`
- **Конструктор присваивания**
- `MyClass & operator = (const MyClass & rhs);`
- `MyClass & operator = (MyClass & rhs);`
- `MyClass & operator = (MyClass rhs);`
- `const MyClass & operator = (const MyClass & rhs);`
- `const MyClass & operator = (MyClass & rhs);`
- `const MyClass & operator = (MyClass rhs);`
- `Оператор MyClass = (const MyClass & rhs);`
- `Оператор MyClass = (MyClass & rhs);`
- `Оператор MyClass = (MyClass rhs);`

параметры

шк	Правая сторона равенства для конструкторов копирования и присваивания. Например, конструктор присваивания: <code>MyClass operator = (MyClass & rhs);</code>
Заполнитель	Заполнитель

замечания

Другие полезные ресурсы для дальнейших исследований:

[В чем разница между оператором присваивания и конструктором копирования?](#)

[оператор присваивания и конструктор копирования C ++](#)

[GeeksForGeeks](#)

[Статьи C ++](#)

Examples

Оператор присваивания

Оператор присваивания - это когда вы заменяете данные уже существующим (ранее инициализированным) объектом данными другого объекта. Давайте рассмотрим это как пример:

```
// Assignment Operator
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo& operator=(const Foo& rhs)
    {
        data = rhs.data;
        return *this;
    }

    int data;
};

int main()
{
    Foo foo(2); //Foo(int data) called
    Foo foo2(42);
    foo = foo2; // Assignment Operator Called
    cout << foo.data << endl; //Prints 42
}
```

Здесь вы можете увидеть оператор присваивания, когда я уже инициализировал объект `foo` . Затем позже я назначаю `foo2` в `foo` . Все изменения, появляющиеся при вызове этого оператора равенства, определяются в вашей функции `operator=` . Вы можете увидеть текущую версию здесь: <http://cpp.sh/3qtbm>

Копировать конструктор

С другой стороны, конструктор копирования - это полная противоположность конструктора присваивания. На этот раз он используется для инициализации уже несуществующего (или не инициализированного) объекта. Это означает, что он копирует все данные из объекта, которому вы его назначаете, без фактической инициализации объекта, на который выполняется копирование. Теперь давайте рассмотрим тот же код, что и раньше, но изменим конструктор присваивания как конструктор копирования:

```
// Copy Constructor
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
        data = rhs.data;
    }

    int data;
};

int main()
{
    Foo foo(2); //Foo(int data) called
    Foo foo2 = foo; // Copy Constructor called
    cout << foo2.data << endl;
}
```

Вы можете увидеть здесь `Foo foo2 = foo;` в основной функции я сразу назначаю объект перед его инициализацией, которая, как сказано ранее, означает, что это конструктор копирования. И обратите внимание, что мне не нужно было передавать параметр `int` для объекта `foo2` так как я автоматически вытащил предыдущие данные из объекта `foo`. Вот пример вывода: <http://cpp.sh/5iu7>

Копировать конструктор Vs Assignment Constructor

Хорошо, мы кратко рассмотрели, что конструктор копирования и конструктор присваивания выше, и приводили примеры каждого из них, теперь давайте посмотрим их обоих в одном и том же коде. Этот код будет аналогичен предыдущему. Давайте возьмем это:

```
// Copy vs Assignment Constructor
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
public:
    Foo(int data)
    {
        this->data = data;
    }
}
```

```

~Foo(){};
Foo(const Foo& rhs)
{
    data = rhs.data;
}

Foo& operator=(const Foo& rhs)
{
    data = rhs.data;
    return *this;
}

int data;
};

int main()
{
    Foo foo(2); //Foo(int data) / Normal Constructor called
    Foo foo2 = foo; //Copy Constructor Called
    cout << foo2.data << endl;

    Foo foo3(42);
    foo3=foo; //Assignment Constructor Called
    cout << foo3.data << endl;
}

```

Выход:

```

2
2

```

Здесь вы можете сначала вызвать конструктор копирования, выполнив строку `Foo foo2 = foo;`, Поскольку мы не инициализировали его ранее. Затем мы вызываем оператор присваивания на `foo3`, поскольку он уже инициализирован `foo3=foo;`;

Прочитайте Копирование и назначение онлайн: <https://riptutorial.com/ru/cplusplus/topic/7158/копирование-и-назначение>

глава 53: литералы

Вступление

Традиционно литерал является выражением, обозначающим константу, тип и значение которой очевидны из ее орфографии. Например, `42` является литералом, в то время как `x` нет, так как нужно, чтобы его объявление знало его тип и читало предыдущие строки кода, чтобы узнать его значение.

Тем не менее, C++ 11 также добавил **пользовательские литералы**, которые не являются литералами в традиционном смысле, но могут использоваться в качестве сокращений для вызовов функций.

Examples

правда

Ключевое слово, обозначающее одно из двух возможных значений типа `bool`.

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

ЛОЖНЫЙ

Ключевое слово, обозначающее одно из двух возможных значений типа `bool`.

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

nullptr

C++ 11

Ключевое слово, обозначающее константу нулевого указателя. Он может быть преобразован в любой тип указателя или указателя-члена, что дает нулевой указатель результирующего типа.

```
Widget* p = new Widget();
delete p;
```

```
p = nullptr; // set the pointer to null after deletion
```

Обратите внимание, что `nullptr` сам по себе не является указателем. Тип `nullptr` - это фундаментальный тип, известный как `std::nullptr_t`.

```
void f(int* p);

template <class T>
void g(T* p);

void h(std::nullptr_t p);

int main() {
    f(nullptr); // ok
    g(nullptr); // error
    h(nullptr); // ok
}
```

ЭТОТ

Внутри функции-члена класса **ключевое слово** `this` является указателем на экземпляр класса, на котором была вызвана функция. `this` не может использоваться в статической функции-члене.

```
struct S {
    int x;
    S& operator=(const S& other) {
        x = other.x;
        // return a reference to the object being assigned to
        return *this;
    }
};
```

Тип `this` зависит от cv-квалификации функции-члена: если `x::f` является `const`, то тип `this` внутри `f` является `const X*`, поэтому `this` не может использоваться для изменения нестатических элементов данных изнутри `const`. Аналогично, `this` наследует `volatile` квалификацию от функции, в которой она появляется.

C++ 11

`this` также можно использовать в элементе для *выравнивания* или *выравнивания* для нестатического элемента данных.

```
struct S;
struct T {
    T(const S* s);
    // ...
};
struct S {
    // ...
    T t{this};
};
```

this значение r, поэтому его нельзя назначить.

Целочисленный литерал

Целочисленный литерал является основным выражением формы

- десятичный литерал

Это отличная от нуля цифра (1, 2, 3, 4, 5, 6, 7, 8, 9), за которой следует ноль или более десятичных цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

```
int d = 42;
```

- восьмеричный литерал

Это цифра ноль (0), за которой следует ноль или более восьмеричные цифры (0, 1, 2, 3, 4, 5, 6, 7)

```
int o = 052
```

- шестигранный буквальный

Это последовательность символов 0x или последовательность символов 0X, за которой следует одна или несколько шестнадцатеричных цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F)

```
int x = 0x2a; int X = 0X2A;
```

- бинарный-литерал (с C ++ 14)

Это последовательность символов 0b или последовательность символов 0B, за которой следует одна или несколько двоичных цифр (0, 1)

```
int b = 0b101010; // C++14
```

Целочисленный суффикс, если он предоставлен, может содержать одно или оба из следующих (если оба они предусмотрены, они могут отображаться в любом порядке:

- unsigned-suffix (символ u или символ U)

```
unsigned int u_1 = 42u;
```

- long-suffix (символ l или символ L) или длинный-длинный суффикс (последовательность символов ll или последовательность символов LL) (поскольку C ++ 11)

Следующие переменные также инициализируются с тем же значением:

```
unsigned long long l1 = 18446744073709550592ull; // C++11
unsigned long long l2 = 18'446'744'073'709'550'592llu; // C++14
```

```
unsigned long long l3 = 1844'6744'0737'0955'0592uLL; // C++14
unsigned long long l4 = 184467'440737'0'95505'92LLU; // C++14
```

Заметки

Буквы в целочисленных литералах не чувствительны к регистру: 0xDeAdBaBeU и 0XdeadBABEu представляют одинаковое число (одним исключением является длинный-длинный суффикс, который является либо ll, либо LL, ни lL, ни Ll)

Нет отрицательных целых литералов. Выражения, такие как -1, применяют унарный оператор минус к значению, представленному литералом, что может включать неявные преобразования типов.

В C до C99 (но не в C++) недопустимые десятичные значения, не соответствующие длинному int, имеют тип unsigned long int.

При использовании в контрольном выражении #if или #elif все подписанные целочисленные константы действуют так, как будто они имеют тип std::intmax_t, и все неподписанные целочисленные константы действуют так, как будто они имеют тип std::uintmax_t.

Прочитайте литералы онлайн: <https://riptutorial.com/ru/cplusplus/topic/7836/литералы>

глава 54: Любопытно повторяющийся шаблон шаблона (CRTP)

Вступление

Шаблон, в котором класс наследует шаблон шаблона сам по себе как один из его параметров шаблона. CRTP обычно используется для обеспечения *статического полиморфизма* в C++.

Examples

Любопытно повторяющийся шаблон шаблона (CRTP)

CRTP - это мощная статическая альтернатива виртуальным функциям и традиционному наследованию, которые могут использоваться для предоставления свойств типов во время компиляции. Он работает, имея шаблон базового класса, который принимает в качестве одного из своих параметров шаблона производный класс. Это позволяет ему юридически выполнить `static_cast` его `this` указателя на производный класс.

Разумеется, это также означает, что класс CRTP *всегда* должен использоваться как базовый класс другого класса. И производный класс должен перейти к базовому классу.

C++ 14

Допустим, у вас есть набор контейнеров, которые поддерживают функции `begin()` и `end()`. Требования к стандартной библиотеке для контейнеров требуют большей функциональности. Мы можем создать базовый класс CRTP, который обеспечивает эту функциональность, основанную исключительно на `begin()` и `end()`:

```
#include <iterator>
template <typename Sub>
class Container {
private:
    // self() yields a reference to the derived type
    Sub& self() { return *static_cast<Sub*>(this); }
    Sub const& self() const { return *static_cast<Sub const*>(this); }

public:
    decltype(auto) front() {
        return *self().begin();
    }

    decltype(auto) back() {
        return *std::prev(self().end());
    }
}
```



```

decltype(auto) size() const {
    return std::distance(self().begin(), self().end());
}

decltype(auto) operator[](std::size_t i) {
    return *std::next(self().begin(), i);
}
};

```

Вышеупомянутый класс предоставляет функции `front()`, `back()`, `size()` и `operator[]` для любого подкласса, который предоставляет `begin()` и `end()`. Примером подкласса является простой динамически распределенный массив:

```

#include <memory>
// A dynamically allocated array
template <typename T>
class DynArray : public Container<DynArray<T>> {
public:
    using Base = Container<DynArray<T>>;

    DynArray(std::size_t size)
        : size_{size},
          data_{std::make_unique<T[]>(size_)}
    { }

    T* begin() { return data_.get(); }
    const T* begin() const { return data_.get(); }
    T* end() { return data_.get() + size_; }
    const T* end() const { return data_.get() + size_; }

private:
    std::size_t size_;
    std::unique_ptr<T[]> data_;
};

```

Пользователи класса `DynArray` могут легко использовать интерфейсы, предоставляемые базовым классом `CRT`, следующим образом:

```

DynArray<int> arr(10);
arr.front() = 2;
arr[2] = 5;
assert(arr.size() == 10);

```

Полезность. Этот шаблон, в частности, избегает вызовов виртуальных функций во время выполнения, которые происходят, чтобы пересечь иерархию наследования и просто полагается на статические приведения:

```

DynArray<int> arr(10);
DynArray<int>::Base & base = arr;
base.begin(); // no virtual calls

```

Единственный статический бросок внутри функции `begin()` в базовом классе `Container<DynArray<int>>` позволяет компилятору радикально оптимизировать код, и при

просмотре виртуальной таблицы не происходит во время выполнения.

Ограничения: поскольку базовый класс шаблонизирован и отличается для двух разных `DynArray S`, невозможно сохранить указатели на их базовые классы в однородном по типу массиве, как обычно можно было бы сделать с обычным наследованием, когда базовый класс не зависит от производного тип:

```
class A {};  
class B: public A{};  
  
A* a = new B;
```

CRTP, чтобы избежать дублирования кода

Пример в шаблоне [посетителей](#) представляет собой убедительный прецедент для CRTP:

```
struct IShape  
{  
    virtual ~IShape() = default;  
  
    virtual void accept(IShapeVisitor&) const = 0;  
};  
  
struct Circle : IShape  
{  
    // ...  
    // Each shape has to implement this method the same way  
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }  
    // ...  
};  
  
struct Square : IShape  
{  
    // ...  
    // Each shape has to implement this method the same way  
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }  
    // ...  
};
```

Каждому дочернему типу `IShape` необходимо реализовать ту же функцию одинаково. Это много лишнего набора текста. Вместо этого мы можем ввести новый тип в иерархию, который делает это для нас:

```
template <class Derived>  
struct IShapeAcceptor : IShape {  
    void accept(IShapeVisitor& visitor) const override {  
        // visit with our exact type  
        visitor.visit(*static_cast<Derived const*>(this));  
    }  
};
```

И теперь каждая форма просто должна унаследовать от акцептора:

```
struct Circle : IShapeAcceptor<Circle>
{
    Circle(const Point& center, double radius) : center(center), radius(radius) {}
    Point center;
    double radius;
};

struct Square : IShapeAcceptor<Square>
{
    Square(const Point& topLeft, double sideLength) : topLeft(topLeft), sideLength(sideLength)
{}
    Point topLeft;
    double sideLength;
};
```

Не требуется дублировать код.

Прочитайте Любопытно повторяющийся шаблон шаблона (CRTP) онлайн:

<https://riptutorial.com/ru/cplusplus/topic/9269/любопытно-повторяющийся-шаблон-шаблона--crtp->

глава 55: Лямбда

Синтаксис

- `[ default-capture , capture-list ] ( аргумент-список )` изменяемые атрибуты *throw-specification* $\rightarrow$ *return-type* `{ lambda-body }` // Порядок спецификаций и атрибутов лямбда.
- `[ capture-list ] ( список-аргумент ) { lambda-body }` // Общее определение лямбда.
- `[ = ] ( argument-list ) { lambda-body }` // Захватывает все необходимые локальные переменные по значению.
- `[ & ] ( argument-list ) { lambda-body }` // Захватывает все необходимые локальные переменные по ссылке.
- `[ capture-list ] { lambda-body }` // Список аргументов и спецификаторы могут быть опущены.

параметры

параметр	подробности
по умолчанию захвата	Указывает, как фиксируются все перечисленные переменные. Может быть = (захват по значению) или & (захват по ссылке). Если они опущены, переменные, не включенные в список, недоступны в лямбда-теле . Захват по умолчанию должен предшествовать списку захвата .
Захват-лист	Указывает, как локальные переменные становятся доступными в лямбда-теле . Переменные без префикса записываются по значению. Переменные с префиксом & записываются по ссылке. В методе класса <code>this</code> можно использовать, чтобы сделать все его члены доступными по ссылке. Перечисленные переменные недоступны, если перед списком не предшествует захват по умолчанию .
аргумент-список	Задаёт аргументы лямбда-функции.
изменчивый	(необязательно) Обычно переменные, захваченные значением, являются <code>const</code> . Указание <code>mutable</code> делает их неконстантными. Изменения этих переменных сохраняются между вызовами.
вбрасывание спецификация	(необязательно) Определяет поведение бросания исключения из лямбда-функции. Например: <code>noexcept</code> или <code>throw (std::exception)</code> .

параметр	подробности
атрибуты	(необязательно) Любые атрибуты для лямбда-функции. Например, если лямбда-тело всегда выдает исключение, то можно использовать <code>[[noreturn]]</code> .
-> return-type	(необязательно) Указывает тип возврата лямбда-функции. Требуется, когда тип возврата не может быть определен компилятором.
лямбда-тело	Блок кода, содержащий реализацию лямбда-функции.

замечания

С ++ 17 (текущий проект) вводит `constexpr`, в основном `constexpr`, который можно оценить во время компиляции. Лямбда автоматически `constexpr` если она удовлетворяет требованиям `constexpr`, но вы также можете указать ее с помощью `constexpr` слова `constexpr`:

```
//Explicitly define this lambdas as constexpr
[]() constexpr {
    //Do stuff
}
```

Examples

Что такое лямбда-выражение?

Выражение лямбда обеспечивает краткий способ создания простых объектов функции. Выражение лямбда является `rvalue`, результат которого называется **замыкающим объектом**, который ведет себя как объект функции.

Название «лямбда-выражение» происходит от **лямбда-исчисления**, которое является математическим формализмом, изобретенным в 1930-х годах церковью Алонсо для изучения вопросов о логике и вычислимости. Лямбда-исчисление легло в основу **LISP**, функционального языка программирования. По сравнению с лямбда-исчислением и LISP, лямбда-выражения C ++ разделяют свойства неназванного и захватывают переменные из окружающего контекста, но им не хватает возможности работать и возвращать функции.

Выражение лямбда часто используется в качестве аргумента для функций, которые принимают вызываемый объект. Это может быть проще, чем создание именованной функции, которая будет использоваться только при передаче в качестве аргумента. В таких случаях лямбда-выражения обычно предпочтительнее, поскольку они позволяют определять функциональные объекты inline.

Лямбда состоит из трех частей: списка захвата `[]` , необязательного списка параметров `()` и тела `{}` , все из которых могут быть пустыми:

```
[](){} // An empty lambda, which does and returns nothing
```

Список захвата

`[]` - **список захвата** . По умолчанию переменные охватывающей области недоступны лямбдой. *Захват* переменной делает ее доступной внутри лямбда, либо [в виде копии](#), либо [в качестве ссылки](#) . Захваченные переменные становятся частью лямбда; в отличие от аргументов функции, они не должны передаваться при вызове лямбда.

```
int a = 0; // Define an integer variable
auto f = []() { return a*9; }; // Error: 'a' cannot be accessed
auto f = [a]() { return a*9; }; // OK, 'a' is "captured" by value
auto f = [&a]() { return a++; }; // OK, 'a' is "captured" by reference
// Note: It is the responsibility of the programmer
// to ensure that a is not destroyed before the
// lambda is called.
auto b = f(); // Call the lambda function. a is taken from the capture list
and not passed here.
```

Список параметров

`()` - **список параметров** , который почти такой же, как и в обычных функциях. Если лямбда не принимает аргументов, эти круглые скобки могут быть опущены (за исключением того, что вам нужно объявить `lambda mutable`). Эти два лямбда эквивалентны:

```
auto call_foo = [x]() { x.foo(); };
auto call_foo2 = [x]{ x.foo(); };
```

C ++ 14

Список параметров может использовать `auto` тип placeholder вместо фактических типов. Таким образом, этот аргумент ведет себя как шаблонный шаблон шаблона функции. Следующие `lambdas` эквивалентны, когда вы хотите отсортировать вектор в общем коде:

```
auto sort_cpp11 = [](std::vector<T>::const_reference lhs, std::vector<T>::const_reference rhs)
{ return lhs < rhs; };
auto sort_cpp14 = [](const auto &lhs, const auto &rhs) { return lhs < rhs; };
```

Тело функции

`{}` - это **тело** , такое же, как и в обычных функциях.

Вызов лямбда

Объектом результата лямбда-выражения является [замыкание](#) , которое можно вызвать с

помощью `operator()` (как и с другими объектами функций):

```
int multiplier = 5;
auto timesFive = [multiplier](int a) { return a * multiplier; };
std::cout << timesFive(2); // Prints 10

multiplier = 15;
std::cout << timesFive(2); // Still prints 2*5 == 10
```

Тип возврата

По умолчанию выводится возвращаемый тип выражения лямбда.

```
[](){ return true; };
```

В этом случае тип возврата - `bool`.

Вы также можете вручную указать тип возврата, используя следующий синтаксис:

```
[]() -> bool { return true; };
```

Mutable Lambda

Объекты, захваченные значением в лямбда, по умолчанию неизменяемы. Это связано с тем, что `operator()` созданного объекта замыкания по умолчанию является `const`.

```
auto func = [c = 0]() { ++c; std::cout << c; }; // fails to compile because ++c
                                                // tries to mutate the state of
                                                // the lambda.
```

Модификация может быть разрешена с помощью ключевого слова `mutable`, которые делают `operator()` не `const`:

```
auto func = [c = 0]() mutable { ++c; std::cout << c; };
```

Если используется вместе с возвращаемым типом, перед ним появляется `mutable`.

```
auto func = [c = 0]() mutable -> int { ++c; std::cout << c; return c; };
```

Пример, иллюстрирующий полезность лямбда

Перед C++ 11:

C++ 11

```
// Generic functor used for comparison
struct islessthan
{
```

```

    islessthan(int threshold) : _threshold(threshold) {}

    bool operator()(int value) const
    {
        return value < _threshold;
    }
private:
    int _threshold;
};

// Declare a vector
const int arr[] = { 1, 2, 3, 4, 5 };
std::vector<int> vec(arr, arr+5);

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
std::vector<int>::iterator it = std::find_if(vec.begin(), vec.end(), islessthan(threshold));

```

Поскольку C ++ 11:

C ++ 11

```

// Declare a vector
std::vector<int> vec{ 1, 2, 3, 4, 5 };

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
auto it = std::find_if(vec.begin(), vec.end(), [threshold](int value) { return value <
threshold; });

```

Указание типа возврата

Для lambdas с одним оператором return или несколькими операторами возврата, выражения которых имеют один и тот же тип, компилятор может выводиться возвращаемый тип:

```

// Returns bool, because "value > 10" is a comparison which yields a Boolean result
auto l = [](int value) {
    return value > 10;
}

```

Для lambdas с несколькими операторами возврата *разных* типов компилятор не может выводиться возвращаемый тип:

```

// error: return types must match if lambda has unspecified return type
auto l = [](int value) {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};

```

В этом случае вы должны явно указать тип возвращаемого значения:


```
// The return type is specified explicitly as 'double'
auto l = [](int value) -> double {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

Правила для этого соответствуют правилам `auto` вычета типов. Lambdas без явно указанных типов возврата никогда не возвращает ссылки, поэтому, если желателен ссылочный тип, он также должен быть явно указан:

```
auto copy = [](X& x) { return x; }; // 'copy' returns an X, so copies its input
auto ref = [](X& x) -> X& { return x; }; // 'ref' returns an X&, no copy
```

Захват по значению

Если вы укажете имя переменной в списке захвата, лямбда зафиксирует ее по значению. Это означает, что сгенерированный тип замыкания для лямбда хранит копию переменной. Это также требует, чтобы тип переменной был выполнен с возможностью *копирования* :

```
int a = 0;

[a]() {
    return a; // Ok, 'a' is captured by value
};
```

C ++ 14

```
auto p = std::unique_ptr<T>(...);

[p]() { // Compile error; `unique_ptr` is not copy-constructible
    return p->createWidget();
};
```

Начиная с C ++ 14, можно инициализировать переменные на месте. Это позволяет перемещать только типы, которые должны быть записаны в лямбда.

C ++ 14

```
auto p = std::make_unique<T>(...);

[p = std::move(p)]() {
    return p->createWidget();
};
```

Несмотря на то, что лямбда фиксирует переменные по значению, когда они даются по их имени, такие переменные не могут быть изменены внутри тела лямбда по умолчанию. Это связано с тем, что тип замыкания ставит лямбда-тело в объявление `operator() const` .

`const` относится к доступу к переменным - членам типа закрытия, и захватили переменные, которые являются членами закрытия (все выступления до наоборот):

```
int a = 0;

[a]() {
    a = 2;          // Illegal, 'a' is accessed via `const`

    decltype(a) a1 = 1;
    a1 = 2; // valid: variable 'a1' is not const
};
```

Чтобы удалить `const`, вы должны указать ключевое слово `mutable` на лямбде:

```
int a = 0;

[a]() mutable {
    a = 2;          // OK, 'a' can be modified
    return a;
};
```

Поскольку `a` был захвачен по значению, любые изменения, сделанные при вызове лямбда, не повлияют на `a`. Значение было скопировано в лямбда, когда он был построен, поэтому копия лямбды отделена от внешнего `a` переменной.

```
int a = 5 ;
auto plus5Val = [a] (void) { return a + 5 ; } ;
auto plus5Ref = [&a] (void) {return a + 5 ; } ;

a = 7 ;
std::cout << a << ", value " << plus5Val() << ", reference " << plus5Ref() ;
// The result will be "7, value 10, reference 12"
```

Обобщенный захват

C++ 14

Lambdas может захватывать выражения, а не просто переменные. Это позволяет lambdas хранить типы только для перемещения:

```
auto p = std::make_unique<T>(...);

auto lamb = [p = std::move(p)]() //Overrides capture-by-value of `p`.
{
    p->SomeFunc();
};
```

Это перемещает внешнюю переменную `p` переменную захвата лямбда, также называемую `p`. `lamb` теперь владеет памятью, выделенную `make_unique`. Поскольку закрытие содержит тип, который не копируется, это означает, что сам `lamb` сам не копируется. Но его можно переместить:

```
auto lamb_copy = lamb; //Illegal
auto lamb_move = std::move(lamb); //legal.
```

Теперь `lamb_move` владеет памятью.

Обратите внимание, что `std::function<>` требует, чтобы сохраненные значения были скопируемыми. Вы можете написать свою собственную `std::function` **only, требующую перемещения только**, или вы можете просто набить лямбду в обертку `shared_ptr`:

```
auto shared_lambda = [](auto&& f){
    return [spf = std::make_shared<std::decay_t<decltype(f)>>(decltype(f)(f))](
        auto&&...args)->decltype(auto) {
        return (*spf)(decltype(args)(args)...);
    };
};
auto lamb_shared = shared_lambda(std::move(lamb_move));
```

берет нашу лямбду только для движения и наполняет ее состояние общим указателем, а затем возвращает лямбду, которую *можно* скопировать, а затем сохранить в `std::function` или аналогичном.

Обобщенный захват использует `auto` вывод типа для типа переменной. Он объявит эти захваты как значения по умолчанию, но они также могут быть ссылками:

```
int a = 0;

auto lamb = [&v = a](int add) //Note that `a` and `v` have different names
{
    v += add; //Modifies `a`
};

lamb(20); //`a` becomes 20.
```

Обобщение захвата вообще не требует захвата внешней переменной. Он может фиксировать произвольное выражение:

```
auto lamb = [p = std::make_unique<T>(...)]()
{
    p->SomeFunc();
}
```

Это полезно для предоставления произвольных значений `lambdas`, которые они могут удерживать и потенциально изменять, без необходимости объявлять их извне в лямбда. Конечно, это полезно только в том случае, если вы не намереваетесь получить доступ к этим переменным после того, как лямбда завершила свою работу.

Захват по ссылке

Если вы предшествуете имени локальной переменной с помощью `&` , тогда переменная будет записана по ссылке. Концептуально это означает, что тип закрытия лямбда будет иметь ссылочную переменную, инициализированную как ссылку на соответствующую переменную из-за пределов лямбда. Любое использование переменной в лямбда-теле будет относиться к исходной переменной:

```
// Declare variable 'a'
int a = 0;

// Declare a lambda which captures 'a' by reference
auto set = [&a]() {
    a = 1;
};

set();
assert(a == 1);
```

Ключевое слово `mutable` не требуется, потому `a` сам не является `const` .

Конечно, захват по ссылке означает, что лямбда **не должна** выходить за пределы диапазона переменных, которые она захватывает. Таким образом, вы можете вызывать функции, которые выполняют функцию, но вы не должны вызывать функцию, которая будет *хранить* лямбда вне сферы ваших ссылок. И вы не должны возвращать лямбду.

Удержание по умолчанию

По умолчанию локальные переменные, которые явно не указаны в списке захвата, не могут быть доступны изнутри тела лямбда. Тем не менее, можно неявно захватывать переменные, названные телом лямбда:

```
int a = 1;
int b = 2;

// Default capture by value
[=]() { return a + b; }; // OK; a and b are captured by value

// Default capture by reference
[&]() { return a + b; }; // OK; a and b are captured by reference
```

Явный захват по-прежнему может выполняться наряду с неявным захватом по умолчанию. Явное определение захвата переопределит захват по умолчанию:

```
int a = 0;
int b = 1;

[=, &b]() {
    a = 2; // Illegal; 'a' is capture by value, and lambda is not 'mutable'
    b = 2; // OK; 'b' is captured by reference
};
```

Родовые лямбды

C++ 14

Лямбда-функции могут принимать аргументы произвольных типов. Это позволяет лямбда быть более общей:

```
auto twice = [](auto x){ return x+x; };

int i = twice(2); // i == 4
std::string s = twice("hello"); // s == "hellohello"
```

Это реализовано в C++, заставив `operator()` типа закрытия `operator()` перегрузить функцию шаблона. Следующий тип имеет эквивалентное поведение для вышеуказанного лямбда-замыкания:

```
struct _unique_lambda_type
{
    template<typename T>
    auto operator() (T x) const {return x + x;}
};
```

Не все параметры в общей лямбде должны быть общими:

```
[](auto x, int y) {return x + y;}
```

Здесь `x` выводится на основе первого аргумента функции, а `y` всегда будет `int`.

Родовые `lambdas` также могут принимать аргументы по ссылке, используя обычные правила для `auto` и `&`. Если общий параметр принимается как `auto&&`, это [ссылка](#) на [пересылку](#) переданного аргумента, а не [ссылку rvalue](#):

```
auto lamb1 = [](int &&x) {return x + 5;};
auto lamb2 = [](auto &&x) {return x + 5;};
int x = 10;
lamb1(x); // Illegal; must use `std::move(x)` for `int&&` parameters.
lamb2(x); // Legal; the type of `x` is deduced as `int&`.
```

Лямбда-функции могут быть вариативными и идеально продвигать свои аргументы:

```
auto lam = [](auto&&... args){return f(std::forward<decltype(args)>(args)...);};
```

или же:

```
auto lam = [](auto&&... args){return f(decltype(args)(args)...);};
```

который работает только «правильно» с переменными типа `auto&&`.

Сильная причина использовать общие лямбды - это синтаксис посещения.

```
boost::variant<int, double> value;
apply_visitor(value, [&](auto&& e){
    std::cout << e;
});
```

Здесь мы находимся полиморфно; но в других контекстах имена типа, который мы проходим, не интересны:

```
mutex_wrapped<std::ostream&> os = std::cout;
os.write([&](auto&& os){
    os << "hello world\n";
});
```

Повторение типа `std::ostream&` is noise здесь; это будет означать необходимость упоминать тип переменной каждый раз, когда вы ее используете. Здесь мы создаем посетителя, но не полиморфного; `auto` используется по той же причине, что вы можете использовать `auto` в цикле `for(:`) .

Преобразование в указатель функции

Если список захвата лямбда пуст, то лямбда имеет неявное преобразование в указатель функции, который принимает те же аргументы и возвращает тот же тип возврата:

```
auto sorter = [](int lhs, int rhs) -> bool {return lhs < rhs;};

using func_ptr = bool (*)(int, int);
func_ptr sorter_func = sorter; // implicit conversion
```

Такое преобразование также может быть выполнено с использованием оператора унарного пользователя:

```
func_ptr sorter_func2 = +sorter; // enforce implicit conversion
```

Вызов этого указателя функции ведет себя точно так же, как вызов `operator()` на лямбда. Указатель этой функции никоим образом не зависит от существования источника лямбда-замыкания. Поэтому он может пережить лямбда-закрывание.

Эта функция в основном полезна для использования `lambdas` с API, которые имеют дело с указателями функций, а не с объектами функций C ++.

C ++ 14

Преобразование в указатель функции также возможно для общих лямбда с пустым списком захвата. При необходимости вычитание аргумента шаблона будет использоваться для выбора правильной специализации.

```

auto sorter = [](auto lhs, auto rhs) { return lhs < rhs; };
using func_ptr = bool (*)(int, int);
func_ptr sorter_func = sorter; // deduces int, int
// note however that the following is ambiguous
// func_ptr sorter_func2 = +sorter;

```

Класс лямбда и захват этого

Выражение `lambda`, оцениваемое в функции-члене класса, неявно является другом этого класса:

```

class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    // definition of a member function
    void Test()
    {
        auto lamb = [](Foo &foo, int val)
        {
            // modification of a private member variable
            foo.i = val;
        };

        // lamb is allowed to access a private member, because it is a friend of Foo
        lamb(*this, 30);
    }
};

```

Такая лямбда не только друг этого класса, но и тот же доступ, что и класс, объявленный внутри.

`Lambdas` может захватить `this` указатель, который представляет экземпляр объекта, на который была вызвана внешняя функция. Это делается путем добавления `this` в список захвата:

```

class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture the this pointer by value
        auto lamb = [this](int val)
        {
            i = val;
        };
    }
};

```

```

        lamb(30);
    }
};

```

Когда `this` будет зафиксировано, лямбда может использовать имена членов своего класса, как если бы они находились в его содержащем классе. Поэтому для таких членов применяется неявное `this->`.

Имейте в виду, что `this` фиксируется значением, но не значением типа. Он захватывается значением `this`, которое является *указателем*. Таким образом, лямбда не *владеет* `this`. Если лямбда вне жизни объекта, который ее создал, лямбда может стать недействительной.

Это также означает, что лямбда может изменить `this` не будучи объявленным `mutable`. Это указатель, который является `const`, а не объектом, на который указывают. То есть, если внешняя функция-член сама не является функцией `const`.

Кроме того, имейте в виду, что положения захвата по умолчанию, как `[=]` и `[&]`, также будут фиксировать `this` неявно. И они оба фиксируют его по значению указателя. В самом деле, это ошибка, указывающая `this` в списке захвата при задании по умолчанию.

C ++ 17

Lambdas может захватить копию `this` объекта, созданного в момент создания лямбда. Это делается путем добавления `*this` в список захвата:

```

class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture a copy of the object given by the this pointer
        auto lamb = [*this](int val) mutable
        {
            i = val;
        };

        lamb(30); // does not change this->i
    }
};

```

Перенос лямбда-функций на C ++ 03 с использованием функторов

Лямбда-функции в C ++ - это синтаксический сахар, который обеспечивает очень сжатый синтаксис для написания [функторов](#). Таким образом, эквивалентная функциональность

может быть получена в C++03 (хотя и намного более подробно) путем преобразования лямбда-функции в функтор:

```
// Some dummy types:
struct T1 {int dummy;};
struct T2 {int dummy;};
struct R {int dummy;};

// Code using a lambda function (requires C++11)
R use_lambda(T1 val, T2 ref) {
    // Use auto because the type of the lambda is unknown.
    auto lambda = [val, &ref](int arg1, int arg2) -> R {
        /* lambda-body */
        return R();
    };
    return lambda(12, 27);
}

// The functor class (valid C++03)
// Similar to what the compiler generates for the lambda function.
class Functor {
    // Capture list.
    T1 val;
    T2& ref;

public:
    // Constructor
    inline Functor(T1 val, T2& ref) : val(val), ref(ref) {}

    // Functor body
    R operator()(int arg1, int arg2) const {
        /* lambda-body */
        return R();
    }
};

// Equivalent to use_lambda, but uses a functor (valid C++03).
R use_functor(T1 val, T2 ref) {
    Functor functor(val, ref);
    return functor(12, 27);
}

// Make this a self-contained example.
int main() {
    T1 t1;
    T2 t2;
    use_functor(t1,t2);
    use_lambda(t1,t2);
    return 0;
}
```

Если лямбда-функция `mutable` то сделайте оператор-оператор функтора неконстантным, то есть:

```
R operator()(int arg1, int arg2) /*non-const*/ {
    /* lambda-body */
    return R();
}
```

Рекурсивные лямбды

Предположим, мы хотим написать `gcd()` Евклида как лямбда. Как функция, это:

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a%b);
}
```

Но лямбда не может быть рекурсивной, она не имеет возможности сослаться на себя. Лямбда не имеет имени и использование `this` в теле лямбды относится к захваченному `this` (предполагая, что лямбда создается в теле функции члена, в противном случае это ошибка). Итак, как мы решаем эту проблему?

Использовать `std::function`

У нас может быть захват лямбда ссылкой на еще не сконструированную `std::function`

```
std::function<int(int, int)> gcd = [&](int a, int b){
    return b == 0 ? a : gcd(b, a%b);
};
```

Это работает, но следует использовать экономно. Он медленный (теперь мы используем стирание стилей вместо прямого вызова функции), он хрупкий (копирование `gcd` или возвращение `gcd` будет ломаться, поскольку лямбда относится к исходному объекту), и он не будет работать с общими лямбдами.

Используя два интеллектуальных указателя:

```
auto gcd_self = std::make_shared<std::unique_ptr< std::function<int(int, int)> >>()>();
* gcd_self = std::make_unique<std::function<int(int, int)>>()>()
    [gcd_self](int a, int b){
        return b == 0 ? a : (**gcd_self)(b, a%b);
    };
};
```

Это добавляет много косвенности (что является накладными расходами), но может быть скопировано / возвращено, и все копии разделяют состояние. Это позволяет вам вернуть лямбду и в противном случае менее хрупким, чем вышеупомянутое решение.

Используйте Y-комбинатор

С помощью короткой служебной структуры мы можем решить все эти проблемы:

```
template <class F>
struct y_combinator {
    F f; // the lambda will be stored here

    // a forwarding operator():

```

```

template <class... Args>
decltype(auto) operator()(Args&&... args) const {
    // we pass ourselves to f, then the arguments.
    // the lambda should take the first argument as `auto&& recurse` or similar.
    return f(*this, std::forward<Args>(args)...);
}
};
// helper function that deduces the type of the lambda:
template <class F>
y_combinator<std::decay_t<F>> make_y_combinator(F&& f) {
    return {std::forward<F>(f)};
}
// (Be aware that in C++17 we can do better than a `make_` function)

```

мы можем реализовать наш gcd как:

```

auto gcd = make_y_combinator(
    [](auto&& gcd, int a, int b){
        return b == 0 ? a : gcd(b, a%b);
    }
);

```

`y_combinator` - это концепция из лямбда-исчисления, которая позволяет вам иметь рекурсию, не имея возможности назвать себя до тех пор, пока вы не определитесь. Это именно то, что есть у лямбда.

Вы создаете лямбду, которая принимает «recurse» в качестве своего первого аргумента. Когда вы хотите рекурсивно, вы передаете аргументы для рекурсии.

Затем `y_combinator` возвращает объект функции, который вызывает эту функцию со своими аргументами, но с подходящим объектом «`y_combinator`» (а именно сам `y_combinator`) в качестве своего первого аргумента. Он перенаправляет остальные аргументы, которые вы называете `y_combinator` на лямбда.

Короче:

```

auto foo = make_y_combinator( [&](auto&& recurse, some arguments) {
    // write body that processes some arguments
    // when you want to recurse, call recurse(some other arguments)
});

```

и у вас есть рекурсия в лямбда без каких-либо серьезных ограничений или значительных накладных расходов.

Использование lambdas для распаковки пакетов встроенных параметров

C ++ 14

Для распаковки пакета параметров традиционно требуется написать вспомогательную функцию для каждого раза, когда вы хотите это сделать.

В этом примере игрушек:

```
template<std::size_t...Is>
void print_indexes( std::index_sequence<Is...> ) {
    using discard=int[];
    (void)discard{0,((void)(
        std::cout << Is << '\n' // here Is is a compile-time constant.
    ),0)...};
}
template<std::size_t I>
void print_indexes_upto() {
    return print_indexes( std::make_index_sequence<I>{} );
}
```

`print_indexes_upto` хочет создать и распаковать пакет параметров индексов. Для этого он должен вызвать вспомогательную функцию. Каждый раз, когда вы хотите распаковать созданный вами пакет параметров, вам нужно создать для него настраиваемую вспомогательную функцию.

Этого можно избежать с помощью лямбда.

Вы можете распаковать пакеты параметров в набор вызовов лямбда, например:

```
template<std::size_t I>
using index_t = std::integral_constant<std::size_t, I>;
template<std::size_t I>
constexpr index_t<I> index{};

template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
    return [] (auto&& f){
        using discard=int[];
        (void)discard{0, (void(
            f( index<Is> )
        ),0)...};
    };
}

template<std::size_t N>
auto index_over(index_t<N> = {}) {
    return index_over( std::make_index_sequence<N>{} );
}
```

C++ 17

С помощью выражений `fold`, `index_over()` можно упростить до:

```
template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
    return [] (auto&& f){
        ((void)(f(index<Is>)), ...);
    };
}
```

Как только вы это сделаете, вы можете использовать это, чтобы заменить необходимость

вручную распаковывать пакеты параметров со второй перегрузкой в другом коде, позволяя распаковать пакеты параметров «inline»:

```
template<class Tup, class F>
void for_each_tuple_element(Tup&& tup, F&& f) {
    using T = std::remove_reference_t<Tup>;
    using std::tuple_size;
    auto from_zero_to_N = index_over< tuple_size<T>{} >();

    from_zero_to_N(
        [&](auto i){
            using std::get;
            f( get<i>( std::forward<Tup>(tup) ) );
        }
    );
}
```

`auto i` переданный лямбда `index_over` является `std::integral_constant<std::size_t, ???>` `index_over` `std::integral_constant<std::size_t, ???>` . Это преобразование `constexpr` в `std::size_t` которое не зависит от состояния `this` , поэтому мы можем использовать его как константу времени компиляции, например, когда мы передаем его в `std::get<i>` выше.

Чтобы вернуться к примеру игрушки в верхней части, перепишите его как:

```
template<std::size_t I>
void print_indexes_upto() {
    index_over(index<I>) ([&](auto i){
        std::cout << i << '\n'; // here i is a compile-time constant
    });
}
```

который намного короче и сохраняет логику в коде, который ее использует.

[Живой пример](#) игры.

Прочитайте Лямбда онлайн: <https://riptutorial.com/ru/cplusplus/topic/572/лямбда>

глава 56: Макет типов объектов

замечания

См. Также [Размер интегральных типов](#) .

Examples

Типы классов

Под «классом» мы подразумеваем тип, который был определен с использованием ключевого слова `class` или `struct` (но не `enum class` или `enum class`).

- Даже пустой класс по-прежнему занимает как минимум один байт памяти; поэтому он будет состоять из заполнения. Это гарантирует, что если `p` указывает на объект пустого класса, то `p + 1` является отдельным адресом и указывает на отдельный объект. Однако при использовании в качестве базового класса для пустого класса может иметь размер 0. См. [Пустую оптимизацию базы](#) .

```
class Empty_1 {}; // sizeof(Empty_1) == 1
class Empty_2 {}; // sizeof(Empty_2) == 1
class Derived : Empty_1 {}; // sizeof(Derived) == 1
class DoubleDerived : Empty_1, Empty_2 {}; // sizeof(DoubleDerived) == 1
class Holder { Empty_1 e; }; // sizeof(Holder) == 1
class DoubleHolder { Empty_1 e1; Empty_2 e2; }; // sizeof(DoubleHolder) == 2
class DerivedHolder : Empty_1 { Empty_1 e; }; // sizeof(DerivedHolder) == 2
```

- Объектное представление типа класса содержит представления объектов базового класса и нестатических типов членов. Поэтому, например, в следующем классе:

```
struct S {
    int x;
    char* y;
};
```

существует последовательная последовательность `sizeof(int)` байтов внутри объекта `s`, называемая *подобъектом*, который содержит значение `x`, и другой подобъект с байтами `sizeof(char*)` который содержит значение `y`. Эти два не могут чередоваться.

- Если тип класса имеет члены и / или базовые классы с типами `t1`, `t2`, ..., `tN`, размер должен быть не меньше `sizeof(t1) + sizeof(t2) + ... + sizeof(tN)` учетом предыдущих точек, Однако в зависимости от требований к [выравниванию](#) членов и базовых классов компилятору может быть принудительно вставлено дополнение между подобъектами или в начале или конце полного объекта.

```

struct AnInt      { int i; };
// sizeof(AnInt)      == sizeof(int)
// Assuming a typical 32- or 64-bit system, sizeof(AnInt)      == 4 (4).
struct TwoInts    { int i, j; };
// sizeof(TwoInts)    >= 2 * sizeof(int)
// Assuming a typical 32- or 64-bit system, sizeof(TwoInts)    == 8 (4 + 4).
struct IntAndChar { int i; char c; };
// sizeof(IntAndChar) >= sizeof(int) + sizeof(char)
// Assuming a typical 32- or 64-bit system, sizeof(IntAndChar) == 8 (4 + 1 +
padding).
struct AnIntDerived : AnInt { long long l; };
// sizeof(AnIntDerived) >= sizeof(AnInt) + sizeof(long long)
// Assuming a typical 32- or 64-bit system, sizeof(AnIntDerived) == 16 (4 + padding +
8).

```

- Если заполнение вставляется в объект из-за требований к выравниванию, размер будет больше, чем сумма размеров членов и базовых классов. При выравнивании n байт размер обычно будет наименьшим кратным n который больше, чем размер всех членов и базовых классов. Каждый член memN , как правило, помещается по адресу, который является кратным $\text{alignof}(\text{memN})$, и n обычно будет самым большим alignof из всех alignof . Из-за этого, если член с меньшим alignof сопровождается элементом с большим alignof , существует вероятность того, что последний элемент не будет правильно выровнен, если он будет установлен сразу после первого. В этом случае прокладка (также известная как *элемент выравнивания*) будет помещена между двумя элементами, так что последний элемент может иметь желаемое выравнивание. И наоборот, если для элемента с большим alignof следует элемент с меньшим alignof , обычно никаких alignof не требуется. Этот процесс также известен как «упаковка». Из-за того, что классы, обычно разделяющие alignof их члена с наибольшим alignof , классы обычно выравниваются с alignof самого большого встроенного типа, который они прямо или косвенно содержат.

```

// Assume sizeof(short) == 2, sizeof(int) == 4, and sizeof(long long) == 8.
// Assume 4-byte alignment is specified to the compiler.
struct Char { char c; };
// sizeof(Char)      == 1 (sizeof(char))
struct Int { int i; };
// sizeof(Int)      == 4 (sizeof(int))
struct CharInt { char c; int i; };
// sizeof(CharInt)   == 8 (1 (char) + 3 (padding) + 4 (int))
struct ShortIntCharInt { short s; int i; char c; int j; };
// sizeof(ShortIntCharInt) == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
//                               3 (padding) + 4 (int))
struct ShortIntCharCharInt { short s; int i; char c; char d; int j; };
// sizeof(ShortIntCharCharInt) == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
//                               1 (char) + 2 (padding) + 4 (int))
struct ShortCharShortInt { short s; char c; short t; int i; };
// sizeof(ShortCharShortInt) == 12 (2 (short) + 1 (char) + 1 (padding) + 2 (short) +
//                               2 (padding) + 4 (int))
struct IntLLInt { int i; long long l; int j; };
// sizeof(IntLLInt)   == 16 (4 (int) + 8 (long long) + 4 (int))
// If packing isn't explicitly specified, most compilers will pack this as
// 8-byte alignment, such that:
// sizeof(IntLLInt)   == 24 (4 (int) + 4 (padding) + 8 (long long) +

```

```

//                                     4 (int) + 4 (padding))

// Assume sizeof(bool) == 1, sizeof(ShortIntCharInt) == 16, and sizeof(IntLLInt) == 24.
// Assume default alignment: alignof(ShortIntCharInt) == 4, alignof(IntLLInt) == 8.
struct ShortChar3ArrShortInt {
    short s;
    char c3[3];
    short t;
    int i;
};
// ShortChar3ArrShortInt has 4-byte alignment: alignof(int) >= alignof(char) &&
//                                     alignof(int) >= alignof(short)
// sizeof(ShortChar3ArrShortInt) == 12 (2 (short) + 3 (char[3]) + 1 (padding) +
//                                     2 (short) + 4 (int))
// Note that t is placed at alignment of 2, not 4. alignof(short) == 2.

struct Large_1 {
    ShortIntCharInt sici;
    bool b;
    ShortIntCharInt tjdj;
};
// Large_1 has 4-byte alignment.
// alignof(ShortIntCharInt) == alignof(int) == 4
// alignof(b) == 1
// Therefore, alignof(Large_1) == 4.
// sizeof(Large_1) == 36 (16 (ShortIntCharInt) + 1 (bool) + 3 (padding) +
//                                     16 (ShortIntCharInt))
struct Large_2 {
    IntLLInt illi;
    float f;
    IntLLInt jmmj;
};
// Large_2 has 8-byte alignment.
// alignof(IntLLInt) == alignof(long long) == 8
// alignof(float) == 4
// Therefore, alignof(Large_2) == 8.
// sizeof(Large_2) == 56 (24 (IntLLInt) + 4 (float) + 4 (padding) + 24 (IntLLInt))

```

C ++ 11

- Если строгое выравнивание принудительно `alignas` с помощью `alignas`, то `alignas` будет использоваться, чтобы заставить тип соответствовать заданному выравниванию, даже если он в противном случае был бы меньшим. Например, с приведенным ниже определением `Chars<5>` будет иметь три (или, возможно, больше) байта заполнения, вставленные в конец, так что его общий размер равен 8. Невозможно, чтобы класс с выравниванием 4 имел размер из 5, потому что было бы невозможно создать массив этого класса, поэтому размер должен быть «округлен» до кратного 4, вставив прописные байты.

```

// This type shall always be aligned to a multiple of 4. Padding shall be inserted as
// needed.
// Chars<1>..Chars<4> are 4 bytes, Chars<5>..Chars<8> are 8 bytes, etc.
template<size_t SZ>
struct alignas(4) Chars { char arr[SZ]; };

static_assert(sizeof(Chars<1>) == sizeof(Chars<4>), "Alignment is strict.\n");

```


- Если два нестатических члена класса имеют один и тот же **спецификатор доступа**, то тот, который приходит позже в порядке объявления, гарантированно будет представлен позже в представлении объекта. Но если два нестатических члена имеют разные спецификаторы доступа, их относительный порядок внутри объекта не указан.
- Не определено, какой порядок субобъекты базового класса появляются внутри объекта, независимо от того, происходят ли они последовательно, и появляются ли они до, после или между субобъектами-членами.

Арифметические типы

Узкие типы символов

Тип `unsigned char` использует все биты для представления двоичного числа. Поэтому, например, если `unsigned char` имеет длину 8 бит, то 256 возможных битовых шаблонов объекта `char` представляют 256 различных значений {0, 1, ..., 255}. Число 42 гарантированно будет представлено битовой диаграммой `00101010`.

`signed char` тип `signed char` не имеет битов заполнения, т. Е. Если `signed char` имеет длину 8 бит, тогда он имеет 8 бит емкости для представления числа.

Обратите внимание, что эти гарантии не применяются к типам, отличным от типов узких символов.

Целочисленные типы

Беззнаковые целые типы используют чистую двоичную систему, но могут содержать биты заполнения. Например, для `unsigned int` может быть (хотя и маловероятно) иметь длину 64 бита, но только быть способным хранить целые числа от 0 до $2^{32} - 1$ включительно. Остальные 32 бита будут битами заполнения, которые не должны записываться напрямую.

Подписанные целочисленные типы используют двоичную систему со знаковым битом и, возможно, битами заполнения. Значения, которые относятся к общему диапазону знакового целочисленного типа и соответствующего беззнакового целочисленного типа, имеют одинаковое представление. Например, если битовый шаблон `0001010010101011` `unsigned short` объекта представляет значение `5291`, тогда он также представляет значение `5291` при интерпретации как `short` объект.

Определяется реализация, используется ли дополнение двух дополнений, дополнений или знаковых значений, поскольку все три системы удовлетворяют требованию в предыдущем абзаце.

Типы с плавающей точкой

Представление значений типов с плавающей запятой определяется реализацией. Чаще всего `float` и `double` типы соответствуют IEEE 754 и имеют длину 32 и 64 бит (так, например, `float` будет иметь 23 бита точности, которые будут следовать за 8 экспоненциальными битами и 1 знаковым битом). Однако стандарт ничего не гарантирует. Типы плавающей точки часто имеют «ловушечные представления», которые вызывают ошибки, когда они используются в вычислениях.

Массивы

Тип массива не имеет дополнений между его элементами. Следовательно, массив с типом элемента `T` является просто последовательностью объектов `T` выложенных в памяти, по порядку.

Многомерный массив представляет собой массив массивов, и приведенное выше относится к рекурсивно. Например, если у нас есть декларация

```
int a[5][3];
```

то `a` представляет собой массив из 5 массивов из 3 `int` s. Поэтому `a[0]`, состоящая из трех элементов `a[0][0]`, `a[0][1]`, `a[0][2]`, изложена в памяти до `a[1]`, которая состоит из `a[1][0]`, `a[1][1]` и `a[1][2]`. Это называется *строчным* порядком.

Прочитайте Макет типов объектов онлайн: <https://riptutorial.com/ru/cplusplus/topic/9329/макет-типов-объектов>

глава 57: Массивы

Вступление

Массивы представляют собой элементы одного типа, помещенные в соседние ячейки памяти. Элементы могут индивидуально ссылаться на уникальный идентификатор с добавленным индексом.

Это позволяет объявлять несколько значений переменных определенного типа и обращаться к ним индивидуально, без необходимости объявлять переменную для каждого значения.

Examples

Размер массива: тип `safe` во время компиляции.

```
#include <stddef.h>          // size_t, ptrdiff_t

//----- Machinery:

using Size = ptrdiff_t;

template< class Item, size_t n >
constexpr auto n_items( Item (&)[n] ) noexcept
    -> Size
{ return n; }

//----- Usage:

#include <iostream>
using namespace std;
auto main()
    -> int
{
    int const    a[]      = {3, 1, 4, 1, 5, 9, 2, 6, 5, 4};
    Size const   n        = n_items( a );
    int          b[n]      = {};          // An array of the same size as a.

    (void) b;
    cout << "Size = " << n << "\n";
}
```

Идиома C для размера массива `sizeof(a)/sizeof(a[0])` примет указатель как аргумент и в итоге даст неверный результат.

Для C ++ 11

используя C ++ 11, вы можете:

```
std::extent<decltype(MyArray)>::value;
```

Пример:

```
char MyArray[] = { 'X','o','c','e' };
const auto n = std::extent<decltype(MyArray)>::value;
std::cout << n << "\n"; // Prints 4
```

До C ++ 17 (начиная с этой записи) У C ++ не было встроенного основного языка или стандартной библиотечной утилиты для получения размера массива, но это можно реализовать, передав массив *по ссылке* на шаблон функции, так как показано выше. Точная, но важная точка: параметр размера шаблона является `size_t` , несколько несовместимым с типом результата функции подписанного `Size` , чтобы разместить компилятор g ++, который иногда настаивает на `size_t` для сопоставления шаблонов.

С C ++ 17 и более поздними версиями можно использовать `std::size` , который специализирован для массивов.

Динамический размерный массив

```
// Example of raw dynamic size array. It's generally better to use std::vector.
#include <algorithm>           // std::sort
#include <iostream>
using namespace std;

auto int_from( istream& in ) -> int { int x; in >> x; return x; }

auto main()
    -> int
{
    cout << "Sorting n integers provided by you.\n";
    cout << "n? ";
    int const    n    = int_from( cin );
    int*         a    = new int[n];          // ← Allocation of array of n items.

    for( int i = 1; i <= n; ++i )
    {
        cout << "The #" << i << " number, please: ";
        a[i-1] = int_from( cin );
    }

    sort( a, a + n );
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';

    delete[] a;
}
```

Программа, объявляющая массив `T a[n];` где `n` определяется временем выполнения, может компилироваться с определенными компиляторами, которые поддерживают *массивы вариативной длины* C99 (VLAs) в качестве расширения языка. Но VLA не поддерживаются стандартным C ++. В этом примере показано, как вручную распределить массив

динамического размера с помощью `new[]` -выражения,

```
int*      a      = new int[n];           // ← Allocation of array of n items.
```

... затем используйте его и, наконец, освободите его с помощью `delete[]` -expression:

```
delete[] a;
```

Выделенный здесь массив имеет неопределенные значения, но его можно инициализировать нулем, просто добавив пустую скобку `()`, например: `new int[n]()`. В более общем плане, для произвольного типа элемента это выполняет *инициализацию значения*.

Как часть функции вниз в иерархии вызовов этот код не будет безопасным для исключений, поскольку исключение перед выражением `delete[]` (и после `new[]`) вызовет утечку памяти. Один из способов решения этой проблемы - автоматизировать очистку, например, с помощью умного указателя `std::unique_ptr`. Но, как правило, лучший способ обратиться к нему - просто использовать `std::vector`: для этого используется `std::vector`.

Расширение массива динамических размеров с помощью `std::vector`.

```
// Example of std::vector as an expanding dynamic size array.
#include <algorithm>           // std::sort
#include <iostream>
#include <vector>              // std::vector
using namespace std;

int int_from( std::istream& in ) { int x = 0; in >> x; return x; }

int main()
{
    cout << "Sorting integers provided by you.\n";
    cout << "You can indicate EOF via F6 in Windows or Ctrl+D in Unix-land.\n";
    vector<int> a;           // ← Zero size by default.

    while( cin )
    {
        cout << "One number, please, or indicate EOF: ";
        int const x = int_from( cin );
        if( !cin.fail() ) { a.push_back( x ); } // Expands as necessary.
    }

    sort( a.begin(), a.end() );
    int const n = a.size();
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';
}
```

`std::vector` - стандартный шаблон класса библиотеки, который предоставляет понятие массива с переменным размером. Он заботится обо всем управлении памятью, а буфер смежен, поэтому указатель на буфер (например, `&v[0]` или `v.data()`) может быть передан

функциям API, требующим сырой массив. `vector` можно даже развернуть во время выполнения, например, через функцию `push_back`, добавляющую элемент.

Сложность последовательности операций n `push_back`, включая копирование или перемещение, участвующие в векторных разложениях, амортизируется $O(n)$. «Амортизировано»: в среднем.

Обычно это достигается путем удвоения вектора размера буфера, его емкости, когда требуется больший буфер. Например, для буфера, начинающегося как размер 1, и при повторном удвоении при $n = 17$ вызовах `push_back` это включает в себя операции копирования $1 + 2 + 4 + 8 + 16 = 31$, которая меньше $2 \times n = 34$. И более общая сумма этой последовательности не может превышать $2 \times n$.

По сравнению с примером необработанного массива динамического размера этот `vector` код не требует, чтобы пользователь предоставлял (и знал) количество предметов спереди. Вместо этого вектор просто расширяется по мере необходимости, для каждого нового значения элемента, указанного пользователем.

Матрица необработанного массива фиксированного размера (т. Е. 2D-массив).

```
// A fixed size raw array matrix (that is, a 2D raw array).
#include <iostream>
#include <iomanip>
using namespace std;

auto main() -> int
{
    int const    n_rows   = 3;
    int const    n_cols   = 7;
    int const    m[n_rows][n_cols] =           // A raw array matrix.
    {
        { 1, 2, 3, 4, 5, 6, 7 },
        { 8, 9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };

    for( int y = 0; y < n_rows; ++y )
    {
        for( int x = 0; x < n_cols; ++x )
        {
            cout << setw( 4 ) << m[y][x];           // Note: do NOT use m[y,x]!
        }
        cout << '\n';
    }
}
```

Выход:

```
1  2  3  4  5  6  7
8  9 10 11 12 13 14
15 16 17 18 19 20 21
```

C++ не поддерживает специальный синтаксис для индексирования многомерного массива. Вместо этого такой массив рассматривается как массив массивов (возможно, массивов и т. Д.), А для каждого уровня используется стандартная однозначная нотация `[ i ]`. В приведенном выше примере `m[y]` ссылается на строку `y` of `m`, где `y` - индекс на основе нуля. Затем эта строка может быть проиндексирована по очереди, например `m[y][x]`, которая ссылается на `x`-й элемент - или столбец строки `y`.

Т.е. последний индекс изменяется быстрее, а в объявлении диапазон этого индекса, который представляет собой число столбцов на строку, является последним и «самым внутренним» размером.

Поскольку C++ не обеспечивает встроенную поддержку массивов динамических размеров, за исключением динамического распределения, динамическая матрица размеров часто реализуется как класс. Тогда номенклатура индексирования матрицы исходного массива `m[y][x]` имеет некоторую стоимость либо путем представления реализации (так что, например, представление транспонированной матрицы становится практически невозможным) или путем добавления некоторых накладных расходов и незначительных неудобств, когда это делается путем возврата прокси-объект от `operator[]`. И поэтому индексирующее обозначение для такой абстракции может и обычно будет отличаться как по внешнему виду, так и по порядку индексов, например `m(x, y)` или `m.at(x, y)` или `m.item(x, y)`.

Матрица динамического размера, использующая `std::vector` для хранения.

К сожалению, с C++ 14 в стандартной библиотеке C++ нет динамического размера. Матричные классы, которые поддерживают динамический размер, однако, доступны из ряда 3-й партии библиотеки, включая библиотеку подталкивание `Matrix` (суб-библиотеку в библиотеке форсирования).

Если вам не нужна зависимость от Boost или какой-либо другой библиотеки, то одна динамическая матрица бедного человека на C++ похожа на

```
vector<vector<int>> m( 3, vector<int>( 7 ) );
```

... где `vector` - `std::vector`. Матрица здесь создается путем копирования вектора строки `n` раз, где `n` - количество строк, здесь 3. Преимущество состоит в том, чтобы предоставить такую же нотацию индексирования `m[y][x]` как для матрицы исходного массива фиксированного размера, но это немного неэффективно, потому что оно включает динамическое распределение для каждой строки, и это немного небезопасно, потому что возможно непреднамеренно изменить размер строки.

Более безопасным и эффективным подходом является использование одного вектора в качестве хранилища для матрицы и сопоставление кода клиента `(x, y)` с соответствующим

ИНДЕКСОМ В ЭТОМ ВЕКТОРЕ:

```
// A dynamic size matrix using std::vector for storage.

//----- Machinery:
#include <algorithm>          // std::copy
#include <assert.h>           // assert
#include <initializer_list>    // std::initializer_list
#include <vector>              // std::vector
#include <stddef.h>            // ptrdiff_t

namespace my {
    using Size = ptrdiff_t;
    using std::initializer_list;
    using std::vector;

    template< class Item >
    class Matrix
    {
    private:
        vector<Item>      items_;
        Size              n_cols_;

        auto index_for( Size const x, Size const y ) const
            -> Size
        { return y*n_cols_ + x; }

    public:
        auto n_rows() const -> Size { return items_.size()/n_cols_; }
        auto n_cols() const -> Size { return n_cols_; }

        auto item( Size const x, Size const y )
            -> Item&
        { return items_[index_for(x, y)]; }

        auto item( Size const x, Size const y ) const
            -> Item const&
        { return items_[index_for(x, y)]; }

        Matrix(): n_cols_( 0 ) {}

        Matrix( Size const n_cols, Size const n_rows )
            : items_( n_cols*n_rows )
            , n_cols_( n_cols )
        {}

        Matrix( initializer_list< initializer_list<Item> > const& values )
            : items_(
                , n_cols_( values.size() == 0? 0 : values.begin()->size() )
            )
        {
            for( auto const& row : values )
            {
                assert( Size( row.size() ) == n_cols_ );
                items_.insert( items_.end(), row.begin(), row.end() );
            }
        }
    };
} // namespace my

//----- Usage:
using my::Matrix;
```



```

auto some_matrix()
    -> Matrix<int>
{
    return
    {
        { 1, 2, 3, 4, 5, 6, 7 },
        { 8, 9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };
}

#include <iostream>
#include <iomanip>
using namespace std;
auto main() -> int
{
    Matrix<int> const m = some_matrix();
    assert( m.n_cols() == 7 );
    assert( m.n_rows() == 3 );
    for( int y = 0, y_end = m.n_rows(); y < y_end; ++y )
    {
        for( int x = 0, x_end = m.n_cols(); x < x_end; ++x )
        {
            cout << setw( 4 ) << m.item( x, y );           // ← Note: not `m[y][x]`!
        }
        cout << '\n';
    }
}

```

Выход:

```

 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21

```

Вышеприведенный код не является промышленным классом: он призван показать основные принципы и удовлетворять потребности студентов, изучающих C++.

Например, можно определить перегрузки `operator()` для упрощения нотации индексирования.

Инициализация массива

Массив - это всего лишь блок последовательных мест памяти для определенного типа переменной. Массивы распределяются так же, как и обычные переменные, но с квадратными скобками, добавленными к его имени `[]` которые содержат количество элементов, которые вписываются в память массива.

В следующем примере массива используется тип `int`, имя переменной `arrayOfInts` и количество элементов `[5]`, в которых массив имеет место для:

```
int arrayOfInts[5];
```

Массив может быть объявлен и инициализирован одновременно с этим

```
int arrayOfInts[5] = {10, 20, 30, 40, 50};
```

При инициализации массива путем перечисления всех его членов нет необходимости включать число элементов в квадратные скобки. Он будет автоматически вычисляться компилятором. В следующем примере это 5:

```
int arrayOfInts[] = {10, 20, 30, 40, 50};
```

Также возможно инициализировать только первые элементы, выделяя больше места. В этом случае определение длины в скобках является обязательным. Далее будет выделен массив длиной 5 с частичной инициализацией, компилятор инициализирует все остальные элементы со стандартным значением типа элемента, в этом случае ноль.

```
int arrayOfInts[5] = {10,20}; // means 10, 20, 0, 0, 0
```

Массивы других базовых типов данных могут быть инициализированы таким же образом.

```
char arrayOfChars[5]; // declare the array and allocate the memory, don't initialize  
char arrayOfChars[5] = { 'a', 'b', 'c', 'd', 'e' } ; //declare and initialize  
double arrayOfDoubles[5] = {1.14159, 2.14159, 3.14159, 4.14159, 5.14159};  
string arrayOfStrings[5] = { "C++", "is", "super", "duper", "great!"};
```

Важно также отметить, что при доступе к элементам массива индекс элемента (или позиция) массива начинается с 0.

```
int array[5] = { 10/*Element no.0*/, 20/*Element no.1*/, 30, 40, 50/*Element no.4*/};  
std::cout << array[4]; //outputs 50  
std::cout << array[0]; //outputs 10
```

Прочитайте Массивы онлайн: <https://riptutorial.com/ru/cplusplus/topic/3017/массивы>

глава 58: Метaproгpаммиpование

Вступление

В C++ Metaprogramming ссылается на использование макросов или шаблонов для генерации кода во время компиляции.

В общем, макросы не одобряются в этой роли, и предпочтительны шаблоны, хотя они не являются общими.

Метaproгpаммиpование шаблонов часто использует вычисления времени компиляции, будь то через шаблоны или функции `constexpr`, для достижения целей генерации кода, однако вычисления времени компиляции не являются метaproгpаммиpованием как таковым.

замечания

Метaproгpаммиpование (или, более конкретно, Template Metaprogramming) - это практика использования [шаблонов](#) для создания констант, функций или структур данных во время компиляции. Это позволяет выполнять вычисления один раз во время компиляции, а не в каждое время выполнения.

Examples

Вычисление факториалов

Факториалы могут быть вычислены во время компиляции с использованием методов метaproгpаммиpования шаблонов.

```
#include <iostream>

template<unsigned int n>
struct factorial
{
    enum
    {
        value = n * factorial<n - 1>::value
    };
};

template<>
struct factorial<0>
{
    enum { value = 1 };
};

int main()
{
```

```
std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

`factorial` - это структура, но в метапрограммировании шаблона он рассматривается как шаблонный метафунг. По соглашению метафайлы шаблонов оцениваются путем проверки определенного члена: `::type` для метафайлов, которые приводят к типам, или `::value` для метафайлов, которые генерируют значения.

В приведенном выше коде мы оцениваем `factorial` метафунгт, создавая экземпляр шаблона с параметрами, которые мы хотим передать, и используя `::value` чтобы получить результат оценки.

Сама метафония полагается на рекурсивно создающую тот же метафунгмент с меньшими значениями. `factorial<0>` представляет собой конечное условие. Метапрограммирование шаблонов имеет большинство ограничений [функционального языка программирования](#), поэтому рекурсия является основной конструкцией «looping».

Поскольку метафайлы шаблонов выполняются во время компиляции, их результаты могут использоваться в контекстах, требующих значений времени компиляции. Например:

```
int my_array[factorial<5>::value];
```

Автоматические массивы должны иметь определенный размер времени компиляции. И результат `metafunction` - константа времени компиляции, поэтому ее можно использовать здесь.

Ограничение . Большинство компиляторов не позволят глубже рекурсии выйти за пределы. Например, компилятор `g++` по умолчанию рекурсия ограничивает 256 уровней. В случае `g++` программист может установить глубину рекурсии с помощью параметра - `ftemplate-depth-X`.

C ++ 11

С C ++ 11 шаблон `std::integral_constant` может использоваться для такого расчета шаблонов:

```
#include <iostream>
#include <type_traits>

template<long long n>
struct factorial :
    std::integral_constant<long long, n * factorial<n - 1>::value> {};

template<>
struct factorial<0> :
    std::integral_constant<long long, 1> {};

int main()
{
```

```
std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

Кроме того, функции `constexpr` становятся более чистой альтернативой.

```
#include <iostream>

constexpr long long factorial(long long n)
{
    return (n == 0) ? 1 : n * factorial(n - 1);
}

int main()
{
    char test[factorial(3)];
    std::cout << factorial(7) << '\n';
}
```

Тело `factorial()` записывается как один оператор, потому что в C++ 11 функции `constexpr` могут использовать только ограниченное подмножество языка.

C++ 14

Начиная с C++ 14, многие ограничения для функций `constexpr` были отброшены, и теперь их можно записать гораздо удобнее:

```
constexpr long long factorial(long long n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Или даже:

```
constexpr long long factorial(int n)
{
    long long result = 1;
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

C++ 17

Поскольку с C++ 17 можно использовать выражение `fold` для вычисления `factorial`:

```
#include <iostream>
#include <utility>

template <class T, T N, class I = std::make_integer_sequence<T, N>>
struct factorial;
```

```
template <class T, T N, T... Is>
struct factorial<T,N,std::index_sequence<T, Is...>> {
    static constexpr T value = (static_cast<T>(1) * ... * (Is + 1));
};

int main() {
    std::cout << factorial<int, 5>::value << std::endl;
}
```

Итерация по пакету параметров

Часто нам нужно выполнить операцию над каждым элементом в пакете параметров вариационного шаблона. Есть много способов сделать это, и решения легче читать и писать с C ++ 17. Предположим, мы просто хотим напечатать каждый элемент в пакете. Самое простое решение - рекурсия:

C ++ 11

```
void print_all(std::ostream& os) {
    // base case
}

template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    print_all(os, rest...);
}
```

Вместо этого мы могли бы использовать трюк экспандера, чтобы выполнить все потоки в одной функции. Это имеет то преимущество, что не требуется вторая перегрузка, но имеет недостаток, превышающий читабельность звезд:

C ++ 11

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

Для объяснения того, как это работает, см . [Отличный ответ ТС](#) .

C ++ 17

С C ++ 17 мы получаем два мощных новых инструмента в нашем арсенале для решения этой проблемы. Первый - это сгиб-выражение:

```
template <class... Ts>
```

```
void print_all(std::ostream& os, Ts const&... args) {
    ((os << args), ...);
}
```

И второе - `if constexpr`, что позволяет нам написать наше оригинальное рекурсивное решение в одной функции:

```
template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    if constexpr (sizeof...(rest) > 0) {
        // this line will only be instantiated if there are further
        // arguments. if rest... is empty, there will be no call to
        // print_all(os).
        print_all(os, rest...);
    }
}
```

Итерация с помощью `std::integer_sequence`

Поскольку C++ 14, стандарт предоставляет шаблон класса

```
template <class T, T... Ints>
class integer_sequence;

template <std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

и генерирующий метафунд для него:

```
template <class T, T N>
using make_integer_sequence = std::integer_sequence<T, /* a sequence 0, 1, 2, ..., N-1 */ >;

template<std::size_t N>
using make_index_sequence = make_integer_sequence<std::size_t, N>;
```

Хотя это стандартно в C++ 14, это можно реализовать с помощью инструментов C++ 11.

Мы можем использовать этот инструмент для вызова функции с помощью `std::tuple` аргументов (стандартизован в C++ 17 как `std::apply`):

```
namespace detail {
    template <class F, class Tuple, std::size_t... Is>
    decltype(auto) apply_impl(F&& f, Tuple&& tpl, std::index_sequence<Is...> ) {
        return std::forward<F>(f) (std::get<Is>(std::forward<Tuple>(tpl))...);
    }
}

template <class F, class Tuple>
decltype(auto) apply(F&& f, Tuple&& tpl) {
    return detail::apply_impl(std::forward<F>(f),
```

```

        std::forward

```

Отправка тегов

Простым способом выбора функций во время компиляции является отправка функции в перегруженную пару функций, которые принимают тег как один (обычно последний) аргумент. Например, для реализации `std::advance()` мы можем отправить в категорию итератора:

```

namespace details {
    template <class RAIter, class Distance>
    void advance(RAIter& it, Distance n, std::random_access_iterator_tag) {
        it += n;
    }

    template <class BidirIter, class Distance>
    void advance(BidirIter& it, Distance n, std::bidirectional_iterator_tag) {
        if (n > 0) {
            while (n--) ++it;
        }
        else {
            while (n++) --it;
        }
    }

    template <class InputIter, class Distance>
    void advance(InputIter& it, Distance n, std::input_iterator_tag) {
        while (n--) {
            ++it;
        }
    }

    template <class Iter, class Distance>
    void advance(Iter& it, Distance n) {
        details::advance(it, n,
            typename std::iterator_traits<Iter>::iterator_category{} );
    }
}

```

Параметры `std::XY_iterator_tag` перегруженных `details::advance` - неиспользуемые параметры функции. Фактическая реализация не имеет значения (на самом деле она полностью пуста). Их единственная цель - разрешить компилятору выбирать перегрузку, на основе которой вызывается класс класса `details::advance`.

В этом примере `advance` использует `iterator_traits<T>::iterator_category` который возвращает один из классов `iterator_tag`, в зависимости от реального типа `Iter`. Построенный по

умолчанию объект типа `iterator_category<Iter>::type` затем позволяет компилятору выбрать одну из разных перегрузок `details::advance`. (Этот параметр функции, скорее всего, будет полностью оптимизирован, так как это построенный по умолчанию объект пустой `struct` и никогда не используется).

Отправка тегов может дать вам код, который намного легче читать, чем эквиваленты, используя `SFINAE` и `enable_if`.

Примечание: в то время как C++ 17, `if constexpr` может упростить реализацию `advance` в частности, он не подходит для открытых реализаций, в отличие от диспетчеризации тегов.

Определить, действительно ли выражение

Можно определить, можно ли вызвать оператор или функцию для типа. Чтобы проверить, имеет ли класс перегрузку `std::hash`, можно сделать следующее:

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type and std::true_type
#include <utility> // for std::declval

template<class, class = void>
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, decltype(std::hash<T>() (std::declval<T>()), void())>
    : std::true_type
{};
```

C++ 17

Поскольку C++ 17, `std::void_t` можно использовать для упрощения этого типа конструкции

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type, std::true_type, std::void_t
#include <utility> // for std::declval

template<class, class = std::void_t<>>
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, std::void_t<decltype(std::hash<T>() (std::declval<T>()))>>
    : std::true_type
{};
```

где `std::void_t` определяется как:

```
template< class... > using void_t = void;
```

Для определения того, является ли оператор, например `operator<`, синтаксисом, почти то же самое:

```
template<class, class = void>
struct has_less_than
    : std::false_type
{};

template<class T>
struct has_less_than<T, decltype(std::declval<T>() < std::declval<T>(), void())>
    : std::true_type
{};
```

Они могут использоваться для использования `std::unordered_map<T>` если `T` имеет перегрузку для `std::hash`, но в противном случае попытается использовать `std::map<T>`:

```
template <class K, class V>
using hash_invariant_map = std::conditional_t<
    has_hash<K>::value,
    std::unordered_map<K, V>,
    std::map<K, V>>;
```

Вычисление мощности с C ++ 11 (и выше)

С C ++ 11 и выше вычисления во время компиляции могут быть намного проще. Например, вычисление мощности заданного числа во время компиляции будет следующим:

```
template <typename T>
constexpr T calculatePower(T value, unsigned power) {
    return power == 0 ? 1 : value * calculatePower(value, power-1);
}
```

Ключевое слово `constexpr` отвечает за вычисление функции во время компиляции, тогда и только тогда, когда все требования для этого будут выполнены (см. Больше в ссылке на `constexpr`), например, все аргументы должны быть известны во время компиляции.

Примечание. В C ++ 11 функция `constexpr` должна составлять только один оператор `return`.

Преимущества: сравнивая это со стандартным способом вычисления времени компиляции, этот метод также полезен для расчетов времени выполнения. Это означает, что если аргументы функции не известны во время компиляции (например, значение и мощность задаются как входные данные через пользователя), то функция запускается во время компиляции, поэтому нет необходимости дублировать код (поскольку мы был бы вынужден в более старых стандартах C ++).

Например

```
void useExample() {
    constexpr int compileTimeCalculated = calculatePower(3, 3); // computes at compile time,
```

```

// as both arguments are known at compilation time
// and used for a constant expression.

int value;
std::cin >> value;
int runtimeCalculated = calculatePower(value, 3); // runtime calculated,
// because value is known only at runtime.
}

```

C++ 17

Другой способ вычисления мощности во время компиляции может использовать выражение `fold`:

```

#include <iostream>
#include <utility>

template <class T, T V, T N, class I = std::make_integer_sequence<T, N>>
struct power;

template <class T, T V, T N, T... Is>
struct power<T, V, N, std::integer_sequence<T, Is...>> {
    static constexpr T value = (static_cast<T>(1) * ... * (V * static_cast<bool>(Is + 1)));
};

int main() {
    std::cout << power<int, 4, 2>::value << std::endl;
}

```

Ручное различие типов при задании любого типа T

При реализации [SFINAE](#) с использованием `std::enable_if` часто бывает полезно иметь доступ к вспомогательным шаблонам, который определяет, соответствует ли данный тип `T` набору критериев.

Чтобы помочь нам в этом, стандарт уже предоставляет два типа аналоговых `std::true_type` и `std::false_type` которые являются `std::true_type` и `std::false_type`.

В следующем примере показано, как определить, является ли тип `T` указателем или нет, шаблон `is_pointer` имитирует поведение стандартного помощника `std::is_pointer`:

```

template <typename T>
struct is_pointer_: std::false_type {};

template <typename T>
struct is_pointer_<T*>: std::true_type {};

template <typename T>
struct is_pointer: is_pointer_<typename std::remove_cv<T>::type> { }

```

В приведенном выше коде есть три шага (иногда вам нужно только два):

1. Первое объявление `is_pointer_` является случаем по умолчанию и наследуется от `std::false_type`. Случай по умолчанию должен всегда наследоваться от `std::false_type`

поскольку он аналогичен « `false` состоянию ».

2. Второе объявление специализируется на шаблоне `is_pointer_` для указателя `T*` не заботясь о том, что такое `T`. Эта версия наследуется от `std::true_type`.
3. Третья декларация (реальная) просто удаляет ненужную информацию из `T` (в этом случае мы удаляем `const` и `volatile` квалификаторы), а затем отступаем к одной из двух предыдущих деклараций.

Поскольку `is_pointer<T>` является классом, для доступа к его значению необходимо:

- Use `::value`, например `is_pointer<int>::value` - `value` - статический член класса типа `bool` унаследованный от `std::true_type` или `std::false_type`;
- Построить объект этого типа, например `is_pointer<int>{} -` Это работает, потому что `std::is_pointer` наследует свой конструктор по умолчанию из `std::true_type` или `std::false_type` (которые имеют конструкторы `constexpr`) и оба `std::true_type` и `std::false_type` имеют `constexpr` преобразования `constexpr` для `bool`.

Это хорошая привычка предоставлять «вспомогательные шаблоны помощников», которые позволяют вам напрямую получать доступ к значению:

```
template <typename T>
constexpr bool is_pointer_v = is_pointer<T>::value;
```

C++ 17

В C++ 17 и выше большинство вспомогательных шаблонов уже предоставляют версию `_v`, например:

```
template< class T > constexpr bool is_pointer_v = is_pointer<T>::value;
template< class T > constexpr bool is_reference_v = is_reference<T>::value;
```

Если-то-иначе

C++ 11

Тип `std::conditional` в заголовке стандартной библиотеки `<type_traits>` может выбрать один или другой тип на основе логического значения времени компиляции:

```
template<typename T>
struct ValueOrPointer
{
    typename std::conditional<(sizeof(T) > sizeof(void*)), T*, T>::type vop;
};
```

Эта структура содержит указатель на `T` если `T` больше размера указателя или самого `T`, если он меньше или равен размеру указателя. Поэтому `sizeof(ValueOrPointer)` всегда будет `<= sizeof(void*)`.

Generic Min / Max с переменным аргументом count

C ++ 11

Можно написать обобщенную функцию (например, `min`), которая принимает различные числовые типы и произвольное количество аргументов по метапрограмме шаблона. Эта функция объявляет `min` для двух аргументов и рекурсивно для большего.

```
template <typename T1, typename T2>
auto min(const T1 &a, const T2 &b)
-> typename std::common_type<const T1&, const T2&>::type
{
    return a < b ? a : b;
}

template <typename T1, typename T2, typename ... Args>
auto min(const T1 &a, const T2 &b, const Args& ... args)
-> typename std::common_type<const T1&, const T2&, const Args& ...>::type
{
    return min(min(a, b), args...);
}

auto minimum = min(4, 5.8f, 3, 1.8, 3, 1.1, 9);
```

Прочитайте Метапрограммирование онлайн: <https://riptutorial.com/ru/cplusplus/topic/462/метапрограммирование>

глава 59: Методы рефакторинга

Вступление

Рефакторинг относится к модификации существующего кода в улучшенную версию. Хотя рефакторинг часто выполняется при смене кода для добавления функций или исправления ошибок, термин, в частности, означает улучшение кода без необходимости добавления функций или исправления ошибок.

Examples

Рефакторинг проходит

Вот программа, которая может помочь в рефакторинге. Это простая программа, использующая C++ 11, которая предназначена для вычисления и печати всех простых чисел от 1 до 100 и основана на программе, которая была отправлена на [CodeReview](#) для обзора.

```
#include <iostream>
#include <vector>
#include <cmath>

int main()
{
    int l = 100;
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < l; no += 2) {
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            } else if (primes[primecount] * primes[primecount] > no) {
                std::cout << no << "\n";
                break;
            }
        }
        if (isprime) {
            std::cout << no << " ";
            primes.push_back(no);
        }
    }
    std::cout << "\n";
}
```

Результат этой программы выглядит следующим образом:

3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Первое, что мы замечаем, это то, что программе не удастся напечатать 2 что является простым числом. Мы могли бы просто добавить строку кода, чтобы просто напечатать одну константу без изменения остальной части программы, но может быть более аккуратно *реорганизовать* программу, чтобы разбить ее на две части: одну, которая создает список простых чисел, другой, который их печатает, Вот как это может выглядеть:

```
#include <iostream>
#include <vector>
#include <cmath>

std::vector<int> prime_list(int limit)
{
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < limit; no += 2) {
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            } else if (primes[primecount] * primes[primecount] > no) {
                break;
            }
        }
        if (isprime) {
            primes.push_back(no);
        }
    }
    return primes;
}

int main()
{
    std::vector<int> primes = prime_list(100);
    for (std::size_t i = 0; i < primes.size(); ++i) {
        std::cout << primes[i] << ' ';
    }
    std::cout << '\n';
}
```

Попытка этой версии, мы видим, что она действительно работает правильно сейчас:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Следующий шаг - заметить, что предложение second `if` действительно не требуется. Логика в цикле ищет простые коэффициенты каждого заданного числа до квадратного корня из этого числа. Это работает, потому что, если есть какие-либо простые множители числа, по крайней мере один из них должен быть меньше или равен квадратному корню из этого числа. Перерабатывая только эту функцию (остальная часть программы остается прежней), мы получаем следующий результат:

```
std::vector<int> prime_list(int limit)
{
```

```

bool isprime;
std::vector<int> primes;
primes.push_back(2);
for (int no = 3; no < limit; no += 2) {
    isprime = true;
    for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
        if (no % primes[primecount] == 0) {
            isprime = false;
            break;
        }
    }
    if (isprime) {
        primes.push_back(no);
    }
}
return primes;
}

```

Мы можем идти дальше, изменяя имена переменных, чтобы быть более подробными. Например, `primecount` самом деле не является числом простых чисел. Вместо этого это переменная индекса в вектор известных простых чисел. Кроме того, в то время как `no` иногда используется как аббревиатура для «числа», в математическом письме чаще используется `n`. Мы также можем внести некоторые изменения, устранив `break`, и объявив переменные ближе к тому, где они используются.

```

std::vector<int> prime_list(int limit)
{
    std::vector<int> primes{2};
    for (int n = 3; n < limit; n += 2) {
        bool isprime = true;
        for (int i=0; isprime && primes[i] <= std::sqrt(n); ++i) {
            isprime &= (n % primes[i] != 0);
        }
        if (isprime) {
            primes.push_back(n);
        }
    }
    return primes;
}

```

Мы можем также реорганизовать `main` использовать «спектр-за», чтобы сделать его немного аккуратнее:

```

int main()
{
    std::vector<int> primes = prime_list(100);
    for (auto p : primes) {
        std::cout << p << ' ';
    }
    std::cout << '\n';
}

```

Это всего лишь один способ рефакторинга. Другие могут делать разные варианты. Тем не менее, цель рефакторинга остается той же, которая заключается в улучшении читаемости

и, возможно, производительности кода без необходимости добавления функций.

Goto Cleanup

В базовых кодах C ++, которые раньше были C, можно найти `goto cleanup` шаблона `goto cleanup`. Поскольку команда `goto` делает рабочий процесс функции сложнее понять, этого часто избегают. Часто он может быть заменен операторами `return`, циклами, функциями. Хотя с `goto cleanup` нужно избавиться от логики очистки.

```
short calculate(VectorStr **data) {
    short result = FALSE;
    VectorStr *vec = NULL;
    if (!data)
        goto cleanup; //< Could become return false

    // ... Calculation which 'new's VectorStr

    result = TRUE;
cleanup:
    delete [] vec;
    return result;
}
```

В C ++ для устранения этой проблемы можно использовать [RAII](#) :

```
struct VectorRAII final {
    VectorStr *data{nullptr};
    VectorRAII() = default;
    ~VectorRAII() {
        delete [] data;
    }
    VectorRAII(const VectorRAII &) = delete;
};

short calculate(VectorStr **data) {
    VectorRAII vec{};
    if (!data)
        return FALSE; //< Could become return false

    // ... Calculation which 'new's VectorStr and stores it in vec.data

    return TRUE;
}
```

С этого момента можно продолжить рефакторинг фактического кода. Например, заменив `VectorRAII` на `std::unique_ptr` или `std::vector`.

Прочитайте Методы рефакторинга онлайн: <https://riptutorial.com/ru/cplusplus/topic/7600/методы-рефакторинга>

глава 60: Многопоточность

Синтаксис

- `нить()`
- `thread (thread && other)`
- `явный поток (Function && func, Args && ... args)`

параметры

параметр	подробности
<code>other</code>	Принимает участие в <code>other</code> , <code>other</code> не владеют нитью больше
<code>func</code>	Функция для вызова в отдельном потоке
<code>args</code>	Аргументы для <code>func</code>

замечания

Некоторые примечания:

- Два объекта `std::thread` **никогда не** могут представлять один и тот же поток.
- Объект `std::thread` может находиться в состоянии, когда он не представляет **ни одного** потока (т. Е. После перемещения, после вызова `join` и т. Д.).

Examples

Режимы работы

Когда вы начинаете поток, он будет выполняться до тех пор, пока он не будет завершен.

Часто, в какой-то момент, вам нужно (возможно, поток уже может быть выполнен) ждать завершения потока, потому что вы хотите использовать результат, например.

```
int n;
std::thread thread{ calculateSomething, std::ref(n) };

//Doing some other stuff

//We need 'n' now!
//Wait for the thread to finish - if it is not already done
thread.join();
```

```
//Now 'n' has the result of the calculation done in the separte thread
std::cout << n << '\n';
```

Вы также можете `detach` резьбу, разрешив ее выполнить:

```
std::thread thread{ doSomething };

//Detaching the thread, we don't need it anymore (for whatever reason)
thread.detach();

//The thread will terminate when it is done, or when the main thread returns
```

Передача ссылки на поток

Вы не можете передать ссылку (или ссылку на `const`) непосредственно в поток, потому что `std::thread` будет копировать / перемещать их. Вместо этого используйте `std::reference_wrapper` :

```
void foo(int& b)
{
    b = 10;
}

int a = 1;
std::thread thread{ foo, std::ref(a) }; //'a' is now really passed as reference

thread.join();
std::cout << a << '\n'; //Outputs 10
```

```
void bar(const ComplexObject& co)
{
    co.doCalculations();
}

ComplexObject object;
std::thread thread{ bar, std::cref(object) }; //'object' is passed as const&

thread.join();
std::cout << object.getResult() << '\n'; //Outputs the result
```

Создание `std :: thread`

В C ++ потоки создаются с использованием класса `std :: thread`. Нить - отдельный поток исполнения; это аналогично тому, как помощник выполняет одну задачу, одновременно выполняя другую. Когда весь код в потоке выполняется, он *завершается* . При создании потока вам нужно передать что-то, что будет выполнено на нем. Несколько вещей, которые вы можете передать в поток:

- Бесплатные функции
- Функции участников

- Объекты Functor
 - Лямбда-выражения
-

Пример бесплатной функции - выполняет функцию в отдельном потоке ([Live Example](#)):

```
#include <iostream>
#include <thread>

void foo(int a)
{
    std::cout << a << '\n';
}

int main()
{
    // Create and execute the thread
    std::thread thread(foo, 10); // foo is the function to execute, 10 is the
                                // argument to pass to it

    // Keep going; the thread is executed separately

    // Wait for the thread to finish; we stay here until it is done
    thread.join();

    return 0;
}
```

Пример функции-члена - выполняет функцию-член в отдельном потоке ([Live Example](#)):

```
#include <iostream>
#include <thread>

class Bar
{
public:
    void foo(int a)
    {
        std::cout << a << '\n';
    }
};

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(&Bar::foo, &bar, 10); // Pass 10 to member function

    // The member function will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Пример объекта functor (пример [Live](#)):

```
#include <iostream>
#include <thread>

class Bar
{
public:
    void operator() (int a)
    {
        std::cout << a << '\n';
    }
};

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(bar, 10); // Pass 10 to functor object

    // The functor object will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Пример выражения Lambda (пример [Live](#)):

```
#include <iostream>
#include <thread>

int main()
{
    auto lambda = [](int a) { std::cout << a << '\n'; };

    // Create and execute the thread
    std::thread thread(lambda, 10); // Pass 10 to the lambda expression

    // The lambda expression will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Операции над текущей нитью

`std::this_thread` - это namespace которое имеет функции делать интересные вещи в текущем потоке из функции, из которой она вызывается.

функция	Описание
<code>get_id</code>	Возвращает идентификатор потока
<code>sleep_for</code>	Спит в течение определенного времени
<code>sleep_until</code>	Спит до определенного времени
<code>yield</code>	Перенастроить текущие потоки, придав приоритет другим потокам

Получение текущего идентификатора потоков с помощью `std::this_thread::get_id`:

```
void foo()
{
    //Print this threads id
    std::cout << std::this_thread::get_id() << '\n';
}

std::thread thread{ foo };
thread.join(); // 'threads' id has now been printed, should be something like 12556

foo(); //The id of the main thread is printed, should be something like 2420
```

Спящий в течение 3 секунд с использованием `std::this_thread::sleep_for`:

```
void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}

std::thread thread{ foo };
foo.join();

std::cout << "Waited for 3 seconds!\n";
```

Спящий до 3 часов в будущем с использованием `std::this_thread::sleep_until`:

```
void foo()
{
    std::this_thread::sleep_until(std::chrono::system_clock::now() + std::chrono::hours(3));
}

std::thread thread{ foo };
thread.join();

std::cout << "We are now located 3 hours after the thread has been called\n";
```

Разрешить другим потокам приоритет, используя `std::this_thread::yield`:

```
void foo(int a)
```

```

{
    for (int i = 0; i < al ++i)
        std::this_thread::yield(); //Now other threads take priority, because this thread
                                   //isn't doing anything important

    std::cout << "Hello World!\n";
}

std::thread thread{ foo, 10 };
thread.join();

```

Использование `std::async` вместо `std::thread`

`std::async` также может создавать потоки. По сравнению с `std::thread` он считается менее мощным, но более простым в использовании, когда вы просто хотите запустить функцию асинхронно.

Асинхронное вызов функции

```

#include <future>
#include <iostream>

unsigned int square(unsigned int i){
    return i*i;
}

int main() {
    auto f = std::async(std::launch::async, square, 8);
    std::cout << "square currently running\n"; //do something while square is running
    std::cout << "result is " << f.get() << '\n'; //getting the result from square
}

```

Общие проблемы

- `std::async` возвращает `std::future` которое содержит возвращаемое значение, которое будет вычисляться функцией. Когда это `future` будет уничтожено, он ждет, пока поток не завершится, делая ваш код эффективно одиночным. Это легко упустить из виду, когда вам не требуется возвращаемое значение:

```

std::async(std::launch::async, square, 5);
//thread already completed at this point, because the returning future got destroyed

```

- `std::async` работает без политики запуска, поэтому `std::async(square, 5);` компилирует. Когда вы это сделаете, система решит, хочет ли он создать поток или нет. Идея заключалась в том, что система выбирает поток, если в нем уже запущено больше потоков, чем он может работать эффективно. К сожалению, в реализациях обычно просто не нужно создавать поток в этой ситуации, поэтому вам нужно переопределить это поведение с помощью `std::launch::async` который заставляет систему создавать поток.

- Остерегайтесь условий гонки.

Подробнее о асинхронных [фьючерсах и обещаниях](#)

Обеспечение непрерывности резьбы

Когда деструктор `std::thread` вызывается, вызов либо `join()` или `detach()` , **должно** быть сделано. Если поток не был соединен или отсоединен, то по умолчанию будет вызываться `std::terminate` . Используя [RAII](#) , это, как правило, достаточно просто для выполнения:

```
class thread_joiner
{
public:

    thread_joiner(std::thread t)
        : t_(std::move(t))
    { }

    ~thread_joiner()
    {
        if(t_.joinable()) {
            t_.join();
        }
    }

private:

    std::thread t_;
}
```

Затем это используется так:

```
void perform_work()
{
    // Perform some work
}

void t()
{
    thread_joiner j{std::thread(perform_work)};
    // Do some other calculations while thread is running
} // Thread is automatically joined here
```

Это также обеспечивает безопасность исключений; если бы мы создали наш поток нормально, и работа, выполненная в `t()` выполняющая другие вычисления, вызвала исключение, `join()` никогда бы не вызывался в нашем потоке, и наш процесс был бы прекращен.

Переназначение объектов потоков

Мы можем создавать пустые объекты потока и назначать им работу позже.

Если мы назначим объект потока другому активному, `joinable` потоку, `std::terminate` будет вызываться автоматически до замены потока.

```
#include <thread>

void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}

//create 100 thread objects that do nothing
std::thread executors[100];

// Some code

// I want to create some threads now

for (int i = 0; i < 100; i++)
{
    // If this object doesn't have a thread assigned
    if (!executors[i].joinable())
        executors[i] = std::thread(foo);
}
```

Базовая синхронизация

Синхронизация потоков может быть выполнена с использованием мьютексов, среди других примитивов синхронизации. Существует несколько типов мьютексов, предоставляемых стандартной библиотекой, но самым простым является `std::mutex`. Чтобы заблокировать мьютекс, вы создаете на нем блокировку. Самый простой тип блокировки - `std::lock_guard`:

```
std::mutex m;
void worker() {
    std::lock_guard<std::mutex> guard(m); // Acquires a lock on the mutex
    // Synchronized code here
} // the mutex is automatically released when guard goes out of scope
```

С `std::lock_guard` мьютекс заблокирован на весь срок службы объекта блокировки. В случаях, когда вам необходимо вручную управлять регионами для блокировки, вместо этого используйте `std::unique_lock`:

```
std::mutex m;
void worker() {
    // by default, constructing a unique_lock from a mutex will lock the mutex
    // by passing the std::defer_lock as a second argument, we
    // can construct the guard in an unlocked state instead and
    // manually lock later.
    std::unique_lock<std::mutex> guard(m, std::defer_lock);
    // the mutex is not locked yet!
    guard.lock();
    // critical section
    guard.unlock();
    // mutex is again released
}
```

Использование переменных условий

Переменная условия - это примитив, используемый в сочетании с мьютезом для организации связи между потоками. Хотя это не единственный и не самый эффективный способ выполнить это, он может быть одним из самых простых для тех, кто знаком с шаблоном.

Один из них ожидает `std::condition_variable` с `std::unique_lock<std::mutex>`. Это позволяет коду безопасно исследовать состояние общего доступа, прежде чем принимать решение о продолжении приобретения.

Ниже представлен эскиз производителя-потребителя, который использует `std::thread`, `std::condition_variable`, `std::mutex` и несколько других, чтобы сделать интересными вещи.

```
#include <condition_variable>
#include <cstdint>
#include <iostream>
#include <mutex>
#include <queue>
#include <random>
#include <thread>

int main()
{
    std::condition_variable cond;
    std::mutex mtx;
    std::queue<int> intq;
    bool stopped = false;

    std::thread producer{[&]()
    {
        // Prepare a random number generator.
        // Our producer will simply push random numbers to intq.
        //
        std::default_random_engine gen{};
        std::uniform_int_distribution<int> dist{};

        std::size_t count = 4006;
        while(count--)
        {
            // Always lock before changing
            // state guarded by a mutex and
            // condition_variable (a.k.a. "condvar").
            std::lock_guard<std::mutex> L{mtx};

            // Push a random int into the queue
            intq.push(dist(gen));

            // Tell the consumer it has an int
            cond.notify_one();
        }

        // All done.
```

```

        // Acquire the lock, set the stopped flag,
        // then inform the consumer.
        std::lock_guard<std::mutex> L{mtx};

        std::cout << "Producer is done!" << std::endl;

        stopped = true;
        cond.notify_one();
    }

    std::thread consumer{[&]()
    {
        do{
            std::unique_lock<std::mutex> L{mtx};
            cond.wait(L, [&]()
            {
                // Acquire the lock only if
                // we've stopped or the queue
                // isn't empty
                return stopped || ! intq.empty();
            });

            // We own the mutex here; pop the queue
            // until it empties out.

            while( ! intq.empty())
            {
                const auto val = intq.front();
                intq.pop();

                std::cout << "Consumer popped: " << val << std::endl;
            }

            if(stopped){
                // producer has signaled a stop
                std::cout << "Consumer is done!" << std::endl;
                break;
            }

        }while(true);
    }

    consumer.join();
    producer.join();

    std::cout << "Example Completed!" << std::endl;

    return 0;
}

```

Создайте простой пул потоков

С ++ 11-примитивы с потоками по-прежнему относительно низки. Они могут использоваться для написания конструкции более высокого уровня, например пула потоков:

С ++ 14

```

struct tasks {
    // the mutex, condition variable and deque form a single
    // thread-safe triggered queue of tasks:
    std::mutex m;
    std::condition_variable v;
    // note that a packaged_task<void> can store a packaged_task<R>:
    std::deque<std::packaged_task<void()>> work;

    // this holds futures representing the worker threads being done:
    std::vector<std::future<void>> finished;

    // queue( lambda ) will enqueue the lambda into the tasks for the threads
    // to use. A future of the type the lambda returns is given to let you get
    // the result out.
    template<class F, class R=std::result_of_t<F&()>>
    std::future<R> queue(F&& f) {
        // wrap the function object into a packaged task, splitting
        // execution from the return value:
        std::packaged_task<R()> p(std::forward<F>(f));

        auto r=p.get_future(); // get the return value before we hand off the task
        {
            std::unique_lock<std::mutex> l(m);
            work.emplace_back(std::move(p)); // store the task<R()> as a task<void()>
        }
        v.notify_one(); // wake a thread to work on the task

        return r; // return the future result of the task
    }

    // start N threads in the thread pool.
    void start(std::size_t N=1){
        for (std::size_t i = 0; i < N; ++i)
        {
            // each thread is a std::async running this->thread_task():
            finished.push_back(
                std::async(
                    std::launch::async,
                    [this]{ thread_task(); }
                )
            );
        }
    }

    // abort() cancels all non-started tasks, and tells every working thread
    // stop running, and waits for them to finish up.
    void abort() {
        cancel_pending();
        finish();
    }

    // cancel_pending() merely cancels all non-started tasks:
    void cancel_pending() {
        std::unique_lock<std::mutex> l(m);
        work.clear();
    }

    // finish enques a "stop the thread" message for every thread, then waits for them:
    void finish() {
        {
            std::unique_lock<std::mutex> l(m);
            for(auto&&unused:finished){
                work.push_back({});
            }
        }
    }
}

```

```

    }
    v.notify_all();
    finished.clear();
}
~tasks() {
    finish();
}
private:
    // the work that a worker thread does:
    void thread_task() {
        while(true){
            // pop a task off the queue:
            std::packaged_task<void()> f;
            {
                // usual thread-safe queue code:
                std::unique_lock<std::mutex> l(m);
                if (work.empty()){
                    v.wait(l, [&]{return !work.empty();});
                }
                f = std::move(work.front());
                work.pop_front();
            }
            // if the task is invalid, it means we are asked to abort:
            if (!f.valid()) return;
            // otherwise, run the task:
            f();
        }
    }
};

```

`tasks.queue( []{ return "hello world"s; } )` возвращает `std::future<std::string>` , который, когда объект задач приближается к запуску, заполняется `hello world` .

Вы создаете потоки, запустив `tasks.start(10)` (который запускает 10 потоков).

Использование `packaged_task<void()>` заключается только в том, что нет эквивалента `std::function` типу, который хранит типы только для перемещения. Написание пользовательского одного из них, вероятно, будет быстрее, чем использование `packaged_task<void()>` .

[Живой пример](#) .

C ++ 11

В C ++ 11 замените `result_of_t<blah>` на `typename result_of<blah>::type` .

Подробнее о [мьютексах](#) .

Потоковое локальное хранилище

`thread_local` локальное хранилище может быть создано с использованием `thread_local` [слова](#) `thread_local` . `thread_local` переменная, объявленная с `thread_local` спецификатора `thread_local` имеет **длительность хранения потоков**.

- Каждый поток в программе имеет свою собственную копию каждой локальной переменной.
- Локальная переменная потока с функциональной (локальной) областью будет инициализирована, первый элемент управления проходит через его определение. Такая переменная неявно статична, если не объявлен `extern`.
- Локальная переменная потока с пространством имен или классом (нелокальной) будет инициализирована как часть запуска потока.
- Поточковые локальные переменные уничтожаются при завершении потока.
- Член класса может быть только поточно-локальным, если он статичен. Следовательно, будет одна копия этой переменной для потока, а не одна копия в (поток, экземпляр).

Пример:

```
void debug_counter() {  
    thread_local int count = 0;  
    Logger::log("This function has been called %d times by this thread", ++count);  
}
```

Прочитайте Многопоточность онлайн: <https://riptutorial.com/ru/cplusplus/topic/699/>
МНОГОПОТОЧНОСТЬ

глава 61: Модель памяти C ++ 11

замечания

Различные потоки, пытающиеся получить доступ к одному и тому же месту памяти, участвуют в *гонке данных*, если хотя бы одна из операций является модификацией (также известной как *операция хранилища*). Эти *расы данных* вызывают *неопределенное поведение*. Чтобы избежать их, необходимо предотвратить одновременное выполнение этими потоками таких конфликтных операций.

Примитивы синхронизации (мьютекс, критический раздел и т. П.) Могут защищать такие обращения. Модель памяти, представленная на C ++ 11, определяет два новых портативных способа синхронизации доступа к памяти в многопоточной среде: атомные операции и заборы.

Атомные операции

Теперь можно читать и записывать в заданную ячейку памяти с помощью операций *атомной нагрузки* и *атомного хранилища*. Для удобства они обернуты в класс шаблонов `std::atomic<t>`. Этот класс обортывает значение типа `t` но на этот раз *загружает и сохраняет* объект в объект атомарным.

Шаблон не доступен для всех типов. Какие типы доступны, специфичны для реализации, но обычно это большинство (или все) доступных интегральных типов, а также типы указателей. Так что `std::atomic<unsigned>` и `std::atomic<std::vector<foo>*>` должны быть доступны, а `std::atomic<std::pair<bool, char>>` скорее всего, не будет.

Атомные операции имеют следующие свойства:

- Все атомарные операции могут выполняться одновременно из нескольких потоков, не вызывая неопределенного поведения.
- *Атомная нагрузка* будет видеть либо начальное значение, с которым был сконструирован атомный объект, либо значение, записанное в него через некоторую операцию *хранения атома*.
- *Атомные хранилища* к одному и тому же атомному объекту упорядочиваются одинаково во всех потоках. Если поток уже видел значение какой-либо операции с *хранилищем атома*, последующие операции *атомной нагрузки* будут видеть либо одно и то же значение, либо значение, сохраненное в результате последующей операции *сохранения атома*.
- Операции *Atomic read-modify-write* позволяют *получать атомную нагрузку и хранить атом*, если нет другого *хранилища атомов* между ними. Например, можно атомарно

увеличивать счетчик от нескольких потоков, и приращение не будет потеряно независимо от конкуренции между потоками.

- Атомные операции получают необязательный параметр `std::memory_order` который определяет, какие дополнительные свойства имеет операция относительно других мест памяти.

станд :: memory_order	Имея в виду
<code>std::memory_order_relaxed</code>	никаких дополнительных ограничений
<code>std::memory_order_release</code> → <code>std::memory_order_acquire</code>	если <code>load-acquire</code> видит значение, хранящееся в <code>store-release</code> затем сохраняет <i>последовательность до</i> того, как <code>store-release</code> произойдет до того, как нагрузки будут упорядочены после <i>загрузки нагрузки</i>
<code>std::memory_order_consume</code>	как <code>memory_order_acquire</code> но только для зависимых нагрузок
<code>std::memory_order_acq_rel</code>	сочетает <code>load-acquire</code> и <code>store-release</code>
<code>std::memory_order_seq_cst</code>	последовательная согласованность

Эти теги порядка памяти позволяют использовать три разных дисциплины упорядочения памяти: *последовательную согласованность*, *расслабление* и *освобождение* с помощью его *освобождения от сестры*.

Последовательная последовательность

Если для атомной операции не указан порядок памяти, порядок по умолчанию соответствует *последовательной согласованности*. Этот режим также может быть явно выбран пометкой операции с помощью `std::memory_order_seq_cst`.

При таком порядке операция памяти не может пересекать атомную операцию. Все операции с памятью, секвентированные до атомной операции, происходят до атомной операции, а атомная операция выполняется перед всеми операциями памяти, которые после нее выполняются. Этот режим, пожалуй, самый простой способ рассуждать, но это также приводит к наибольшему штрафу за производительность. Он также предотвращает все оптимизаторы компилятора, которые в противном случае могли бы попытаться изменить порядок операций над атомной операцией.

Расслабленный заказ

Противоположное *последовательной непротиворечивости* *расслабленного* упорядочение памяти. Он выбирается с тегом `std::memory_order_relaxed`. Устраненная атомная операция не накладывает никаких ограничений на другие операции с памятью. Единственный эффект,

который остается, заключается в том, что операция сама по себе является атомарной.

Заказ на выпуск-Приобретение

Операцию *атомного хранилища* можно `std::memory_order_release` с помощью `std::memory_order_release` а операцию *атомной нагрузки* можно `std::memory_order_acquire` с помощью `std::memory_order_acquire` . Первая операция называется (*атомная*) *store-release*, а вторая называется (*атомарной*) *загрузкой-загрузкой* .

Когда *load-purchase* видит значение, записанное в *магазине-релизе*, происходит следующее: все операции хранения, упорядоченные до того, как *хранилище* становится видимым для операций загрузки (которые *произошли до*), которые упорядочиваются после *загрузки нагрузки* .

Операции Atomic read-modify-write также могут принимать кумулятивный тег `std::memory_order_acq_rel` . Это делает *атомную* порции *нагрузки* на операциях *атомных нагрузки-то* время *приобретает атомный магазин* участок становится *атомным магазином-релиз*.

Компилятору не разрешается перемещать операции хранения после операции хранения в *хранилище атомов* . Также не допускается перемещать операции загрузки до получения *атомной нагрузки* (или *нагрузки*).

Также обратите внимание на то, что не существует *атомной загрузки нагрузки* или *приобретения атомарного хранилища* . Попытка создания таких операций делает их *расслабленными* .

Заказ релиза-потребления

Эта комбинация похожа на *release-приобретать* , но на этот раз *атомная нагрузка* помечена `std::memory_order_consume` и становится (*атомной*) *нагрузкой* . Этот режим является таким же, как и при получении *освобождения*, с той лишь разницей, что между операциями загрузки, упорядоченными после *потребления нагрузки*, только они зависят от значения, загруженного *потреблением нагрузки* .

Заборы

Заборы также позволяют упорядочивать операции памяти между потоками. Забор - это либо забор, либо забор.

Если забор забор происходит до захвата забора, тогда сохраняются секвенсированные до того, как заборный забор будет видимым для нагрузок, секционированных после захвата забора. Чтобы гарантировать, что забор для выпуска происходит до получения забора, можно использовать другие примитивы синхронизации, включая расслабленные атомные

операции.

Examples

Необходимость в модели памяти

```
int x, y;
bool ready = false;

void init()
{
    x = 2;
    y = 3;
    ready = true;
}

void use()
{
    if (ready)
        std::cout << x + y;
}
```

Один поток вызывает функцию `init()` а другой поток (или обработчик сигнала) вызывает функцию `use()`. Можно было бы ожидать, что функция `use()` будет либо печатать 5 либо ничего не делать. Это может быть не всегда по нескольким причинам:

- ЦП может переупорядочить записи, которые происходят в `init()` так что исполняемый код может выглядеть так:

```
void init()
{
    ready = true;
    x = 2;
    y = 3;
}
```

- ЦП может изменить порядок чтения, которое происходит при `use()` так что фактически исполняемый код может стать:

```
void use()
{
    int local_x = x;
    int local_y = y;
    if (ready)
        std::cout << local_x + local_y;
}
```

- Оптимизирующий компилятор C++ может решить переупорядочить программу аналогичным образом.

Такое переупорядочение не может изменить поведение программы, выполняемой в одном потоке, потому что поток не может чередовать вызовы `init()` и `use()`. С другой стороны, в

многопоточной настройке один поток может видеть часть записей, выполняемых другим потоком, где может случиться, что `use()` может видеть `ready==true` и мусор в `x` или `y` или `y` и другое.

Модель памяти C++ позволяет программисту указать, какие операции переупорядочения разрешены, а какие нет, чтобы многопоточная программа также могла вести себя так, как ожидалось. Приведенный выше пример можно переписать в потокобезопасном виде следующим образом:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
    x = 2;
    y = 3;
    ready.store(true, std::memory_order_release);
}

void use()
{
    if (ready.load(std::memory_order_acquire))
        std::cout << x + y;
}
```

Здесь `init()` выполняет операцию *хранения с использованием атома*. Это не только сохраняет значение `true` в `ready`, но также сообщает компилятору, что он не может переместить эту операцию перед выполнением операций записи, которые были *секвенированы до нее*.

Функция `use()` выполняет операцию *получения атомарной нагрузки*. Он считывает текущее значение `ready` а также запрещает компилятору размещать операции чтения, которые *секвенированы после того, как это произойдет до получения атомной нагрузки*.

Эти атомические операции также заставляют компилятор помещать любые инструкции аппаратного обеспечения, чтобы сообщить ЦП воздерживаться от нежелательных переупорядочений.

Поскольку *атомный хранилище-хранилище* находится в том же месте памяти, что и при получении *атомной нагрузки*, модель памяти предусматривает, что если операция *загрузки нагрузки* видит значение, записанное в операции *хранения-хранилища*, то все записи, выполняемые `init()` 's до этого *хранилища* будет видна для загрузок, которые `use()` поток `use()` после его *загрузки*. То есть, если `use()` видит `ready==true`, тогда гарантировано увидеть `x==2` и `y==3`.

Обратите внимание, что компилятору и процессору по-прежнему разрешено записывать в `y` до записи на `x`, и аналогично чтение этих переменных в `use()` может происходить в любом порядке.

Пример забора

Приведенный выше пример также может быть реализован с помощью заборов и расслабленных атомных операций:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
    x = 2;
    y = 3;
    atomic_thread_fence(std::memory_order_release);
    ready.store(true, std::memory_order_relaxed);
}

void use()
{
    if (ready.load(std::memory_order_relaxed))
    {
        atomic_thread_fence(std::memory_order_acquire);
        std::cout << x + y;
    }
}
```

Если операция атомной нагрузки видит значение, записанное в хранилище атомов, то происходит до загрузки, а также ограждения: забор заготовки происходит до того, как забор заготовки делает запись в `x` и `y` которые предшествуют заборному ограждению, чтобы стать видимым к выражению `std::cout` которое следует за заборщиком.

Забор может быть полезен, если он может уменьшить общее количество операций получения, выпуска или других операций синхронизации. Например:

```
void block_and_use()
{
    while (!ready.load(std::memory_order_relaxed))
        ;
    atomic_thread_fence(std::memory_order_acquire);
    std::cout << x + y;
}
```

Функция `block_and_use()` возвращается, пока флаг `ready` будет установлен с помощью ослабленной атомной нагрузки. Затем для обеспечения необходимого порядка памяти используется один забор.

Прочитайте Модель памяти C ++ 11 онлайн: <https://riptutorial.com/ru/cplusplus/topic/7975/модель-памяти-c-plusplus-11>

глава 62: Мьютексы

замечания

Лучше использовать `std :: shared_mutex`, чем `std :: shared_timed_mutex` .

Разница в производительности более чем в два раза.

Если вы хотите использовать RWLock, вы обнаружите, что есть два варианта.

Это `std :: shared_mutex` и `shared_timed_mutex`.

вы можете думать, что `std :: shared_timed_mutex` - это только метод «`std :: shared_mutex` + `time`» версии.

Но реализация совершенно иная.

Ниже приведен код MSVC14.1: реализация `std :: shared_mutex`.

```
class shared_mutex
{
public:
    typedef _Smtx_t * native_handle_type;

    shared_mutex() _NOEXCEPT
        : _Myhandle(0)
        { // default construct
        }

    ~shared_mutex() _NOEXCEPT
        { // destroy the object
        }

    void lock() _NOEXCEPT
        { // lock exclusive
        _Smtx_lock_exclusive(&_Myhandle);
        }

    bool try_lock() _NOEXCEPT
        { // try to lock exclusive
        return (_Smtx_try_lock_exclusive(&_Myhandle) != 0);
        }

    void unlock() _NOEXCEPT
        { // unlock exclusive
        _Smtx_unlock_exclusive(&_Myhandle);
        }
```

```

void lock_shared() _NOEXCEPT
{
    // lock non-exclusive
    _Smtx_lock_shared(&_Myhandle);
}

bool try_lock_shared() _NOEXCEPT
{
    // try to lock non-exclusive
    return (_Smtx_try_lock_shared(&_Myhandle) != 0);
}

void unlock_shared() _NOEXCEPT
{
    // unlock non-exclusive
    _Smtx_unlock_shared(&_Myhandle);
}

native_handle_type native_handle() _NOEXCEPT
{
    // get native handle
    return (&_Myhandle);
}

shared_mutex(const shared_mutex&) = delete;
shared_mutex& operator=(const shared_mutex&) = delete;
private:
    _Smtx_t _Myhandle;
};

void __cdecl _Smtx_lock_exclusive(_Smtx_t * smtx)
{
    /* lock shared mutex exclusively */
    AcquireSRWLockExclusive(reinterpret_cast<PSRWLOCK> (smtx));
}

void __cdecl _Smtx_lock_shared(_Smtx_t * smtx)
{
    /* lock shared mutex non-exclusively */
    AcquireSRWLockShared(reinterpret_cast<PSRWLOCK> (smtx));
}

int __cdecl _Smtx_try_lock_exclusive(_Smtx_t * smtx)
{
    /* try to lock shared mutex exclusively */
    return (TryAcquireSRWLockExclusive(reinterpret_cast<PSRWLOCK> (smtx)));
}

int __cdecl _Smtx_try_lock_shared(_Smtx_t * smtx)
{
    /* try to lock shared mutex non-exclusively */
    return (TryAcquireSRWLockShared(reinterpret_cast<PSRWLOCK> (smtx)));
}

void __cdecl _Smtx_unlock_exclusive(_Smtx_t * smtx)
{
    /* unlock exclusive shared mutex */
    ReleaseSRWLockExclusive(reinterpret_cast<PSRWLOCK> (smtx));
}

void __cdecl _Smtx_unlock_shared(_Smtx_t * smtx)
{
    /* unlock non-exclusive shared mutex */
    ReleaseSRWLockShared(reinterpret_cast<PSRWLOCK> (smtx));
}

```

Вы можете видеть, что `std::shared_mutex` реализован в Windows Slim Reader / Write Locks ([https://msdn.microsoft.com/ko-kr/library/windows/desktop/aa904937\(v=vs.85\).aspx](https://msdn.microsoft.com/ko-kr/library/windows/desktop/aa904937(v=vs.85).aspx))

Теперь рассмотрим реализацию `std::shared_timed_mutex`.

Ниже приведен код MSVC14.1: реализация `std::shared_timed_mutex`.

```
class shared_timed_mutex
{
    typedef unsigned int _Read_cnt_t;
    static constexpr _Read_cnt_t _Max_readers = _Read_cnt_t(-1);
public:
    shared_timed_mutex() _NOEXCEPT
        : _Mymtx(), _Read_queue(), _Write_queue(),
          _Readers(0), _Writing(false)
    { // default construct
    }

    ~shared_timed_mutex() _NOEXCEPT
    { // destroy the object
    }

    void lock()
    { // lock exclusive
        unique_lock<mutex> _Lock(_Mymtx);
        while (_Writing)
            _Write_queue.wait(_Lock);
        _Writing = true;
        while (0 < _Readers)
            _Read_queue.wait(_Lock); // wait for writing, no readers
    }

    bool try_lock()
    { // try to lock exclusive
        lock_guard<mutex> _Lock(_Mymtx);
        if (_Writing || 0 < _Readers)
            return (false);
        else
        { // set writing, no readers
            _Writing = true;
            return (true);
        }
    }

    template<class _Rep,
              class _Period>
    bool try_lock_for(
        const chrono::duration<_Rep, _Period>& _Rel_time)
    { // try to lock for duration
        return (try_lock_until(chrono::steady_clock::now() + _Rel_time));
    }

    template<class _Clock,
              class _Duration>
    bool try_lock_until(
        const chrono::time_point<_Clock, _Duration>& _Abs_time)
    { // try to lock until time point
        auto _Not_writing = [this] { return (!_Writing); };
        auto _Zero_readers = [this] { return (_Readers == 0); };
        unique_lock<mutex> _Lock(_Mymtx);
```

```

    if (!_Write_queue.wait_until(_Lock, _Abs_time, _Not_writing))
        return (false);

    _Writing = true;

    if (!_Read_queue.wait_until(_Lock, _Abs_time, _Zero_readers))
    {
        // timeout, leave writing state
        _Writing = false;
        _Lock.unlock(); // unlock before notifying, for efficiency
        _Write_queue.notify_all();
        return (false);
    }

    return (true);
}

void unlock()
{
    // unlock exclusive
    {
        // unlock before notifying, for efficiency
        lock_guard<mutex> _Lock(_Mymtx);

        _Writing = false;
    }

    _Write_queue.notify_all();
}

void lock_shared()
{
    // lock non-exclusive
    unique_lock<mutex> _Lock(_Mymtx);
    while (_Writing || _Readers == _Max_readers)
        _Write_queue.wait(_Lock);
    ++_Readers;
}

bool try_lock_shared()
{
    // try to lock non-exclusive
    lock_guard<mutex> _Lock(_Mymtx);
    if (_Writing || _Readers == _Max_readers)
        return (false);
    else
    {
        // count another reader
        ++_Readers;
        return (true);
    }
}

template<class _Rep,
         class _Period>
bool try_lock_shared_for(
    const chrono::duration<_Rep, _Period>& _Rel_time)
{
    // try to lock non-exclusive for relative time
    return (try_lock_shared_until(_Rel_time
        + chrono::steady_clock::now()));
}

template<class _Time>
bool _Try_lock_shared_until(_Time _Abs_time)
{
    // try to lock non-exclusive until absolute time
    auto _Can_acquire = [this] {

```



```

        return (!_Writing && _Readers < _Max_readers); };

unique_lock<mutex> _Lock(_Mymtx);

if (!_Write_queue.wait_until(_Lock, _Abs_time, _Can_acquire))
    return (false);

++_Readers;
return (true);
}

template<class _Clock,
        class _Duration>
bool try_lock_shared_until(
    const chrono::time_point<_Clock, _Duration>& _Abs_time)
{    // try to lock non-exclusive until absolute time
    return (_Try_lock_shared_until(_Abs_time));
}

bool try_lock_shared_until(const xtime *_Abs_time)
{    // try to lock non-exclusive until absolute time
    return (_Try_lock_shared_until(_Abs_time));
}

void unlock_shared()
{    // unlock non-exclusive
    _Read_cnt_t _Local_readers;
    bool _Local_writing;

    {    // unlock before notifying, for efficiency
        lock_guard<mutex> _Lock(_Mymtx);
        --_Readers;
        _Local_readers = _Readers;
        _Local_writing = _Writing;
    }

    if (_Local_writing && _Local_readers == 0)
        _Read_queue.notify_one();
    else if (!_Local_writing && _Local_readers == _Max_readers - 1)
        _Write_queue.notify_all();
}

shared_timed_mutex(const shared_timed_mutex&) = delete;
shared_timed_mutex& operator=(const shared_timed_mutex&) = delete;
private:
mutex _Mymtx;
condition_variable _Read_queue, _Write_queue;
_Read_cnt_t _Readers;
bool _Writing;
};

class stl_condition_variable_win7 final : public stl_condition_variable_interface
{
public:
    stl_condition_variable_win7()
    {
        __crtInitializeConditionVariable(&m_condition_variable);
    }

    ~stl_condition_variable_win7() = delete;
    stl_condition_variable_win7(const stl_condition_variable_win7&) = delete;

```

```

    stl_condition_variable_win7& operator=(const stl_condition_variable_win7&) = delete;

    virtual void destroy() override {}

    virtual void wait(stl_critical_section_interface *lock) override
    {
        if (!stl_condition_variable_win7::wait_for(lock, INFINITE))
            std::terminate();
    }

    virtual bool wait_for(stl_critical_section_interface *lock, unsigned int timeout) override
    {
        return __crtSleepConditionVariableSRW(&m_condition_variable,
        static_cast<stl_critical_section_win7 *>(lock)->native_handle(), timeout, 0) != 0;
    }

    virtual void notify_one() override
    {
        __crtWakeConditionVariable(&m_condition_variable);
    }

    virtual void notify_all() override
    {
        __crtWakeAllConditionVariable(&m_condition_variable);
    }

private:
    CONDITION_VARIABLE m_condition_variable;
};

```

Вы можете видеть, что `std::shared_timed_mutex` реализован в `std::condition_variable`.

Это огромная разница.

Итак, давайте проверим производительность двух из них.

```

STLSharedMutex READ :           486647
STLSharedMutex WRITE :          205986
TOTAL READ&WRITE :             692633

STLSharedTimedMutex READ :       140291
STLSharedTimedMutex WRITE :      178849
TOTAL READ&WRITE :             319140

```

Это результат теста чтения / записи на 1000 миллисекунд.

`std::shared_mutex` обрабатывает чтение / запись в 2 раза больше, чем `std::shared_timed_mutex`.

В этом примере коэффициент чтения / записи одинаков, но скорость чтения более частая, чем скорость записи в реальном режиме.

Поэтому разница в производительности может быть больше.

код ниже - это код в этом примере.

```

void useSTLSharedMutex()
{
    std::shared_mutex shared_mtx_lock;

    std::vector<std::thread> readThreads;
    std::vector<std::thread> writeThreads;

    std::list<int> data = { 0 };
    volatile bool exit = false;

    std::atomic<int> readProcessedCnt(0);
    std::atomic<int> writeProcessedCnt(0);

    for (unsigned int i = 0; i < std::thread::hardware_concurrency(); i++)
    {
        readThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&readProcessedCnt]() {
            std::list<int> mydata;
            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock_shared();

                mydata.push_back(data.back());
                ++localProcessCnt;

                shared_mtx_lock.unlock_shared();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&readProcessedCnt, localProcessCnt);

        }));

        writeThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&writeProcessedCnt]() {

            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock();

                data.push_back(rand() % 100);
                ++localProcessCnt;

                shared_mtx_lock.unlock();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&writeProcessedCnt, localProcessCnt);

        }));
    }
}

```

```

std::this_thread::sleep_for(std::chrono::milliseconds(MAIN_WAIT_MILLISECONDS));
exit = true;

for (auto &r : readThreads)
    r.join();

for (auto &w : writeThreads)
    w.join();

std::cout << "STLSharedMutex READ :          " << readProcessedCnt << std::endl;
std::cout << "STLSharedMutex WRITE :         " << writeProcessedCnt << std::endl;
std::cout << "TOTAL READ&WRITE :             " << readProcessedCnt + writeProcessedCnt <<
std::endl << std::endl;
}

void useSTLSharedTimedMutex()
{
    std::shared_timed_mutex shared_mtx_lock;

    std::vector<std::thread> readThreads;
    std::vector<std::thread> writeThreads;

    std::list<int> data = { 0 };
    volatile bool exit = false;

    std::atomic<int> readProcessedCnt(0);
    std::atomic<int> writeProcessedCnt(0);

    for (unsigned int i = 0; i < std::thread::hardware_concurrency(); i++)
    {
        readThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&readProcessedCnt]() {
            std::list<int> mydata;
            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock_shared();

                mydata.push_back(data.back());
                ++localProcessCnt;

                shared_mtx_lock.unlock_shared();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&readProcessedCnt, localProcessCnt);

        }));

        writeThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&writeProcessedCnt]() {

            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock();

```

```

        data.push_back(rand() % 100);
        ++localProcessCnt;

        shared_mtx_lock.unlock();

        if (exit)
            break;
    }

    std::atomic_fetch_add(&writeProcessedCnt, localProcessCnt);

    }));
}

std::this_thread::sleep_for(std::chrono::milliseconds(MAIN_WAIT_MILLISECONDS));
exit = true;

for (auto &r : readThreads)
    r.join();

for (auto &w : writeThreads)
    w.join();

std::cout << "STLSharedTimedMutex READ :      " << readProcessedCnt << std::endl;
std::cout << "STLSharedTimedMutex WRITE :     " << writeProcessedCnt << std::endl;
std::cout << "TOTAL READ&WRITE :          " << readProcessedCnt + writeProcessedCnt <<
std::endl << std::endl;
}

```

Examples

`std::unique_lock`, `std::shared_lock`, `std::lock_guard`

Используется для воспроизведения в стиле RAII попыток блокировки, синхронизированных попыток блокировки и рекурсивных блокировок.

`std::unique_lock` допускает эксклюзивное владение мьютексами.

`std::shared_lock` разрешает совместное использование мьютексов. Несколько потоков могут содержать `std::shared_locks` на `std::shared_mutex`. Доступно с C++ 14.

`std::lock_guard` - легкая альтернатива `std::unique_lock` и `std::shared_lock`.

```

#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
public:
    std::string getPhoneNo( const std::string & name )
    {

```

```

        std::shared_lock<std::shared_timed_mutex> l(_protect);
        auto it = _phonebook.find( name );
        if ( it != _phonebook.end() )
            return (*it).second;
        return "";
    }
    void addPhoneNo ( const std::string & name, const std::string & phone )
    {
        std::unique_lock<std::shared_timed_mutex> l(_protect);
        _phonebook[name] = phone;
    }

    std::shared_timed_mutex _protect;
    std::unordered_map<std::string, std::string> _phonebook;
};

```

Стратегии для классов блокировки: `std::try_to_lock`, `std::prinyatie_lock`, `std::defer_lock`

При создании `std::unique_lock` существует три стратегии блокировки: `std::try_to_lock`, `std::defer_lock` И `std::adopt_lock`

1. `std::try_to_lock` позволяет попробовать блокировку без блокировки:

```

{
    std::atomic_int temp {0};
    std::mutex _mutex;

    std::thread t( [&]() {

        while( temp!= -1){
            std::this_thread::sleep_for(std::chrono::seconds(5));
            std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);

            if(lock.owns_lock()){
                //do something
                temp=0;
            }
        }
    });

    while ( true )
    {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
        if(lock.owns_lock()){
            if (temp < INT_MAX){
                ++temp;
            }
            std::cout << temp << std::endl;
        }
    }
}

```

2. `std::defer_lock` позволяет создать структуру блокировки без приобретения блокировки. При блокировке более одного мьютекса есть окно возможностей для

тупиковой ситуации, если два вызывающих абонента пытаются приобрести блокировки одновременно:

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
    std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
    lock1.lock();
    lock2.lock(); // deadlock here
    std::cout << "Locked! << std::endl;
    //...
}
```

С помощью следующего кода, что бы ни случилось в этой функции, блокировки будут получены и выпущены в соответствующем порядке:

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
    std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
    std::lock(lock1, lock2); // no deadlock possible
    std::cout << "Locked! << std::endl;
    //...
}
```

3. `std::adopt_lock` не пытается заблокировать второй раз, если вызывающий поток в настоящее время владеет блокировкой.

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::adopt_lock);
    std::unique_lock<std::mutex> lock2(_mutex2, std::adopt_lock);
    std::cout << "Locked! << std::endl;
    //...
}
```

Что-то нужно иметь в виду, так это то, что `std::lock` не является заменой рекурсивного использования мьютекса. Когда блокировка выходит из сферы действия, мьютекс **освобождается**.

станд :: мьютекс

`std::mutex` - это простая, нерекурсивная структура синхронизации, которая используется для защиты данных, к которым обращаются несколько потоков.

```
std::atomic_int temp{0};
std::mutex _mutex;

std::thread t( [&]() {
    while( temp != -1 ) {
        std::this_thread::sleep_for( std::chrono::seconds(5) );
        std::unique_lock<std::mutex> lock( _mutex );
```

```

        temp=0;
    }
    });

while ( true )
{
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
    if ( temp < INT_MAX )
        temp++;
    cout << temp << endl;
}

```

std :: scoped_lock (C ++ 17)

`std::scoped_lock` предоставляет семантику стиля RAII для `std::scoped_lock` одного мьютекса в сочетании с алгоритмами предотвращения блокировки, используемыми `std::lock`. Когда `std::scoped_lock` уничтожается, мьютексы освобождаются в обратном порядке, откуда они будут получены.

```

{
    std::scoped_lock lock{_mutex1,_mutex2};
    //do something
}

```

Типы Мьютекс

C ++ 1x предлагает выбор классов `mutex`:

- `std :: mutex` - предлагает простые функции блокировки.
- `std :: timed_mutex` - предлагает функцию `try_to_lock`
- `std :: recursive_mutex` - позволяет рекурсивную блокировку по тому же потоку.
- `std :: shared_mutex`, `std :: shared_timed_mutex` - предлагает общие и уникальные функции блокировки.

станд :: замок

`std::lock` использует алгоритмы избежания блокировки для блокировки одного или нескольких мьютексов. Если во время вызова блокируется несколько исключений, `std::lock` разблокирует успешно заблокированные объекты перед повторным броском исключения.

```
std::lock(_mutex1, _mutex2);
```

Прочитайте Мьютексы онлайн: <https://riptutorial.com/ru/cplusplus/topic/9895/мьютексы>

глава 63: Неопределенное поведение

Вступление

Что такое неопределенное поведение (UB)? Согласно стандарту ISO C ++ (§1.3.24, N4296), это «поведение, для которого настоящий международный стандарт не предъявляет никаких требований».

Это означает, что когда программа встречается UB, ей разрешено делать все, что захочет. Это часто означает сбой, но он не может просто ничего не делать, [делают демоны вылетают из вашего носа](#), или даже по всей *видимости*, работать должным образом!

Излишне говорить, что вам следует избегать написания кода, который вызывает UB.

замечания

Если программа содержит неопределенное поведение, стандарт C ++ не устанавливает ограничений на его поведение.

- Возможно, он работает как разработчик, но может также привести к сбою или получить странные результаты.
- Поведение может варьироваться в зависимости от прогонов одной и той же программы.
- Любая часть программы может работать некорректно, включая строки, которые идут до строки, содержащей неопределенное поведение.
- Реализация не требуется для документирования результата неопределенного поведения.

Реализация *может* документировать результат операции, которая производит неопределенное поведение в соответствии со стандартом, но программа, которая зависит от такого документированного поведения, не переносима.

Почему существует неопределенное поведение

Интуитивно неопределенное поведение считается плохой, так как такие ошибки не могут быть обработаны любезно, скажем, обработчиками исключений.

Но оставляя какое-то поведение неопределенным, на самом деле является неотъемлемой частью обещания C ++ «вы не платите за то, что не используете». Неопределенное поведение позволяет компилятору предположить, что разработчик знает, что он делает, и не вводит код для проверки ошибок, указанных в приведенных выше примерах.

Поиск и устранение неопределенного поведения

Некоторые инструменты могут использоваться для обнаружения неопределенного поведения во время разработки:

- У большинства компиляторов есть предупреждающие флаги, предупреждающие о некоторых случаях неопределенного поведения во время компиляции.
- Более новые версии gcc и clang включают в себя так называемый флаг «Undefined Behavior Sanitizer» (`-fsanitize=undefined`), который будет проверять неопределенное поведение во время выполнения при стоимости исполнения.
- `lint` подобные инструменты могут выполнять более тщательный неопределенный анализ поведения.

Неопределенное, неопределенное и **определяемое реализацией** поведение

Из стандарта C ++ 14 (ISO / IEC 14882: 2014), раздел 1.9 (Исполнение программы):

1. Семантические описания в этом международном стандарте определяют параметризованную недетерминированную абстрактную машину. [РЕЗАТЬ]
2. Некоторые аспекты и операции абстрактной машины описаны в этом Международном стандарте в качестве **реализации** (например, `sizeof(int)`). Они составляют *параметры абстрактной машины* . Каждая реализация должна включать документацию, описывающую ее характеристики и поведение в этих отношениях. [РЕЗАТЬ]
3. Некоторые другие аспекты и операции абстрактной машины описаны в этом Международном стандарте как **неопределенные** (например, оценка выражений в *новом-инициализаторе*, если функция распределения не выделяет память). Там, где это возможно, настоящий международный стандарт определяет набор допустимых видов поведения. Они определяют недетерминированные аспекты абстрактной машины. Таким образом, экземпляр абстрактной машины может иметь более одного возможного исполнения для данной программы и заданного ввода.
4. Некоторые другие операции описаны в этом Международном стандарте как **неопределенные** (или пример, эффект попытки изменения объекта `const`). [*Примечание* : этот международный стандарт не налагает никаких требований на поведение программ, которые содержат неопределенное поведение. - *конечная нота*]

Examples

Чтение или запись через нулевой указатель

```
int *ptr = nullptr;
*ptr = 1; // Undefined behavior
```

Это **неопределенное поведение** , потому что нулевой указатель не указывает на какой-либо действительный объект, поэтому в `*ptr` для записи нет объекта.

Хотя это чаще всего вызывает ошибку сегментации, оно не определено, и все может случиться.

Нет оператора возврата для функции с непустым возвратным типом

Опускание оператора `return` в функции, которая имеет тип возврата, который не является `void` является **неопределенным поведением** .

```
int function() {  
    // Missing return statement  
}  
  
int main() {  
    function(); //Undefined Behavior  
}
```

Большинство современных компиляторов дня выдают предупреждение во время компиляции для такого рода неопределенного поведения.

Примечание: `main` правило является единственным исключением из правила. Если `main` не имеет оператора `return` , компилятор автоматически вставляет `return 0;` для вас, поэтому его можно безопасно исключить.

Изменение строкового литерала

C ++ 11

```
char *str = "hello world";  
str[0] = 'H';
```

"hello world" - строковый литерал, поэтому его изменение дает неопределенное поведение.

Инициализация `str` в приведенном выше примере официально устарела (запланирована для удаления из будущей версии стандарта) в C ++ 03. Ряд компиляторов до 2003 года может выдать предупреждение об этом (например, подозрительное преобразование). После 2003 года компиляторы обычно предупреждают об отказоустойчивом преобразовании.

C ++ 11

Вышеприведенный пример является незаконным и приводит к диагностике компилятора в C ++ 11 и более поздних версиях. Подобный пример может быть сконструирован так, чтобы демонстрировать неопределенное поведение, явно разрешая преобразование типа, например:

```
char *str = const_cast<char *>("hello world");
str[0] = 'H';
```

Доступ к индексу вне пределов

Неопределенное поведение позволяет получить доступ к индексу, который выходит за рамки для массива (или стандартного библиотечного контейнера, поскольку все они реализованы с использованием *необработанного* массива):

```
int array[] = {1, 2, 3, 4, 5};
array[5] = 0; // Undefined behavior
```

Разрешено иметь указатель, указывающий на конец массива (в этом случае `array + 5`), вы просто не можете его разыменовать, так как он не является допустимым элементом.

```
const int *end = array + 5; // Pointer to one past the last index
for (int *p = array; p != end; ++p)
    // Do something with `p`
```

В общем, вам не разрешено создавать указатель вне пределов. Указатель должен указывать на элемент внутри массива или один за концом.

Целочисленное деление на ноль

```
int x = 5 / 0; // Undefined behavior
```

Разделение на 0 является математически неопределенным, и, как таковое, имеет смысл, что это неопределенное поведение.

Тем не менее:

```
float x = 5.0f / 0.0f; // x is +infinity
```

Большинство реализации реализуют IEEE-754, который определяет деление с плавающей запятой на ноль, чтобы вернуть NaN (если числитель равен 0.0f), infinity (если числитель положителен) или -infinity (если числитель отрицательный).

Подписанное переполнение целых чисел

```
int x = INT_MAX + 1;

// x can be anything -> Undefined behavior
```

Если во время оценки выражения результат не определяется математически или нет в диапазоне представляемых значений для его типа, поведение не определено.

(C ++ 11 Стандарт, пункт 5/4)

Это один из самых неприятных, поскольку он обычно дает воспроизводимое, не сбивающее с толку поведение, поэтому разработчики могут испытывать соблазн сильно полагаться на наблюдаемое поведение.

С другой стороны:

```
unsigned int x = UINT_MAX + 1;

// x is 0
```

хорошо определен, поскольку:

Беззнаковые целые числа, объявленные без знака, должны подчиняться законам арифметики по модулю 2^n где n - количество бит в представлении значений этого конкретного размера целого числа.

(C ++ 11 Стандарт, пункт 3.9.1 / 4)

Иногда компиляторы могут использовать неопределенное поведение и оптимизировать

```
signed int x ;
if(x > x + 1)
{
    //do something
}
```

Здесь, поскольку знаковое целочисленное переполнение не определено, компилятор может предположить, что он может никогда не произойти, и, следовательно, он может оптимизировать блок «if»

Использование неинициализированной локальной переменной

```
int a;
std::cout << a; // Undefined behavior!
```

Это приводит к **неопределенному поведению** , поскольку `a` не инициализируется.

Часто неверно заявляется, что это потому, что значение «неопределено» или «какое бы значение в этом месте памяти раньше». Тем не менее, это действие доступа к значению `a` в приведенном выше примере, которое дает неопределенное поведение. На практике печать «значения мусора» является общим симптомом в этом случае, но это только одна возможная форма неопределенного поведения.

Хотя на практике это маловероятно (поскольку оно зависит от конкретной аппаратной поддержки), компилятор может одинаково хорошо сжечь программиста при компиляции

вышеприведенного примера кода. При такой поддержке компилятора и аппаратного обеспечения такой ответ на неопределенное поведение заметно увеличил бы среднее (живое) программирование понимание истинного значения неопределенного поведения, а именно, что стандарт не ограничивает результирующее поведение.

C ++ 14

Использование неопределенного значения типа `unsigned char` не приводит к неопределенному поведению, если значение используется как:

- второй или третий операнд тернарного условного оператора;
- правый операнд встроенного оператора запятой;
- операнд преобразования в `unsigned char` ;
- правый операнд оператора присваивания, если левый операнд также имеет тип `unsigned char` ;
- инициализатор для объекта `unsigned char` ;

или если значение отбрасывается. В таких случаях неопределенное значение просто распространяется на результат выражения, если это применимо.

Обратите внимание, что `static` переменная **всегда** инициализируется нулем (если возможно):

```
static int a;
std::cout << a; // Defined behavior, 'a' is 0
```

Несколько неидентичных определений (правило одного определения)

Если класс, enum, встроенная функция, шаблон или член шаблона имеют внешнюю связь и определены в нескольких единицах трансляции, все определения должны быть идентичными или поведение не определено в соответствии с [Правилом Единого определения \(ODR\)](#) .

foo.h :

```
class Foo {
public:
    double x;
private:
    int y;
};

Foo get_foo();
```

foo.cpp :

```
#include "foo.h"
Foo get_foo() { /* implementation */ }
```

main.cpp :

```
// I want access to the private member, so I am going to replace Foo with my own type
class Foo {
public:
    double x;
    int y;
};
Foo get_foo(); // declare this function ourselves since we aren't including foo.h
int main() {
    Foo foo = get_foo();
    // do something with foo.y
}
```

Вышеупомянутая программа демонстрирует неопределенное поведение, поскольку она содержит два определения класса `::Foo`, который имеет внешнюю связь, в разных единицах перевода, но эти два определения не идентичны. В отличие от переопределения класса внутри той же единицы перевода, эта проблема не требуется для диагностики компилятором.

Неправильное сопряжение выделения и освобождения памяти

Объект может быть освобожден только `delete` если он был назначен `new` и не является массивом. Если аргумент для `delete` не был возвращен `new` или является массивом, поведение не определено.

Объект может быть освобожден только путем `delete[]` если он был назначен `new` и является массивом. Если аргумент для `delete[]` не был возвращен `new` или не является массивом, поведение не определено.

Если аргумент `free` не был возвращен `malloc`, поведение не определено.

```
int* p1 = new int;
delete p1; // correct
// delete[] p1; // undefined
// free(p1); // undefined

int* p2 = new int[10];
delete[] p2; // correct
// delete p2; // undefined
// free(p2); // undefined

int* p3 = static_cast<int*>(malloc(sizeof(int)));
free(p3); // correct
// delete p3; // undefined
// delete[] p3; // undefined
```

Таких проблем можно избежать, полностью избегая `malloc` и `free` в программах на C++, предпочитая интеллектуальные указатели стандартной библиотеки поверх raw `new` и `delete` и предпочитая `std::vector` и `std::string` поверх raw `new` и `delete[]`.

Доступ к объекту как неправильному типу

В большинстве случаев незаконно обращаться к объекту одного типа, как если бы он был другого типа (без учета св-квалификаторов). Пример:

```
float x = 42;
int y = reinterpret_cast<int*>(x);
```

Результатом является неопределенное поведение.

Есть некоторые исключения из этого правила *строгого сглаживания*:

- К объекту типа класса можно получить доступ, как если бы он был типа, который является базовым классом фактического типа класса.
- Любой тип может быть доступен как `char` или `unsigned char`, но обратное неверно: массив символов недоступен, как если бы он был произвольным типом.
- Подписанный целочисленный тип можно получить как соответствующий неподписанный тип и *наоборот*.

Связанное правило состоит в том, что если нестатическая функция-член вызывается в объекте, который фактически не имеет того же типа, что и определяющий класс функции, или производный класс, то происходит неопределенное поведение. Это верно, даже если функция не имеет доступа к объекту.

```
struct Base {
};
struct Derived : Base {
    void f() {}
};
struct Unrelated {};
Unrelated u;
Derived& r1 = reinterpret_cast<Derived*>(u); // ok
r1.f(); // UB
Base b;
Derived& r2 = reinterpret_cast<Derived*>(b); // ok
r2.f(); // UB
```

Переполнение плавающей точки

Если арифметическая операция, которая дает тип с плавающей точкой, выдает значение, которое не находится в диапазоне представляемых значений типа результата, поведение не определено в соответствии со стандартом C++, но может быть определено другими стандартами, которые может соответствовать машине, таких как IEEE 754.

```
float x = 1.0;
for (int i = 0; i < 10000; i++) {
    x *= 10.0; // will probably overflow eventually; undefined behavior
}
```


Вызов (чистых) виртуальных членов из конструктора или деструктора

Стандарт (10.4) гласит:

Функции-члены могут быть вызваны из конструктора (или деструктора) абстрактного класса; эффект виртуального вызова (10.3) на чистую виртуальную функцию, прямо или косвенно для создаваемого (или уничтоженного) объекта из такого конструктора (или деструктора), не определен.

В более общем плане некоторые авторитеты C++, например Скотт Майерс, [предлагают](#) никогда не вызывать виртуальные функции (даже не чистые) от конструкторов и деструкторов.

Рассмотрим следующий пример, измененный из приведенной выше ссылки:

```
class transaction
{
public:
    transaction(){ log_it(); }
    virtual void log_it() const = 0;
};

class sell_transaction : public transaction
{
public:
    virtual void log_it() const { /* Do something */ }
};
```

Предположим, мы создали объект `sell_transaction`:

```
sell_transaction s;
```

Это неявно вызывает конструктор `sell_transaction`, который сначала вызывает конструктор `transaction`. Когда конструктор `transaction` вызывается хотя, объект еще не является типом `sell_transaction`, а скорее только `transaction` типа.

Следовательно, вызов в `transaction::transaction()` в `log_it` не будет делать то, что может показаться интуитивным, а именно `call sell_transaction::log_it`.

- Если `log_it` является чисто виртуальным, как в этом примере, поведение не определено.
- Если `log_it` является чистым виртуальным, будет вызываться `transaction::log_it`.

Удаление производного объекта с помощью указателя на базовый класс, который не имеет виртуального деструктора.

```
class base { };
class derived: public base { };

int main() {
    base* p = new derived();
    delete p; // The is undefined behavior!
}
```

В разделе [expr.delete] §5.3.5 / 3 стандарт говорит, что если `delete` вызывается на объекте, статический тип которого не имеет `virtual` деструктора:

если статический тип подлежащего удалению объекта отличается от его динамического типа, статический тип должен быть базовым классом динамического типа подлежащего удалению объекта, а статический тип должен иметь виртуальный деструктор или поведение не определено.

Это имеет место независимо от того, добавил ли производный класс какие-либо члены данных в базовый класс.

Доступ к справочной ссылке

Недопустимо обращение к ссылке на объект, который вышел из сферы действия или иначе уничтожен. Такая ссылка называется *свисающей*, поскольку она больше не относится к действительному объекту.

```
#include <iostream>
int& getX() {
    int x = 42;
    return x;
}
int main() {
    int& r = getX();
    std::cout << r << "\n";
}
```

В этом примере локальная переменная `x` выходит из области действия при `getX`. (Обратите внимание, что *продление жизни* не может продлить время жизни локальной переменной за пределы области, в которой она определена.) Поэтому `r` является оборванной ссылкой. Эта программа имеет неопределенное поведение, хотя в некоторых случаях она может работать и печатать `42`.

Расширение пространства имен `std` или `posix`

[Стандарт \(17.6.4.2.1 / 1\)](#) обычно запрещает расширение пространства имен `std`:

Поведение программы на C++ не определено, если оно добавляет объявления или определения в пространство имен `std` или в пространство имен в пространстве имен `std`, если не указано иное.

То же самое касается `posix` (17.6.4.2.2 / 1):

Поведение программы на C++ не определено, если оно добавляет объявления или определения в пространство имен `posix` или в пространство имен в пространстве имен `posix`, если не указано иное.

Рассмотрим следующее:

```
#include <algorithm>

namespace std
{
    int foo(){}
}
```

Ничто в стандарте не запрещает `algorithm` (или один из его заголовков), определяющий одно и то же определение, и поэтому этот код будет нарушать [правило одного определения](#).

Так, в общем, это запрещено. Однако есть [определенные исключения](#). Возможно, наиболее полезно, чтобы добавлять специализации для определенных пользователем типов. Так, например, предположим, что ваш код имеет

```
class foo
{
    // Stuff
};
```

Тогда все нормально.

```
namespace std
{
    template<>
    struct hash<foo>
    {
    public:
        size_t operator()(const foo &f) const;
    };
}
```

Переполнение во время преобразования в тип или с плавающей запятой

Если при конверсии:

- целочисленный тип для типа с плавающей точкой,
- тип с плавающей точкой для целочисленного типа или
- тип с плавающей точкой для более короткого типа с плавающей точкой,

значение источника находится вне диапазона значений, которые могут быть представлены в типе назначения, результатом является неопределенное поведение. Пример:

```
double x = 1e100;
int y = x; // int probably cannot hold numbers that large, so this is UB
```

Недействительный статический приведение базы к производным

Если `static_cast` используется для преобразования указателя (соответствующей ссылки) в базовый класс в указатель (соотв. Ссылку) на производный класс, но операнд не указывает (соответственно ссылаться) на объект типа производного класса, поведение не определено. См. Раздел [База для производного преобразования](#).

Вызов функции с помощью неправильного типа указателя функции

Чтобы вызвать функцию через указатель функции, тип указателя функции должен точно соответствовать типу функции. В противном случае поведение не определено. Пример:

```
int f();
void (*p)() = reinterpret_cast<void(*)>()(f);
p(); // undefined
```

Изменение объекта `const`

Любая попытка изменить объект `const` приводит к неопределенному поведению. Это относится к `const` переменным, членам `const` объектов и членам класса, объявленным `const`. (Тем не менее, `mutable` член `const` объекта не `const`.)

Такая попытка может быть выполнена через `const_cast`:

```
const int x = 123;
const_cast<int&>(x) = 456;
std::cout << x << '\n';
```

Компилятор обычно встраивает значение объекта `const int`, поэтому возможно, что этот код компилирует и печатает `123`. Компиляторы также могут помещать значения `const` объектов в постоянную память, поэтому может возникнуть ошибка сегментации. В любом случае, поведение не определено, и программа может что-то сделать.

Следующая программа скрывает гораздо более тонкую ошибку:

```
#include <iostream>

class Foo* instance;

class Foo {
public:
    int get_x() const { return m_x; }
    void set_x(int x) { m_x = x; }
private:
    Foo(int x, Foo*& this_ref): m_x(x) {
        this_ref = this;
    }
};
```

```

    }
    int m_x;
    friend const Foo& getFoo();
};

const Foo& getFoo() {
    static const Foo foo(123, instance);
    return foo;
}

void do_evil(int x) {
    instance->set_x(x);
}

int main() {
    const Foo& foo = getFoo();
    do_evil(456);
    std::cout << foo.get_x() << '\n';
}

```

В этом коде `getFoo` создает одноэлемент типа `const Foo` а его член `m_x` инициализируется до 123 . Затем `do_evil` и значение `foo.m_x` по-видимому, изменено на 456. Что пошло не так?

Несмотря на свое имя, `do_evil` не делает ничего особо злого; все, что он делает, это вызвать сеттер через `Foo*` . Но этот указатель указывает на объект `const Foo` хотя `const_cast` не использовался. Этот указатель был получен с помощью конструктора `Foo` . Объект `const` не становится `const` до тех пор, пока его инициализация не будет завершена, поэтому в конструкторе `this` имеет тип `Foo*` , а не `const Foo*` .

Поэтому неопределенное поведение происходит, хотя в этой программе нет явно опасных конструкций.

Доступ к несуществующему члену через указатель на члена

При доступе к нестатическому члену объекта через указатель на член, если объект фактически не содержит элемент, обозначенный указателем, поведение не определено. (Такой указатель на член может быть получен через `static_cast` .)

```

struct Base { int x; };
struct Derived : Base { int y; };
int Derived::*pdy = &Derived::y;
int Base::*pby = static_cast<int Base::*>(pdy);

Base* b1 = new Derived;
b1->*pby = 42; // ok; sets y in Derived object to 42
Base* b2 = new Base;
b2->*pby = 42; // undefined; there is no y member in Base

```

Недопустимое преобразование на основе базы данных для указателей на членов

Когда `static_cast` используется для преобразования `TD::*` в `TB::*` , указанный указатель

должен принадлежать классу, который является базовым классом или производным классом в В противном случае поведение не определено. См. « [Производное преобразование базы](#)» для указателей на участников

Недопустимая арифметика указателя

Следующие виды использования арифметики указателя вызывают неопределенное поведение:

- Добавление или вычитание целого числа, если результат не принадлежит к одному и тому же объекту массива, как и операнд указателя. (Здесь элемент один за концом считается все еще принадлежащим массиву.)

```
int a[10];
int* p1 = &a[5];
int* p2 = p1 + 4; // ok; p2 points to a[9]
int* p3 = p1 + 5; // ok; p2 points to one past the end of a
int* p4 = p1 + 6; // UB
int* p5 = p1 - 5; // ok; p2 points to a[0]
int* p6 = p1 - 6; // UB
int* p7 = p3 - 5; // ok; p7 points to a[5]
```

- Вычитание двух указателей, если они не принадлежат к одному и тому же объекту массива. (Опять же, элемент один за концом считается принадлежащим массиву.)
Исключением является то, что два нулевых указателя могут быть вычтены, что дает 0.

```
int a[10];
int b[10];
int *p1 = &a[8], *p2 = &a[3];
int d1 = p1 - p2; // yields 5
int *p3 = p1 + 2; // ok; p3 points to one past the end of a
int d2 = p3 - p2; // yields 7
int *p4 = &b[0];
int d3 = p4 - p1; // UB
```

- Вычитание двух указателей, если результат переполняется `std::ptrdiff_t`.
- Любая арифметика указателя, где любой тип указателя операнда не соответствует динамическому типу объекта, на который указывает (игнорируя cv-qualification). Согласно стандарту, «в частности, указатель на базовый класс не может использоваться для арифметики указателя, когда массив содержит объекты производного типа типа».

```
struct Base { int x; };
struct Derived : Base { int y; };
Derived a[10];
Base* p1 = &a[1];           // ok
Base* p2 = p1 + 1;          // UB; p1 points to Derived
Base* p3 = p1 - 1;          // likewise
Base* p4 = &a[2];           // ok
```

```
auto p5 = p4 - p1;           // UB; p4 and p1 point to Derived
const Derived* p6 = &a[1];
const Derived* p7 = p6 + 1; // ok; cv-qualifiers don't matter
```

Смещение недопустимым числом позиций

Для встроенного оператора сдвига правый операнд должен быть неотрицательным и строго меньше ширины бита продвинутого левого операнда. В противном случае поведение не определено.

```
const int a = 42;
const int b = a << -1; // UB
const int c = a << 0;  // ok
const int d = a << 32; // UB if int is 32 bits or less
const int e = a >> 32; // also UB if int is 32 bits or less
const signed char f = 'x';
const int g = f << 10; // ok even if signed char is 10 bits or less;
                      // int must be at least 16 bits
```

Возврат из функции [[noreturn]]

C++ 11

Пример из стандарта, [dcl.attr.noreturn]:

```
[[ noreturn ]] void f() {
    throw "error"; // OK
}
[[ noreturn ]] void q(int i) { // behavior is undefined if called with an argument <= 0
    if (i > 0)
        throw "positive";
}
```

Уничтожение объекта, который уже был уничтожен

В этом примере деструктор явно вызывается для объекта, который впоследствии будет автоматически уничтожен.

```
struct S {
    ~S() { std::cout << "destroying S\n"; }
};
int main() {
    S s;
    s.~S();
} // UB: s destroyed a second time here
```

Аналогичная проблема возникает, когда `std::unique_ptr<T>` делается для указания на `T` с автоматической или статической продолжительностью хранения.

```
void f(std::unique_ptr<S> p);
```

```
int main() {
    S s;
    std::unique_ptr<S> p(&s);
    f(std::move(p)); // s destroyed upon return from f
}                  // UB: s destroyed
```

Другой способ уничтожить объект дважды - это сделать два `shared_ptr` которые управляют объектом, не разделяя права собственности друг на друга.

```
void f(std::shared_ptr<S> p1, std::shared_ptr<S> p2);
int main() {
    S* p = new S;
    // I want to pass the same object twice...
    std::shared_ptr<S> sp1(p);
    std::shared_ptr<S> sp2(p);
    f(sp1, sp2);
} // UB: both sp1 and sp2 will destroy s separately
// NB: this is correct:
// std::shared_ptr<S> sp(p);
// f(sp, sp);
```

Бесконечная рекурсия шаблона

Пример из стандарта, [temp.inst] / 17:

```
template<class T> class X {
    X<T>* p; // OK
    X<T*> a; // implicit generation of X<T> requires
             // the implicit instantiation of X<T*> which requires
             // the implicit instantiation of X<T**> which ...
};
```

Прочитайте Неопределенное поведение онлайн: <https://riptutorial.com/ru/cplusplus/topic/1812/неопределенное-поведение>

глава 64: Неопределенное поведение

замечания

Если поведение конструкции не указано, то стандарт помещает некоторые ограничения на поведение, но оставляет некоторую свободу для реализации, что *не* требуется для документирования того, что происходит в данной ситуации. Это контрастирует с [реализацией определяется поведением](#), в которых требуется осуществление документировать то, что происходит, и непредсказуемое поведение, в котором все может случиться.

Examples

Порядок инициализации globals через TU

Если внутри единицы перевода указывается порядок инициализации глобальных переменных, порядок инициализации между единицами перевода не указан.

Итак, программа со следующими файлами

- foo.cpp

```
#include <iostream>

int dummyFoo = ((std::cout << "foo"), 0);
```

- bar.cpp

```
#include <iostream>

int dummyBar = ((std::cout << "bar"), 0);
```

- main.cpp

```
int main() {}
```

может вырабатываться как результат:

```
foobar
```

или же

```
barfoo
```

Это может привести к *статическому порядку инициализации Fiasco* .

Значение перераспределения вне диапазона

Если облачное перечисление преобразуется в интегральный тип, который слишком мал, чтобы удерживать его значение, результирующее значение не указывается. Пример:

```
enum class E {
    X = 1,
    Y = 1000,
};
// assume 1000 does not fit into a char
char c1 = static_cast<char>(E::X); // c1 is 1
char c2 = static_cast<char>(E::Y); // c2 has an unspecified value
```

Кроме того, если целое число преобразуется в перечисление, а значение целочисленного значения выходит за пределы значений перечисления, результирующее значение не указывается. Пример:

```
enum Color {
    RED = 1,
    GREEN = 2,
    BLUE = 3,
};
Color c = static_cast<Color>(4);
```

Однако в следующем примере поведение *не* является неопределенным, так как исходное значение находится в пределах *диапазона* перечисления, хотя оно неравномерно для всех счетчиков:

```
enum Scale {
    ONE = 1,
    TWO = 2,
    FOUR = 4,
};
Scale s = static_cast<Scale>(3);
```

Здесь `s` будет иметь значение 3 и быть неравным с `ONE` , `TWO` и `FOUR` .

Статический литье из фиктивного значения `void *`

Если значение `void*` преобразуется в указатель на тип объекта, `T*` , но не правильно выровнено для `T` , результирующее значение указателя не указывается. Пример:

```
// Suppose that alignof(int) is 4
int x = 42;
void* p1 = &x;
// Do some pointer arithmetic...
void* p2 = static_cast<char*>(p1) + 2;
int* p3 = static_cast<int*>(p2);
```

Значение `p3` не определено, поскольку `p2` не может указывать на объект типа `int` ; его значение не является правильно выровненным адресом.

Результат некоторых преобразований `reinterpret_cast`

Результат `reinterpret_cast` из одного типа указателя функции в другой или один тип ссылочного типа в другой не указан. Пример:

```
int f();
auto fp = reinterpret_cast<int(*) (int)>(&f); // fp has unspecified value
```

C ++ 03

Результат `reinterpret_cast` от одного типа указателя объекта к другому, или один тип ссылки объекта к другому, не указан. Пример:

```
int x = 42;
char* p = reinterpret_cast<char*>(&x); // p has unspecified value
```

Однако с большинством компиляторов это было равнозначно

`static_cast<char*>(static_cast<void*>(&x))` поэтому полученный указатель `p` указывал на первый байт `x` . Это стало стандартным поведением в C ++ 11. Подробнее см. В разделе «[Преобразование чисел](#)» .

Результат некоторых сравнений указателей

Если два указателя сравниваются с использованием `<` , `>` , `<=` , или `>=` , результат не указывается в следующих случаях:

- Указатели указывают на разные массивы. (Объект без массива считается массивом размера 1.)

```
int x;
int y;
const bool b1 = &x < &y;           // unspecified
int a[10];
const bool b2 = &a[0] < &a[1];      // true
const bool b3 = &a[0] < &x;         // unspecified
const bool b4 = (a + 9) < (a + 10); // true
                                   // note: a+10 points past the end of the array
```

- Указатели указывают на один и тот же объект, но на участников с различным контролем доступа.

```
class A {
public:
    int x;
    int y;
    bool f1() { return &x < &y; } // true; x comes before y
```

```
bool f2() { return &x < &z; } // unspecified
private:
    int z;
};
```

Пространство, занятое ссылкой

Ссылка не является объектом, и, в отличие от объекта, не гарантируется, что он занимает несколько смежных байтов памяти. Стандарт оставляет его неопределенным, если ссылка требует какого-либо хранения вообще. Ряд особенностей языкового сговора, чтобы сделать невозможным портативное исследование любого хранилища, которое может занимать эта ссылка:

- Если `sizeof` применяется к ссылке, он возвращает размер ссылочного типа, тем самым не давая информации о том, занимает ли эта ссылка какое-либо хранилище.
- Массивы ссылок являются незаконными, поэтому невозможно определить адреса двух последовательных элементов гипотетической ссылки массивов, чтобы определить размер ссылки.
- Если адрес ссылки сделан, результатом будет адрес референта, поэтому мы не сможем получить указатель на саму ссылку.
- Если у класса есть ссылочный элемент, попытка извлечь адрес этого члена с помощью `offsetof` дает неопределенное поведение, поскольку такой класс не является стандартным классом макета.
- Если класс имеет ссылочный элемент, класс больше не является стандартным макетом, поэтому попытки доступа к любым данным, используемым для хранения ссылочных результатов, в неопределенном или неуказанном поведении.

На практике в некоторых случаях эталонная переменная может быть реализована аналогично переменной указателя и, следовательно, занимает ту же сумму хранения, что и указатель, в то время как в других случаях эта ссылка может вообще не занимать места, поскольку ее можно оптимизировать. Например, в:

```
void f() {
    int x;
    int& r = x;
    // do something with r
}
```

компилятор может просто рассматривать `r` как псевдоним для `x` и заменять все вхождения `r` в остальную часть функции `f` на `x` и не выделять какое-либо хранилище для хранения `r`.

Порядок оценки аргументов функции

Если функция имеет несколько аргументов, то не указано, в какой порядок они оцениваются. Следующий код может печатать `x = 1, y = 2` или `x = 2, y = 1` но не указано, что.

```
int f(int x, int y) {
    printf("x = %d, y = %d\n", x, y);
}
int get_val() {
    static int x = 0;
    return ++x;
}
int main() {
    f(get_val(), get_val());
}
```

C ++ 17

В C ++ 17 порядок оценки аргументов функции остается неопределенным.

Тем не менее, каждый аргумент функции полностью оценивается, и вызывающий объект гарантированно оценивается перед любыми аргументами функции.

```
struct from_int {
    from_int(int x) { std::cout << "from_int (" << x << ")\n"; }
};
int make_int(int x){ std::cout << "make_int (" << x << ")\n"; return x; }

void foo(from_int a, from_int b) {
}
void bar(from_int a, from_int b) {
}

auto which_func(bool b){
    std::cout << b?"foo":"bar" << "\n";
    return b?foo:bar;
}

int main(int argc, char const*const* argv) {
    which_func( true )( make_int(1), make_int(2) );
}
```

это должно печатать:

```
bar
make_int(1)
from_int(1)
make_int(2)
from_int(2)
```

или же

```
bar
make_int(2)
from_int(2)
make_int(1)
from_int(1)
```

он *не* может печатать `bar` после любого из `make` или `from` 's, и он не может печатать:

```
bar
make_int (2)
make_int (1)
from_int (2)
from_int (1)
```

или похожие. До `bar` C++ 17 после `make_int s` было законным, так как делало оба `make_int` перед тем, как делать какие-либо `from_int s`.

Перемещение из состояния большинства стандартных классов библиотеки

C++ 11

Все контейнеры стандартной библиотеки остаются в *правильном, но неуказанном* состоянии после перемещения из. Например, в следующем коде `v2` будет содержать `{1, 2, 3, 4}` после перемещения, но `v1` не гарантированно будет пустым.

```
int main() {
    std::vector<int> v1{1, 2, 3, 4};
    std::vector<int> v2 = std::move(v1);
}
```

Некоторые классы имеют точно определенное перемещенное состояние. Самый важный случай - это `std::unique_ptr<T>`, который после переноса будет иметь значение `null`.

Прочитайте Неопределенное поведение онлайн: <https://riptutorial.com/ru/cplusplus/topic/4939/неопределенное-поведение>

глава 65: Области применения

Examples

Простая область блока

Объем переменной в блоке `{ ... }`, начинается после объявления и заканчивается в конце блока. Если есть вложенный блок, внутренний блок может скрыть область видимости переменной, которая объявлена во внешнем блоке.

```
{
    int x = 100;
    //   ^
    //   Scope of `x` begins here
    //
} // <- Scope of `x` ends here
```

Если вложенный блок начинается во внешнем блоке, новая объявленная переменная с тем же именем, которая находится перед внешним классом, скрывает первый.

```
{
    int x = 100;

    {
        int x = 200;

        std::cout << x; // <- Output is 200
    }

    std::cout << x; // <- Output is 100
}
```

Глобальные переменные

Чтобы объявить один экземпляр переменной, доступной в разных исходных файлах, можно сделать ее в глобальной области с ключевым словом `extern`. Это ключевое слово говорит компилятору, что где-то в коде есть определение для этой переменной, поэтому его можно использовать повсюду, и все записи / чтения будут выполняться в одном месте памяти.

```
// File my_globals.h:

#ifndef __MY_GLOBALS_H__
#define __MY_GLOBALS_H__

extern int circle_radius; // Promise to the compiler that circle_radius
                          // will be defined somewhere

#endif
```

```
// File foo1.cpp:

#include "my_globals.h"

int circle_radius = 123; // Defining the extern variable
```

```
// File main.cpp:

#include "my_globals.h"
#include <iostream>

int main()
{
    std::cout << "The radius is: " << circle_radius << "\n";
    return 0;
}
```

Выход:

```
The radius is: 123
```

Прочитайте **Области применения** онлайн: <https://riptutorial.com/ru/cplusplus/topic/3453/области-применения>

глава 66: Обратный тип возврата

Синтаксис

- *function_name* ([*function_args*]) [*function_attributes*] [*function_qualifiers*] -> *trailing-return-type* [*require_clause*]

замечания

Вышеприведенный синтаксис показывает объявление полной функции с использованием конечного типа, где квадратные скобки указывают необязательную часть объявления функции (например, список аргументов, если функция no-arg).

Кроме того, синтаксис возвращаемого типа возврата запрещает определять тип класса, union или enum внутри возвращаемого возвращаемого типа (обратите внимание, что это также недопустимо в первом типе возврата). Помимо этого, типы могут быть записаны таким же образом после -> как и в других местах.

Examples

Избегайте квалифицировать вложенное имя типа

```
class ClassWithAReallyLongName {
public:
    class Iterator { /* ... */ };
    Iterator end();
};
```

Определение `end` элемента с возвращаемым типом возврата:

```
auto ClassWithAReallyLongName::end() -> Iterator { return Iterator(); }
```

Определение `end` элемента без возвращаемого типа возврата:

```
ClassWithAReallyLongName::Iterator ClassWithAReallyLongName::end() { return Iterator(); }
```

Возвращаемый тип возвращаемого типа просматривается в классе класса, в то время как ведущий тип возвращаемого значения просматривается в охватывающей области пространства имен и поэтому может потребовать «избыточной» квалификации.

Лямбда-выражения

Лямбда может иметь *только* возвращаемый тип возврата; синтаксис исходного типа

возврата не применим к lambdas. Обратите внимание, что во многих случаях нет необходимости указывать тип возврата для лямбды вообще.

```
struct Base {};  
struct Derived1 : Base {};  
struct Derived2 : Base {};  
auto lambda = [](bool b) -> Base* { if (b) return new Derived1; else return new Derived2; };  
// ill-formed: auto lambda = Base* [](bool b) { ... };
```

Прочитайте Обратный тип возврата онлайн: <https://riptutorial.com/ru/cplusplus/topic/4142/обратный-тип-возврата>

глава 67: Общие ошибки компиляции / компоновщика (GCC)

Examples

Ошибка: «*» не было объявлено в этой области**

Эта ошибка возникает, если используется неизвестный объект.

переменные

Не компиляция:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
    }

    std::cout << i << std::endl; // i is not in the scope of the main function

    return 0;
}
```

Fix:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
        std::cout << i << std::endl;
    }

    return 0;
}
```

функции

В большинстве случаев эта ошибка возникает, если нужный заголовок не включен (например, с помощью `std::cout` без `#include <iostream>`)

Не компиляция:

```
#include <iostream>

int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}

void doCompile()
{
    std::cout << "No!" << std::endl;
}
```

Fix:

```
#include <iostream>

void doCompile(); // forward declare the function

int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}

void doCompile()
{
    std::cout << "No!" << std::endl;
}
```

Или же:

```
#include <iostream>

void doCompile() // define the function before using it
{
    std::cout << "No!" << std::endl;
}

int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}
```

Примечание . Компилятор интерпретирует код сверху вниз (упрощение). Перед использованием все должно быть [объявлено \(или определено\)](#) .

неопределенная ссылка на `\*\*\*`

Эта ошибка компоновщика возникает, если компоновщик не может найти используемый символ. В большинстве случаев это происходит, если используемая библиотека не связана.

QMAKE:

```
LIBS += nameOfLib
```

CMake:

```
TARGET_LINK_LIBRARIES(target nameOfLib)
```

g ++ call:

```
g++ -o main main.cpp -Llibrary/dir -lnameOfLib
```

Также можно забыть компилировать и связывать все используемые .cpp файлы (.cpp определяет нужную функцию):

```
g++ -o binName main.o functionsModule.o
```

Неустраняемая ошибка: ***: Нет такого файла или каталога

Компилятор не может найти файл (исходный файл использует `#include "someFile.hpp"`).

QMAKE:

```
INCLUDEPATH += dir/Of/File
```

CMake:

```
include_directories(dir/Of/File)
```

g ++ call:

```
g++ -o main main.cpp -I dir/Of/File
```

Прочитайте [Общие ошибки компиляции / компоновщика \(GCC\) онлайн:](https://riptutorial.com/ru/cplusplus/topic/4256/общие-ошибки-компиляции---компоновщика-gcc-)

<https://riptutorial.com/ru/cplusplus/topic/4256/общие-ошибки-компиляции---компоновщика-gcc->

глава 68: Одно правило определения (ODR)

Examples

Умножённая функция

Важнейшим следствием правила Единого определения является то, что не-встроенные функции с внешней связью следует определять только один раз в программе, хотя они могут быть объявлены несколько раз. Поэтому такие функции не должны определяться в заголовках, поскольку заголовок может быть включен несколько раз из разных единиц перевода.

foo.h :

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
void foo() { std::cout << "foo"; }
void bar();
#endif
```

foo.cpp :

```
#include "foo.h"
void bar() { std::cout << "bar"; }
```

main.cpp :

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

В этой программе функция `foo` определена в заголовке `foo.h`, который включается дважды: один раз из `foo.cpp` и один раз из `main.cpp`. Поэтому каждая единица перевода содержит свое собственное определение `foo`. Обратите внимание, что включение `foo.h` в `foo.h` не мешает этому, поскольку `foo.cpp` и `main.cpp` оба *отдельно* включают `foo.h`. Наиболее вероятным результатом попытки создания этой программы является ошибка времени соединения, определяющая `foo` как многократно определенную.

Чтобы избежать таких ошибок, следует *объявлять* функции в заголовках и *определять* их в соответствующих файлах `.cpp`, за некоторыми исключениями (см. Другие примеры).

Встроенные функции

Функция, объявленная в `inline` может быть определена в нескольких единицах перевода, при условии, что все определения идентичны. Он также должен быть определен в каждой единицы перевода, в которой он используется. Поэтому встроенные функции *должны* быть определены в заголовках, и нет необходимости упоминать их в файле реализации.

Программа будет вести себя так, как если бы было одно определение функции.

foo.h :

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
inline void foo() { std::cout << "foo"; }
void bar();
#endif
```

foo.cpp :

```
#include "foo.h"
void bar() {
    // more complicated definition
}
```

main.cpp :

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

В этом примере более простая функция `foo` определена внутри строки в файле заголовка, тогда как более сложная функциональная `bar` не является встроенной и определена в файле реализации. Оба `foo.cpp` перевода `foo.cpp` и `main.cpp` содержат определения `foo` , но эта программа хорошо сформирована, поскольку `foo` является встроенным.

Функция, определенная в определении класса (которая может быть функцией-членом или функцией друга), *неявно* встроена. Поэтому, если класс определен в заголовке, функции-члены класса могут быть определены в определении класса, хотя определения могут быть включены в несколько единиц перевода:

```
// in foo.h
class Foo {
    void bar() { std::cout << "bar"; }
    void baz();
};

// in foo.cpp
```

```
void Foo::baz() {  
    // definition  
}
```

Функция `Foo::baz` определяется вне строки, поэтому она *не* является встроенной функцией и не должна определяться в заголовке.

Нарушение ODR с помощью разрешения перегрузки

Даже при идентичных токенах для встроенных функций ODR может быть нарушен, если поиск имен не относится к одному и тому же объекту. рассмотрим `func` в следующем:

- `header.h`

```
void overloaded(int);  
inline void func() { overloaded('*'); }
```

- `foo.cpp`

```
#include "header.h"  
  
void foo()  
{  
    func(); // `overloaded` refers to `void overloaded(int)`  
}
```

- `bar.cpp`

```
void overloaded(char); // can come from other include  
#include "header.h"  
  
void bar()  
{  
    func(); // `overloaded` refers to `void overloaded(char)`  
}
```

У нас есть нарушение ODR, поскольку `overloaded` относится к различным объектам в зависимости от единицы перевода.

Прочитайте [Одно правило определения \(ODR\) онлайн](https://riptutorial.com/ru/cplusplus/topic/4907/одно-правило-определения--odr-):

<https://riptutorial.com/ru/cplusplus/topic/4907/одно-правило-определения--odr->

глава 69: Операторы бит

замечания

Операции сдвига бит не переносимы во всех архитектурах процессоров, разные процессоры могут иметь разную ширину бит. Другими словами, если вы написали

```
int a = ~0;
int b = a << 1;
```

Это значение будет отличаться на 64-битной машине против 32-битной машины или от процессора на базе x86 до процессора на базе PIC.

Endian-ness не нужно принимать во внимание сами бит-операции, т. Е. Правый сдвиг (>>) сдвигает биты в сторону младшего значащего бита, а XOR будет выполнять исключение или по битам. Endian-ness нужно учитывать только сами данные, то есть, если конечная точка является проблемой для вашего приложения, это вызывает беспокойство, независимо от бит-мудрейных операций.

Examples

& - побитовое И

```
int a = 6;      // 0110b (0x06)
int b = 10;     // 1010b (0x0A)
int c = a & b;  // 0010b (0x02)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Выход

a = 6, b = 10, c = 2

Зачем

Немного мудрый AND работает на уровне бит и использует следующую логическую таблицу истинности:

```
TRUE  AND TRUE  = TRUE
TRUE  AND FALSE = FALSE
FALSE AND FALSE = FALSE
```

Когда двоичное значение для (a 0110) и бинарное значение b (1010) являются AND «эд вместе мы получим двоичное значение 0010 :

```
int a = 0 1 1 0
int b = 1 0 1 0 &
      -----
int c = 0 0 1 0
```

Бит мудрый И не изменяет значения исходных значений, если специально не назначено использование битового умножения оператора-оператора `&=` :

```
int a = 5;    // 0101b (0x05)
a &= 10;      // a = 0101b & 1010b
```

| - побитовое ИЛИ

```
int a = 5;      // 0101b (0x05)
int b = 12;     // 1100b (0x0C)
int c = a | b;  // 1101b (0x0D)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Выход

a = 5, b = 12, c = 13

Зачем

Бит мудрый OR работает на уровне бит и использует следующую таблицу логических истин:

```
true OR true = true
true OR false = true
false OR false = false
```

Когда двоичное значение для (a 0101) и бинарное значение b (1100) являются OR «эд вместе мы получим двоичное значение 1101 :

```
int a = 0 1 0 1
int b = 1 1 0 0 |
      -----
int c = 1 1 0 1
```

Бит мудрый ИЛИ не изменяет значения исходных значений, если только специально не назначено использование битового умножения оператора соединения `|=` :

```
int a = 5;    // 0101b (0x05)
a |= 12;      // a = 0101b | 1101b
```

^ - побитовое XOR (исключающее ИЛИ)

```
int a = 5;      // 0101b (0x05)
int b = 9;      // 1001b (0x09)
```

```
int c = a ^ b; // 1100b (0x0C)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Выход

a = 5, b = 9, c = 12

Зачем

Немного мудрый XOR (эксклюзивный или) работает на уровне бит и использует следующую логическую таблицу истинности:

```
true OR true = false
true OR false = true
false OR false = false
```

Обратите внимание, что при операции XOR `true OR true = false` где, как и при действии `true AND/OR true = true`, следовательно, исключительный характер операции XOR.

Используя это, когда двоичное значение для (a 0101) и бинарное значение b (1001) являются XOR - е изд «вместе мы получим двоичное значение 1100 :

```
int a = 0 1 0 1
int b = 1 0 0 1 ^
      -----
int c = 1 1 0 0
```

Бит мудрый XOR не изменяет значения исходных значений, если специально не назначено использование битового оператора присваивания соединения `^=` :

```
int a = 5; // 0101b (0x05)
a ^= 9; // a = 0101b ^ 1001b
```

Бит мудрый XOR может использоваться многими способами и часто используется в операциях бит-маски для шифрования и сжатия.

Примечание . Следующий пример часто показан в качестве примера приятного трюка. Но не следует использовать в производственном коде (для достижения такого же результата лучше использовать `std::swap()`).

Вы также можете использовать операцию XOR для замены двух переменных без временного:

```
int a = 42;
int b = 64;

// XOR swap
a ^= b;
b ^= a;
```

```
a ^= b;

std::cout << "a = " << a << ", b = " << b << "\n";
```

Чтобы произвести это, вам нужно добавить чек, чтобы убедиться, что он может быть использован.

```
void doXORSwap(int& a, int& b)
{
    // Need to add a check to make sure you are not swapping the same
    // variable with itself. Otherwise it will zero the value.
    if (&a != &b)
    {
        // XOR swap
        a ^= b;
        b ^= a;
        a ^= b;
    }
}
```

Таким образом, хотя это выглядит как хороший трюк в изоляции, он не полезен в реальном коде. xor не является базовой логической операцией, а комбинацией других: $a \oplus c = \sim(a \& c) \& (a \mid c)$

также в 2015 году переменные компиляторов могут быть назначены как двоичные:

```
int cn=0b0111;
```

~ - побитовое NOT (унарное дополнение)

```
unsigned char a = 234; // 1110 1010b (0xEA)
unsigned char b = ~a; // 0001 0101b (0x15)

std::cout << "a = " << static_cast<int>(a) <<
    ", b = " << static_cast<int>(b) << std::endl;
```

Выход

```
a = 234, b = 21
```

Зачем

Немного мудрый NOT (унарное дополнение) работает на уровне бит и просто переворачивает каждый бит. Если оно равно 1, оно изменилось на 0, если оно равно 0, оно изменилось на 1. Бит wise NOT имеет тот же эффект, что и XOR'ing значение против максимального значения для определенного типа:

```
unsigned char a = 234; // 1110 1010b (0xEA)
unsigned char b = ~a; // 0001 0101b (0x15)
unsigned char c = a ^ ~0;
```

Бит мудрый NOT также может быть удобным способом проверки максимального значения для определенного интегрального типа:

```
unsigned int i = ~0;
unsigned char c = ~0;

std::cout << "max uint = " << i << std::endl <<
    "max uchar = " << static_cast<short>(c) << std::endl;
```

Бит мудрый NOT не изменяет значение исходного значения и не имеет составного оператора присваивания, поэтому вы не можете сделать `a ~= 10` например.

Бит мудрый NOT (~) не следует путать с логическим NOT (!); где бит мудрый НЕ будет переворачивать каждый бит, логическое НЕ будет использовать все значение для его работы, другими словами (!1) != (~1)

<< - сдвиг влево

```
int a = 1;          // 0001b
int b = a << 1;     // 0010b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Выход

a = 1, b = 2

Зачем

Левый бит мудрый сдвиг сдвинет биты значения левой руки (a) числа, указанного справа (1), по существу, заполняя младшие значащие биты 0, поэтому сдвиг значения 5 (двоичный 0000 0101) влево 4 раза (например, `5 << 4`) даст значение 80 (двоичный 0101 0000). Вы можете заметить, что смещение значения влево 1 раз также совпадает с умножением значения на 2, например:

```
int a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a <<= 1;
}

a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a *= 2;
}
```

Но следует отметить, что операция сдвига влево сдвигает *все* биты влево, включая бит знака, например:

```
int a = 2147483647; // 0111 1111 1111 1111 1111 1111 1111 1111
int b = a << 1;      // 1111 1111 1111 1111 1111 1111 1111 1110

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Возможный выход: a = 2147483647, b = -2

В то время как некоторые компиляторы дают ожидаемые результаты, следует отметить, что если вы оставили сдвиг значного числа, чтобы повлиять на бит знака, результат не **определен** . Также **не определено**, если количество бит, которое вы хотите сдвинуть, является отрицательным числом или больше, чем количество бит, которое может удерживать тип слева, например:

```
int a = 1;
int b = a << -1; // undefined behavior
char c = a << 20; // undefined behavior
```

Бит мудрый сдвиг влево не изменяет значения исходных значений, если специально не назначено использование битового оператора соединения соединения <<= :

```
int a = 5; // 0101b
a <<= 1;   // a = a << 1;
```

>> - правая смена

```
int a = 2;      // 0010b
int b = a >> 1; // 0001b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Выход

a = 2, b = 1

Зачем

Правильный битовый сдвиг сдвигает биты значения левой руки (a) числа, указанного справа (1); следует отметить, что, хотя операция правого сдвига является стандартной, то, что происходит с битами правого сдвига по *значению отрицательного* числа, является *определенной реализацией* и, следовательно, не может быть гарантировано быть переносимым, например:

```
int a = -2;
int b = a >> 1; // the value of b will be depend on the compiler
```

Также не определено, если количество бит, которое вы хотите сдвинуть, является отрицательным числом, например:

```
int a = 1;
int b = a >> -1; // undefined behavior
```

Правильный сдвиг по битам не изменяет значения исходных значений, если специально не назначено использование битового оператора присваивания соединения `>>=` :

```
int a = 2; // 0010b
a >>= 1; // a = a >> 1;
```

Прочитайте Операторы бит онлайн: <https://riptutorial.com/ru/cplusplus/topic/2572/операторы-бит>

глава 70: оптимизация

Вступление

При компиляции компилятор часто изменяет программу для повышения производительности. Это разрешено по [правилу as-if](#), которое допускает любые и все преобразования, которые не изменяют наблюдаемое поведение.

Examples

Встроенное расширение / встраивание

Встроенное расширение (также известное как `inlining`) - это оптимизация компилятора, которая заменяет вызов функции с телом этой функции. Это экономит служебные данные вызова функции, но ценой пространства, так как функция может дублироваться несколько раз.

```
// source:

int process(int value)
{
    return 2 * value;
}

int foo(int a)
{
    return process(a);
}

// program, after inlining:

int foo(int a)
{
    return 2 * a; // the body of process() is copied into foo()
}
```

Вложение обычно выполняется для небольших функций, где служебные данные вызова функции значительны по сравнению с размером тела функции.

Пустая оптимизация базы

Размер любого объекта или субобъекта участника должен быть не менее 1, даже если тип представляет собой пустой тип `class` (то есть `class` или `struct`, у которых нет нестатических членов данных), чтобы иметь возможность гарантировать, что адреса отдельных объектов одного и того же типа всегда различны.

Однако подобъекты базового `class` не так ограничены и могут быть полностью

оптимизированы из макета объекта:

```
#include <cassert>

struct Base {}; // empty class

struct Derived1 : Base {
    int i;
};

int main() {
    // the size of any object of empty class type is at least 1
    assert(sizeof(Base) == 1);

    // empty base optimization applies
    assert(sizeof(Derived1) == sizeof(int));
}
```

Пустая оптимизация базы обычно используется стандартными библиотечными классами, поддерживающими распределители (`std::vector` , `std::function` , `std::shared_ptr` и т. Д.), Чтобы избежать использования какого-либо дополнительного хранилища для его элемента-распределителя, если распределитель не имеет состояния. Это достигается за счет хранения одного из требуемых элементов данных (например, указателя `begin` , `end` или `capacity` для `vector`).

Ссылка: [cppreference](#)

Прочитайте оптимизация онлайн: <https://riptutorial.com/ru/cplusplus/topic/9767/оптимизация>

глава 71: Оптимизация в C ++

Examples

Пустая оптимизация базового класса

Объект не может занимать менее 1 байт, так как тогда члены массива этого типа будут иметь один и тот же адрес. Таким образом, `sizeof(T) >= 1` всегда выполняется. Верно также, что производный класс не может быть меньше *любого* из его базовых классов. Однако, когда базовый класс пуст, его размер необязательно добавляется к производному классу:

```
class Base {};  
  
class Derived : public Base  
{  
public:  
    int i;  
};
```

В этом случае не требуется выделять байт для `Base Derived` чтобы иметь отдельный адрес для каждого типа объекта. Если выполняется пустая оптимизация базового класса (и не требуется заполнить), тогда `sizeof(Derived) == sizeof(int)`, то есть никакого дополнительного распределения для пустой базы не выполняется. Это возможно и с несколькими базовыми классами (в C ++ несколько баз не могут иметь один и тот же тип, поэтому из этого не возникает никаких проблем).

Обратите внимание, что это может быть выполнено только в том случае, если первый член `Derived` отличается по типу от любого из базовых классов. Сюда входят любые прямые или косвенные общие основания. Если это тот же тип, что и один из оснований (или общая база), требуется хотя бы выделение одного байта, чтобы гарантировать, что два разных объекта одного и того же типа не имеют одинакового адреса.

Введение в производительность

C и C ++ хорошо известны как высокопроизводительные языки - во многом из-за большого количества настроек кода, позволяя пользователю указывать производительность по выбору структуры.

При оптимизации важно провести оценку соответствующего кода и полностью понять, как будет использоваться код.

Общие ошибки оптимизации включают:

- **Преждевременная оптимизация:** сложный код может *ухудшиться* после оптимизации, тратя время и силы. Первым приоритетом должно быть написание

правильного и поддерживаемого кода, а не оптимизированный код.

- **Оптимизация для неправильного варианта использования:** добавление накладных расходов для 1% может не стоить замедления для других 99%
- **Микро-оптимизация:** компиляторы делают это очень эффективно, и микро-оптимизация может даже повредить компиляторам возможность дальнейшей оптимизации кода

Типичными целями оптимизации являются:

- Делать меньше работы
- Использовать более эффективные алгоритмы / структуры
- Чтобы лучше использовать аппаратное обеспечение

Оптимизированный код может иметь отрицательные побочные эффекты, в том числе:

- Использование более высокой памяти
- Сложный код - трудно читать или поддерживать
- Компромисс API и дизайн кода

Оптимизация путем выполнения меньшего количества кода

Самый простой подход к оптимизации - это сокращение кода. Этот подход обычно дает фиксированное ускорение без изменения временной сложности кода.

Несмотря на то, что этот подход дает вам четкое ускорение, это только даст заметные улучшения, когда код называется много.

Удаление бесполезного кода

```
void func(const A *a); // Some random function

// useless memory allocation + deallocation for the instance
auto a1 = std::make_unique<A>();
func(a1.get());

// making use of a stack object prevents
auto a2 = A{};
func(&a2);
```

C ++ 14

C ++ 14 компиляторам разрешено оптимизировать этот код, чтобы удалить выделение и сопоставить освобождение.

Выполнение кода только один раз

```
std::map<std::string, std::unique_ptr<A>> lookup;
```

```
// Slow insertion/lookup
// Within this function, we will traverse twice through the map lookup an element
// and even a thirth time when it wasn't in
const A *lazyLookupSlow(const std::string &key) {
    if (lookup.find(key) != lookup.cend())
        lookup.emplace_back(key, std::make_unique<A>());
    return lookup[key].get();
}

// Within this function, we will have the same noticeable effect as the slow variant while
// going at double speed as we only traverse once through the code
const A *lazyLookupSlow(const std::string &key) {
    auto &value = lookup[key];
    if (!value)
        value = std::make_unique<A>();
    return value.get();
}
```

Аналогичный подход к этой оптимизации может быть использован для реализации стабильной версии `unique`

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // As insert returns if the insertion was successful, we can deduce if the element was
        // already in or not
        // This prevents an insertion, which will traverse through the map for every unique
        // element
        // As a result we can almost gain 50% if v would not contain any duplicates
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Предотвращение бесполезного перераспределения и копирования / перемещения

В предыдущем примере мы уже предотвратили поиск в `std::set`, однако `std::vector` все еще содержит растущий алгоритм, в котором он должен будет перераспределить его хранилище. Этого можно предотвратить, предварительно зарезервировав нужный размер.

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    // By reserving 'result', we can ensure that no copying or moving will be done in the
    // vector
    // as it will have capacity for the maximum number of elements we will be inserting
    // If we make the assumption that no allocation occurs for size zero
    // and allocating a large block of memory takes the same time as a small block of memory
    // this will never slow down the program
    // Side note: Compilers can even predict this and remove the checks the growing from the
    // generated code
    result.reserve(v.size());
}
```

```

std::set<std::string> checkUnique;
for (const auto &s : v) {
    // See example above
    if (checkUnique.insert(s).second)
        result.push_back(s);
}
return result;
}

```

Использование эффективных контейнеров

Оптимизация с использованием правильных структур данных в нужное время может изменить временную сложность кода.

```

// This variant of stableUnique contains a complexity of N log(N)
// N > number of elements in v
// log(N) > insert complexity of std::set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}

```

Используя контейнер, который использует другую реализацию для хранения своих элементов (хэш-контейнер вместо дерева), мы можем преобразовать нашу реализацию в сложность N . В качестве побочного эффекта мы будем называть оператор сравнения для `std::string` `less`, поскольку он нужно только вызывать, когда вставленная строка должна заканчиваться в том же ведре.

```

// This variant of stableUnique contains a complexity of N
// N > number of elements in v
// 1 > insert complexity of std::unordered_set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::unordered_set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}

```

Оптимизация малых объектов

Оптимизация небольших объектов - это метод, который используется в структурах данных низкого уровня, например `std::string` (иногда называемый Short / Small String Optimization). Он предназначен для использования пространства стека в качестве буфера вместо

некоторой выделенной памяти, если контент достаточно мал, чтобы влезть в зарезервированное пространство.

Добавляя дополнительные накладные расходы памяти и дополнительные вычисления, он пытается предотвратить дорогостоящее распределение кучи. Преимущества этого метода зависят от использования и могут даже повредить производительность при неправильном использовании.

пример

Очень наивный способ реализации строки с этой оптимизацией:

```
#include <cstring>

class string final
{
    constexpr static auto SMALL_BUFFER_SIZE = 16;

    bool _isAllocated{false};           ///< Remember if we allocated memory
    char *_buffer{nullptr};              ///< Pointer to the buffer we are using
    char _smallBuffer[SMALL_BUFFER_SIZE]= {'\0'};  ///< Stack space used for SMALL OBJECT
    OPTIMIZATION

public:
    ~string()
    {
        if (_isAllocated)
            delete [] _buffer;
    }

    explicit string(const char *cStyleString)
    {
        auto stringSize = std::strlen(cStyleString);
        _isAllocated = (stringSize > SMALL_BUFFER_SIZE);
        if (_isAllocated)
            _buffer = new char[stringSize];
        else
            _buffer = &_amp_smallBuffer[0];
        std::strcpy(_buffer, &cStyleString[0]);
    }

    string(string &&rhs)
        : _isAllocated(rhs._isAllocated)
        , _buffer(rhs._buffer)
        , _smallBuffer(rhs._smallBuffer)  ///< Not needed if allocated
    {
        if (_isAllocated)
        {
            // Prevent double deletion of the memory
            rhs._buffer = nullptr;
        }
        else
        {
            // Copy over data
            std::strcpy(_smallBuffer, rhs._smallBuffer);
        }
    }
};
```

```
        _buffer = &_smallBuffer[0];
    }
}
// Other methods, including other constructors, copy constructor,
// assignment operators have been omitted for readability
};
```

Как вы можете видеть в приведенном выше коде, добавлена дополнительная сложность, чтобы предотвратить некоторые `new` операции и `delete`. Кроме того, класс имеет большой объем памяти, который может не использоваться, за исключением нескольких случаев.

Часто он пытается кодировать значение `bool _isAllocated` в указателе `_buffer` с помощью [бит-манипуляции](#) для уменьшения размера одного экземпляра (бит 64-разрядной версии: может уменьшить размер на 8 байт). Оптимизация, которая возможна только тогда, когда известно, какие правила выравнивания платформы.

Когда использовать?

Поскольку эта оптимизация добавляет много сложности, не рекомендуется использовать эту оптимизацию для каждого отдельного класса. Это часто встречается в часто используемых структурах данных низкого уровня. В обычных реализациях `standard library` C++ 11 можно найти `std::basic_string<>` использования в `std::basic_string<>` и `std::function<>`.

Поскольку эта оптимизация только предотвращает выделение памяти, когда хранящиеся данные меньше, чем буфер, это даст только преимущества, если класс часто используется с небольшими данными.

Конечным недостатком этой оптимизации является то, что при перемещении буфера требуется дополнительное усилие, что делает операцию перемещения более дорогой, чем когда буфер не будет использоваться. Это особенно верно, когда буфер содержит не-POD-тип.

Прочитайте [Оптимизация в C++ онлайн: https://riptutorial.com/ru/cplusplus/topic/4474/оптимизация-в-c-plusplus](https://riptutorial.com/ru/cplusplus/topic/4474/оптимизация-в-c-plusplus)

глава 72: Основные ключевые слова

Examples

ИНТ

Обозначает целочисленный тип со знаком с «естественным размером, предложенным архитектурой среды исполнения», диапазон которого включает, по меньшей мере, от -32767 до +32767 включительно.

```
int x = 2;
int y = 3;
int z = x + y;
```

Может комбинироваться с `unsigned`, `short`, `long` и `long long` (qv) для получения других целочисленных типов.

BOOL

Целочисленный тип, значение которого может быть `true` или `false`.

```
bool is_even(int x) {
    return x%2 == 0;
}
const bool b = is_even(47); // false
```

голец

Целочисленный тип, который «достаточно велик для хранения любого элемента базового набора символов реализации». Определяется реализация, независимо от того, подписан ли `char` (и имеет диапазон от -127 до +127 включительно) или без знака (и имеет диапазон от 0 до 255 включительно).

```
const char zero = '0';
const char one = zero + 1;
const char newline = '\n';
std::cout << one << newline; // prints 1 followed by a newline
```

char16_t

C ++ 11

Беззнаковый целочисленный тип с таким же размером и выравниванием, что и `uint_least16_t`, поэтому он достаточно велик, чтобы содержать код UTF-16.


```
const char16_t message[] = u"你好\n";           // Chinese for "hello, world\n"
std::cout << sizeof(message)/sizeof(char16_t) << "\n"; // prints 7
```

char32_t

C++ 11

Беззнаковый целочисленный тип с таким же размером и выравниванием, что и `uint_least32_t`, поэтому он достаточно велик, чтобы содержать код UTF-32.

```
const char32_t full_house[] = U"𐀀𐀁𐀂𐀃";           // non-BMP characters
std::cout << sizeof(full_house)/sizeof(char32_t) << "\n"; // prints 6
```

поплавков

Тип с плавающей точкой. Имеет самый узкий диапазон из трех типов с плавающей точкой в C++.

```
float area(float radius) {
    const float pi = 3.14159f;
    return pi*radius*radius;
}
```

двойной

Тип с плавающей точкой. Его диапазон включает в себя `float`. В сочетании с `long` означает `long double` тип с плавающей запятой, диапазон которого включает `double`.

```
double area(double radius) {
    const double pi = 3.141592653589793;
    return pi*radius*radius;
}
```

долго

Обозначает целочисленный целочисленный тип, который, по меньшей мере, длинный `int`, и диапазон которого включает в себя по меньшей мере от -2147483647 до +2147483647, включительно (то есть - $(2^{31} - 1)$ до + $(2^{31} - 1)$). Этот тип также может быть записан как `long int`.

```
const long approx_seconds_per_year = 60L*60L*24L*365L;
```

Комбинация `long double` обозначает тип с плавающей точкой, который имеет самый широкий диапазон из трех типов с плавающей запятой.

```
long double area(long double radius) {
    const long double pi = 3.1415926535897932385L;
```

```
    return pi*radius*radius;
}
```

С ++ 11

Когда `long` спецификатор происходит дважды, как в `long long`, он обозначает целочисленный тип со знаком, который, по меньшей мере, `long`, и диапазон которого включает в себя по меньшей мере -9223372036854775807 до +9223372036854775807 включительно (то есть - $(2^{63} - 1)$ до + $(2^{63} - 1)$).

```
// support files up to 2 TiB
const long long max_file_size = 2LL << 40;
```

короткая

Обозначает целочисленный тип со знаком, который, по меньшей мере, длиннее `char`, и диапазон которого включает, по меньшей мере, от -32767 до +32767 включительно. Этот тип также может быть записан как `short int`.

```
// (during the last year)
short hours_worked(short days_worked) {
    return 8*days_worked;
}
```

недействительным

Неполный тип; невозможно, чтобы объект имел тип `void`, и нет массивов `void` или ссылок на `void`. Он используется как возвращаемый тип функций, которые ничего не возвращают.

Более того, функция может быть избыточно объявлена с помощью одного параметра типа `void`; это эквивалентно объявлению функции без параметров (например, `int main()` и `int main(void)` объявляют одну и ту же функцию). Этот синтаксис разрешен для совместимости с C (где объявления функций имеют другое значение, чем в C ++).

Тип `void*` ("pointer to `void`") обладает тем свойством, что любой указатель объекта может быть преобразован в него и обратно и в результате будет иметь тот же указатель. Эта функция делает тип `void*` подходящим для определенных типов (типа небезопасных) типов стирающих интерфейсов, например, для общих контекстов в API-интерфейсах C-стиля (например, `qsort`, `pthread_create`).

Любое выражение может быть преобразовано в выражение типа `void`; это называется *выражением отбрасываемого значения*:

```
static_cast<void>(std::printf("Hello, %s!\n", name)); // discard return value
```

Это может быть полезно явно указывать на то, что значение выражения не представляет

интереса и что выражение должно оцениваться только для его побочных эффектов.

wchar_t

Целочисленный тип, достаточно большой для представления всех символов самого большого поддерживаемого расширенного набора символов, также известного как широкосимвольный набор. (Нельзя использовать предположение, что `wchar_t` использует какую-либо конкретную кодировку, такую как UTF-16.)

Он обычно используется, когда вам нужно хранить символы по ASCII 255, так как он имеет больший размер, чем символ типа `char`.

```
const wchar_t message_ahmaric[] = L"ሰላም ልዑል\n"; //Ahmaric for "hello, world\n"
const wchar_t message_chinese[] = L"你好\n"; // Chinese for "hello, world\n"
const wchar_t message_hebrew[] = L"שלום עולם\n"; //Hebrew for "hello, world\n"
const wchar_t message_russian[] = L"Привет мир\n"; //Russian for "hello, world\n"
const wchar_t message_tamil[] = L"ஹலோ உலகம்\n"; //Tamil for "hello, world\n"
```

Прочитайте Основные ключевые слова онлайн: <https://riptutorial.com/ru/cplusplus/topic/7839/основные-ключевые-слова>

глава 73: Пакеты параметров

Examples

Шаблон с пакетом параметров

```
template<class ... Types> struct Tuple {};
```

Пакет параметров - это параметр шаблона, принимающий ноль или более аргументов шаблона. Если шаблон имеет хотя бы один пакет параметров, это *вариационный шаблон*.

Расширение пакета параметров

Паттерн `parameter_pack ...` раскрывается в список разделенных запятыми подстановок `parameter_pack` с каждым из его параметров

```
template<class T> // Base of recursion
void variadic_printer(T last_argument) {
    std::cout << last_argument;
}

template<class T, class ...Args>
void variadic_printer(T first_argument, Args... other_arguments) {
    std::cout << first_argument << "\n";
    variadic_printer(other_arguments...); // Parameter pack expansion
}
```

Вышеописанный код с помощью `variadic_printer(1, 2, 3, "hello");` печать

```
1
2
3
hello
```

Прочитайте Пакеты параметров онлайн: <https://riptutorial.com/ru/cplusplus/topic/7668/пакеты-параметров>

глава 74: Параллельность с OpenMP

Вступление

В этом разделе рассматриваются основы параллелизма в C++ с использованием OpenMP. OpenMP документируется более подробно в [теге OpenMP](#).

Параллелизм или параллелизм подразумевают выполнение кода одновременно.

замечания

OpenMP не требует каких-либо специальных заголовков или библиотек, поскольку это встроенная функция компилятора. Однако, если вы используете какие-либо функции OpenMP API, такие как `omp_get_thread_num()`, вам нужно будет `omp.h` и его библиотеку.

Операторы `pragma` OpenMP игнорируются, когда опция OpenMP не включена во время компиляции. Вы можете обратиться к опции компилятора в руководстве вашего компилятора.

- GCC использует `-fopenmp`
- Кланг использует `-fopenmp`
- MSVC использует `/openmp`

Examples

OpenMP: параллельные разделы

Этот пример иллюстрирует основы выполнения разделов кода параллельно.

Поскольку OpenMP является встроенной функцией компилятора, он работает с любыми поддерживаемыми компиляторами без включения каких-либо библиотек. Возможно, вы захотите включить `omp.h` если вы хотите использовать какие-либо функции OpenMP API.

Образец кода

```
std::cout << "begin ";
// This pragma statement hints the compiler that the
// contents within the { } are to be executed in as
// parallel sections using openMP, the compiler will
// generate this chunk of code for parallel execution
#pragma omp parallel sections
{
    // This pragma statement hints the compiler that
    // this is a section that can be executed in parallel
    // with other section, a single section will be executed
```

```

//    by a single thread.
//    Note that it is "section" as opposed to "sections" above
#pragma omp section
{
    std::cout << "hello " << std::endl;
    /** Do something */
}
#pragma omp section
{
    std::cout << "world " << std::endl;
    /** Do something */
}
}
//    This line will not be executed until all the
//    sections defined above terminates
std::cout << "end" << std::endl;

```

Выходы

Этот пример дает 2 возможных выхода и зависит от операционной системы и оборудования. Результат также иллюстрирует проблему **состояния гонки**, которая возникла бы из такой реализации.

ВЫХОД А	ВЫХОД В
начать привет мир	начать мир привет

OpenMP: параллельные разделы

В этом примере показано, как выполнять куски кода параллельно

```

std::cout << "begin ";
//    Start of parallel sections
#pragma omp parallel sections
{
    //    Execute these sections in parallel
    #pragma omp section
    {
        ... do something ...
        std::cout << "hello ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "world ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "forever ";
    }
}
//    end of parallel sections
std::cout << "end";

```

Выход

- начать привет мир навсегда конец
- начать мир привет навсегда конец
- начать привет навсегда
- начинать навсегда hello world end

Поскольку порядок выполнения не гарантируется, вы можете наблюдать за любым из вышеперечисленных результатов.

OpenMP: Параллельный для цикла

В этом примере показано, как разделить цикл на равные части и выполнить их параллельно.

```
// Splits element vector into element.size() / Thread Qty
// and allocate that range for each thread.
#pragma omp parallel for
for (size_t i = 0; i < element.size(); ++i)
    element[i] = ...

// Example Allocation (100 element per thread)
// Thread 1 : 0 ~ 99
// Thread 2 : 100 ~ 199
// Thread 2 : 200 ~ 299
// ...

// Continue process
// Only when all threads completed their allocated
// loop job
...
```

* Пожалуйста, проявляйте особую осторожность, чтобы не изменять размер вектора, который используется параллельно для циклов, поскольку **выделенные индексы диапазона не обновляются автоматически**.

OpenMP: параллельная сборка / сокращение

Этот пример иллюстрирует концепцию выполнения сокращения или сбора с использованием `std::vector` и OpenMP.

Предположим, у нас есть сценарий, когда мы хотим, чтобы несколько потоков помогли нам создать кучу материала, `int` используется здесь для простоты и может быть заменен другими типами данных.

Это особенно полезно, когда вам нужно объединить результаты с ведомыми устройствами, чтобы избежать сбоев в сегментировании или нарушения доступа к памяти, и не хотите использовать библиотеки или пользовательские библиотеки контейнеров синхронизации.

```

//    The Master vector
//    We want a vector of results gathered from slave threads
std::vector<int> Master;

//    Hint the compiler to parallelize this { } of code
//    with all available threads (usually the same as logical processor qty)
#pragma omp parallel
{
    //    In this area, you can write any code you want for each
    //    slave thread, in this case a vector to hold each of their results
    //    We don't have to worry about how many threads were spawn or if we need
    //    to repeat this declaration or not.
    std::vector<int> Slave;

    //    Tell the compiler to use all threads allocated for this parallel region
    //    to perform this loop in parts. Actual load appx = 1000000 / Thread Qty
    //    The nowait keyword tells the compiler that the slave threads don't
    //    have to wait for all other slaves to finish this for loop job
    #pragma omp for nowait
    for (size_t i = 0; i < 1000000; ++i)
    {
        /* Do something */
        ....
        Slave.push_back(...);
    }

    //    Slaves that finished their part of the job
    //    will perform this thread by thread one at a time
    //    critical section ensures that only 0 or 1 thread performs
    //    the { } at any time
    #pragma omp critical
    {
        //    Merge slave into master
        //    use move iterators instead, avoid copy unless
        //    you want to use it for something else after this section
        Master.insert(Master.end(),
                     std::make_move_iterator(Slave.begin()),
                     std::make_move_iterator(Slave.end()));
    }
}

//    Have fun with Master vector
...

```

Прочитайте Параллельность с OpenMP онлайн: <https://riptutorial.com/ru/cplusplus/topic/8222/параллельность-с-openmp>

глава 75: Перегрузка оператора

Вступление

В C++ можно определить операторы, такие как `+` и `->` для пользовательских типов. Например, заголовок `<string>` определяет оператор `+` для конкатенации строк. Это делается путем определения *операторной функции* с использованием **ключевого слова** `operator`.

замечания

Операторы для встроенных типов не могут быть изменены, операторы могут быть перегружены только для пользовательских типов. То есть, по крайней мере, один из операндов должен быть определенного пользователем типа.

Следующие операторы *не могут* быть перегружены:

- Доступ к члену или «точка» `.`
- Указатель на оператор доступа к члену `.*`
- Оператор разрешения области, `::`
- Тройной условный оператор, `?:`
- `dynamic_cast`, `static_cast`, `reinterpret_cast`, `const_cast`, `typeid`, `sizeof`, `alignof` **и** `noexcept`
- Директивы предварительной обработки, `#` и `##`, которые выполняются перед любой информацией типа, доступны.

Есть некоторые операторы, которых вы **не** должны (99,98% времени) перегружать:

- `&&` и `||` (вместо этого следует использовать неявное преобразование в `bool`)
- `,`
- Адрес-оператор (унарный `&`)

Зачем? Потому что они перегружают операторы, которых не может ожидать другой программист, что приводит к поведению, чем ожидалось.

Например, пользователь определил `&&` и `||` перегрузки этих операторов **теряют свою оценку короткого замыкания** и **теряют свое особое свойства секвенирования (C++ 17)**, вопрос секвенирования также относится и к `,` оператор перегрузка.

Examples

Арифметические операторы

Вы можете перегрузить все основные арифметические операторы:

- `+ И +=`
- `- И -=`
- `* И *=`
- `/ И /=`
- `& and &=`
- `| И |=`
- `^ И ^=`
- `>> И >>=`
- `<< И <<=`

Перегрузка для всех операторов одинакова. *Прокрутите вниз для объяснения*

Перегрузка вне `class / struct` :

```
//operator+ should be implemented in terms of operator+=
T operator+(T lhs, const T& rhs)
{
    lhs += rhs;
    return lhs;
}

T& operator+=(T& lhs, const T& rhs)
{
    //Perform addition
    return lhs;
}
```

Перегрузка внутри `class / struct` :

```
//operator+ should be implemented in terms of operator+=
T operator+(const T& rhs)
{
    *this += rhs;
    return *this;
}

T& operator+=(const T& rhs)
{
    //Perform addition
    return *this;
}
```

Примечание: `operator+` должен возвращаться неконстантным значением, поскольку возврат ссылки не имеет смысла (он возвращает *новый* объект) и не возвращает значение `const` (вы, как правило, не должны возвращать `const`). Первый аргумент передается по значению, почему? Так как

1. Вы не можете изменить исходный объект (`Object foobar = foo + bar;` не следует изменять `foo` конце концов, это не имеет смысла)
2. Вы не можете сделать его `const` , потому что вам нужно будет иметь возможность

изменять объект (потому что `operator+` реализован в терминах `operator+=` , который изменяет объект)

Передача `const&` будет вариантом, но тогда вам придется сделать временную копию переданного объекта. Переходя по значению, компилятор делает это за вас.

`operator+=` возвращает ссылку на себя, потому что тогда можно связать их (не используйте одну и ту же переменную, хотя это будет неопределенное поведение из-за точек последовательности).

Первый аргумент - это ссылка (мы хотим ее изменить), но не `const` , потому что тогда вы не сможете ее изменить. Второй аргумент не должен быть изменен, поэтому аргумент производительности передается командой `const&` (передача по заданию `const` быстрее, чем по значению).

Унарные операторы

Вы можете перегрузить два унарных оператора:

- `++foo` И `foo++`
- `--foo` И `foo--`

Перегрузка одинакова для обоих типов (`++` и `--`). *Прокрутите вниз для объяснения*

Перегрузка вне `class` / `struct` :

```
//Prefix operator ++foo
T& operator++(T& lhs)
{
    //Perform addition
    return lhs;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(T& lhs, int)
{
    T t(lhs);
    ++lhs;
    return t;
}
```

Перегрузка внутри `class` / `struct` :

```
//Prefix operator ++foo
T& operator++()
{
    //Perform addition
    return *this;
}
```

```
//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(int)
{
    T t(*this);
    ++(*this);
    return t;
}
```

Примечание. Оператор префикса возвращает ссылку на себя, чтобы продолжить работу над ним. Первый аргумент - это ссылка, поскольку оператор префикса меняет объект, это также причина, по которой он не `const` (вы не смогли бы его изменить иначе).

Постфиксный оператор возвращает по значению временное (предыдущее значение), и поэтому он не может быть ссылкой, так как это будет ссылка на временную, которая будет значением мусора в конце функции, поскольку временная переменная гаснет (объект). Он также не может быть `const`, потому что вы должны иметь возможность изменять его напрямую.

Первый аргумент - это `const` ссылка на объект «вызов», потому что если он был `const`, вы не сможете его модифицировать, и если бы это не было ссылкой, вы не изменили бы исходное значение.

Это из-за копирования, необходимого для перегрузки операторов postfix, что лучше сделать привычкой использовать prefix ++ вместо postfix ++ in `for` циклов. С `for` точки контура, они, как правило, функционально эквивалентны, но может быть небольшое преимущество в производительности с использованием префикса ++, особенно с «тучными» классами с большим количеством участников для копирования. Пример использования префикса ++ в цикле `for`:

```
for (list<string>::const_iterator it = tokens.begin();
     it != tokens.end();
     ++it) { // Don't use it++
    ...
}
```

Операторы сравнения

Вы можете перегрузить все операторы сравнения:

- `==` и `!=`
- `>` и `<`
- `>=` и `<=`

Рекомендуемый способ перегрузить все эти операторы - это реализовать только 2 оператора (`==` и `<`), а затем использовать их для определения остальных. *Прокрутите вниз*

для объяснения

Перегрузка вне `class / struct` :

```
//Only implement those 2
bool operator==(const T& lhs, const T& rhs) { /* Compare */ }
bool operator<(const T& lhs, const T& rhs) { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& lhs, const T& rhs) { return !(lhs == rhs); }
bool operator>(const T& lhs, const T& rhs) { return rhs < lhs; }
bool operator<=(const T& lhs, const T& rhs) { return !(lhs > rhs); }
bool operator>=(const T& lhs, const T& rhs) { return !(lhs < rhs); }
```

Перегрузка внутри `class / struct` :

```
//Note that the functions are const, because if they are not const, you wouldn't be able
//to call them if the object is const

//Only implement those 2
bool operator==(const T& rhs) const { /* Compare */ }
bool operator<(const T& rhs) const { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& rhs) const { return !(*this == rhs); }
bool operator>(const T& rhs) const { return rhs < *this; }
bool operator<=(const T& rhs) const { return !(*this > rhs); }
bool operator>=(const T& rhs) const { return !(*this < rhs); }
```

Операторы, очевидно, возвращают `bool` , указав `true` или `false` для соответствующей операции.

Все операторы принимают свои аргументы `const&` потому, что единственное, что делают операторы, - это сравнивать, поэтому они не должны изменять объекты. Проходя мимо `&` (ссылка) быстрее , чем по значению, и , чтобы убедиться , что операторы не изменяют его, это `const` -reference.

Обратите внимание, что операторы внутри `class / struct` определены как `const` , причина в том, что без функций `const` , сравнение объектов `const` будет невозможно, поскольку компилятор не знает, что операторы ничего не изменяют.

Операторы преобразования

Вы можете перегружать операторы типа, чтобы ваш тип мог быть неявно преобразован в указанный тип.

Оператор преобразования **должен** быть определен в `class / struct` :

```
operator T() const { /* return something */ }
```

Примечание: оператор `const` является `const` позволяющей преобразовывать объекты `const` .

Пример:

```
struct Text
{
    std::string text;

    // Now Text can be implicitly converted into a const char*
    /*explicit*/ operator const char*() const { return text.data(); }
    // ^^^^^^^
    // to disable implicit conversion
};

Text t;
t.text = "Hello world!";

//Ok
const char* copyoftext = t;
```

Оператор индексирования массива

Вы даже можете перегрузить оператор индексирования массива `[]` .

Вы **всегда** должны (99.98% времени) реализовывать 2 версии, версию `const` и `not-const` , потому что, если объект `const` , он не должен изменять объект, возвращенный `[]` .

Аргументы передаются `const&` вместо значения, потому что передача по ссылке быстрее, чем по значению, и `const` чтобы оператор не менял индекс случайно.

Операторы возвращаются по ссылке, потому что по дизайну вы можете изменить объект `[]` `return`, то есть:

```
std::vector<int> v{ 1 };
v[0] = 2; //Changes value of 1 to 2
          //wouldn't be possible if not returned by reference
```

Вы можете перегружать **только** внутри `class` / `struct` :

```
//I is the index type, normally an int
T& operator[](const I& index)
{
    //Do something
    //return something
}

//I is the index type, normally an int
const T& operator[](const I& index) const
{
    //Do something
    //return something
}
```

Множественные индексы, `[][]...`, могут быть достигнуты через прокси-объекты. Следующий пример простого строкового матричного класса демонстрирует следующее:

```
template<class T>
class matrix {
    // class enabling [][] overload to access matrix elements
    template <class C>
    class proxy_row_vector {
        using reference = decltype(std::declval<C>()[0]);
        using const_reference = decltype(std::declval<C const>()[0]);
    public:
        proxy_row_vector(C& _vec, std::size_t _r_ind, std::size_t _cols)
            : vec(_vec), row_index(_r_ind), cols(_cols) {}
        const_reference operator[](std::size_t _col_index) const {
            return vec[row_index*cols + _col_index];
        }
        reference operator[](std::size_t _col_index) {
            return vec[row_index*cols + _col_index];
        }
    private:
        C& vec;
        std::size_t row_index; // row index to access
        std::size_t cols; // number of columns in matrix
    };

    using const_proxy = proxy_row_vector<const std::vector<T>>;
    using proxy = proxy_row_vector<std::vector<T>>>;
public:
    matrix() : mtx(), rows(0), cols(0) {}
    matrix(std::size_t _rows, std::size_t _cols)
        : mtx(_rows*_cols), rows(_rows), cols(_cols) {}

    // call operator[] followed by another [] call to access matrix elements
    const_proxy operator[](std::size_t _row_index) const {
        return const_proxy(mtx, _row_index, cols);
    }

    proxy operator[](std::size_t _row_index) {
        return proxy(mtx, _row_index, cols);
    }
private:
    std::vector<T> mtx;
    std::size_t rows;
    std::size_t cols;
};
```

Оператор вызова функции

Вы можете перегрузить оператор вызова функции `()` :

Перегрузка должна выполняться внутри `class` / `struct` :

```
//R -> Return type
//Types -> any different type
R operator()(Type name, Type2 name2, ...)
{
    //Do something
```

```
//return something
}

//Use it like this (R is return type, a and b are variables)
R foo = object(a, b, ...);
```

Например:

```
struct Sum
{
    int operator()(int a, int b)
    {
        return a + b;
    }
};

//Create instance of struct
Sum sum;
int result = sum(1, 1); //result == 2
```

Оператор присваивания

Оператор присваивания является одним из наиболее важных операторов, поскольку он позволяет изменять статус переменной.

Если вы не перегружаете оператор-ассистент для своего `class` / `struct`, он автоматически генерируется компилятором: автоматически созданный оператор присваивания выполняет «членное присвоение», то есть путем вызова операторов присваивания всем членам, так что один объект копируется другому, член в момент времени. Оператор присваивания должен быть перегружен, когда простое назначение по порядку не подходит для вашего `class` / `struct`, например, если вам нужно выполнить **глубокую копию** объекта.

Перегрузка оператора присваивания = легко, но вы должны выполнить несколько простых шагов.

1. **Тест для самостоятельного назначения.** Эта проверка важна по двум причинам:
 - самоназначение - это ненужная копия, поэтому нет смысла ее выполнять;
 - следующий шаг не будет работать в случае самоопределения.
2. **Очистите старые данные.** Старые данные должны быть заменены новыми. Теперь вы можете понять вторую причину предыдущего шага: если содержимое объекта было уничтожено, самоопределение не сможет выполнить копию.
3. **Скопируйте всех участников.** Если вы перегружаете оператор-ассистент для своего `class` или вашей `struct`, он не генерируется автоматически компилятором, поэтому вам нужно будет взять на себя ответственность за копирование всех членов из другого объекта.
4. **Верните `*this`.** Оператор возвращает себя по ссылке, потому что он позволяет связывать (т. `int b = (a = 6) + 4; //b == 10`).


```
//T is some type
T& operator=(const T& other)
{
    //Do something (like copying values)
    return *this;
}
```

Примечание: `other` передается `const&` потому, что назначаемый объект не должен быть изменен, а передача по ссылке быстрее, чем по значению, и чтобы убедиться, что `operator=` не модифицирует его случайно, это `const`.

Оператор присваивания может **только** быть перегружен в `class / struct`, так как слева значение = **всегда** `class / struct` сама по себе. Из-за этого определение этой функции как свободной функции не имеет этой гарантии.

Когда вы объявляете его в `class / struct`, левое значение неявно является самим `class / struct`, поэтому с этим не возникает никаких проблем.

Побитовый оператор NOT

Перегрузка побитового NOT (`~`) довольно проста. *Прокрутите вниз для объяснения*

Перегрузка вне `class / struct`:

```
T operator~(T lhs)
{
    //Do operation
    return lhs;
}
```

Перегрузка внутри `class / struct`:

```
T operator~()
{
    T t(*this);
    //Do operation
    return t;
}
```

Примечание: `operator~` возвращает значение, потому что он должен возвращать новое значение (измененное значение), а не ссылку на значение (это будет ссылка на временный объект, который будет иметь значение мусора в нем, как только оператор выполнен). Не `const` потому что вызывающий код должен иметь возможность модифицировать его впоследствии (`T int a = ~a + 1;` должно быть возможно).

Внутри `class / struct` вы должны сделать временный объект, потому что вы не можете изменить `this`, как бы изменить исходный объект, который не должен быть так.

Операторы сдвига битов для ввода / вывода

Операторы `<<` и `>>` обычно используются как операторы «write» и «read»:

- `std::ostream` перегружает `<<` для записи переменных в базовый поток (пример: `std::cout`)
- `std::istream` overloads `>>` читать из базового потока в переменную (пример: `std::cin`)

То, как они это делают, аналогично, если вы хотели перегрузить их «нормально» вне `class` / `struct` , за исключением того, что указание аргументов не одного типа:

- Тип возврата - это поток, который вы хотите перегрузить (например, `std::ostream`), переданный по ссылке, чтобы обеспечить цепочку (Chaining: `std::cout << a << b;`).
Пример: `std::ostream&`
- `lhs` будет таким же, как тип возврата
- `rhs` типа вы хотите разрешить перегрузку из (т.е. `T`), принятый `const&` вместо значения по причине производительности (`rhs` не должны быть изменены в любом случае).
Пример: `const Vector& .`

Пример:

```
//Overload std::ostream operator<< to allow output from Vector's
std::ostream& operator<<(std::ostream& lhs, const Vector& rhs)
{
    lhs << "x: " << rhs.x << " y: " << rhs.y << " z: " << rhs.z << '\n';
    return lhs;
}

Vector v = { 1, 2, 3};

//Now you can do
std::cout << v;
```

Сложные номера

В приведенном ниже коде реализована очень простая сложная тип номера, для которой автоматически создается базовое поле, следуя правилам продвижения типа языка, при применении четырех основных операторов (+, -, * и /) с членом другого поля (будь то другой `complex<T>` или некоторый скалярный тип).

Это предназначено для целостного примера, охватывающего перегрузку оператора наряду с основным использованием шаблонов.

```
#include <type_traits>

namespace not_std{

using std::decay_t;
```

```

//-----
// complex< value_t >
//-----

template<typename value_t>
struct complex
{
    value_t x;
    value_t y;

    complex &operator += (const value_t &x)
    {
        this->x += x;
        return *this;
    }
    complex &operator += (const complex &other)
    {
        this->x += other.x;
        this->y += other.y;
        return *this;
    }

    complex &operator -= (const value_t &x)
    {
        this->x -= x;
        return *this;
    }
    complex &operator -= (const complex &other)
    {
        this->x -= other.x;
        this->y -= other.y;
        return *this;
    }

    complex &operator *= (const value_t &s)
    {
        this->x *= s;
        this->y *= s;
        return *this;
    }
    complex &operator *= (const complex &other)
    {
        (*this) = (*this) * other;
        return *this;
    }

    complex &operator /= (const value_t &s)
    {
        this->x /= s;
        this->y /= s;
        return *this;
    }
    complex &operator /= (const complex &other)
    {
        (*this) = (*this) / other;
        return *this;
    }

    complex(const value_t &x, const value_t &y)
    : x{x}
    , y{y}

```

```

{}

template<typename other_value_t>
explicit complex(const complex<other_value_t> &other)
: x{static_cast<const value_t &>(other.x)}
, y{static_cast<const value_t &>(other.y)}
{}

complex &operator = (const complex &) = default;
complex &operator = (complex &&) = default;
complex(const complex &) = default;
complex(complex &&) = default;
complex() = default;
};

// Absolute value squared
template<typename value_t>
value_t absqr(const complex<value_t> &z)
{ return z.x*z.x + z.y*z.y; }

//-----
// operator - (negation)
//-----

template<typename value_t>
complex<value_t> operator - (const complex<value_t> &z)
{ return {-z.x, -z.y}; }

//-----
// operator +
//-----

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x + b.x)>>
{ return{a.x + b.x, a.y + b.y}; }

template<typename left_t,typename right_t>
auto operator + (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a + b.x)>>
{ return{a + b.x, b.y}; }

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x + b)>>
{ return{a.x + b, a.y}; }

//-----
// operator -
//-----

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x - b.x)>>
{ return{a.x - b.x, a.y - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a - b.x)>>
{ return{a - b.x, - b.y}; }

```

```

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x - b)>>
{ return{a.x - b, a.y}; }

//-----
// operator *
//-----

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x * b.x)>>
{
    return {
        a.x*b.x - a.y*b.y,
        a.x*b.y + a.y*b.x
    };
}

template<typename left_t, typename right_t>
auto operator * (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a * b.x)>>
{ return {a * b.x, a * b.y}; }

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x * b)>>
{ return {a.x * b, a.y * b}; }

//-----
// operator /
//-----

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x / b.x)>>
{
    const auto r = absqr(b);
    return {
        ( a.x*b.x + a.y*b.y) / r,
        (-a.x*b.y + a.y*b.x) / r
    };
}

template<typename left_t, typename right_t>
auto operator / (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a / b.x)>>
{
    const auto s = a/absqr(b);
    return {
        b.x * s,
        -b.y * s
    };
}

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x / b)>>
{ return {a.x / b, a.y / b}; }

} // namespace not_std

```

```

int main(int argc, char **argv)
{
    using namespace not_std;

    complex<float> fz{4.0f, 1.0f};

    // makes a complex<double>
    auto dz = fz * 1.0;

    // still a complex<double>
    auto idz = 1.0f/dz;

    // also a complex<double>
    auto one = dz * idz;

    // a complex<double> again
    auto one_again = fz * idz;

    // Operator tests, just to make sure everything compiles.

    complex<float> a{1.0f, -2.0f};
    complex<double> b{3.0, -4.0};

    // All of these are complex<double>
    auto c0 = a + b;
    auto c1 = a - b;
    auto c2 = a * b;
    auto c3 = a / b;

    // All of these are complex<float>
    auto d0 = a + 1;
    auto d1 = 1 + a;
    auto d2 = a - 1;
    auto d3 = 1 - a;
    auto d4 = a * 1;
    auto d5 = 1 * a;
    auto d6 = a / 1;
    auto d7 = 1 / a;

    // All of these are complex<double>
    auto e0 = b + 1;
    auto e1 = 1 + b;
    auto e2 = b - 1;
    auto e3 = 1 - b;
    auto e4 = b * 1;
    auto e5 = 1 * b;
    auto e6 = b / 1;
    auto e7 = 1 / b;

    return 0;
}

```

Именованные операторы

Вы можете расширить C++ с помощью именованных операторов, которые «цитируются» стандартными операторами C++.

Сначала мы начинаем с десятичной библиотеки:

```
namespace named_operator {
    template<class D>struct make_operator{constexpr make_operator(){};

    template<class T, char, class O> struct half_apply { T&& lhs; };

    template<class Lhs, class Op>
    half_apply<Lhs, '*', Op> operator*( Lhs&& lhs, make_operator<Op> ) {
        return {std::forward<Lhs>(lhs)};
    }

    template<class Lhs, class Op, class Rhs>
    auto operator*( half_apply<Lhs, '*', Op>&& lhs, Rhs&& rhs )
    -> decltype( named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) ) )
    {
        return named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) );
    }
}
```

это еще ничего не делает.

Во-первых, добавляющие векторы

```
namespace my_ns {
    struct append_t : named_operator::make_operator<append_t> {};
    constexpr append_t append{};

    template<class T, class A0, class A1>
    std::vector<T, A0> named_invoke( std::vector<T, A0> lhs, append_t, std::vector<T, A1> const&
    rhs ) {
        lhs.insert( lhs.end(), rhs.begin(), rhs.end() );
        return std::move(lhs);
    }
}
using my_ns::append;

std::vector<int> a {1,2,3};
std::vector<int> b {4,5,6};

auto c = a *append* b;
```

Ядро здесь состоит в том, что мы определяем объект `append` типа

`append_t:named_operator::make_operator<append_t>` .

Затем мы перегружаем `named_invoke (lhs, append_t, rhs)` для типов, которые мы хотим, справа и слева.

Библиотека перегружает `lhs*append_t` , возвращая временный объект `half_apply` . Он также перегружает `half_apply*rhs` для вызова `named_invoke ( lhs, append_t, rhs )` .

Нам просто нужно создать правильный токен `append_t` и сделать ADL-friendly `named_invoke` соответствующей подписи, и все будет работать и работать.

Для более сложного примера предположим, что вы хотите иметь элементарное умножение

элементов `std::array`:

```
template<class=void, std::size_t...Is>
auto indexer( std::index_sequence<Is...> ) {
    return [] (auto&& f) {
        return f( std::integral_constant<std::size_t, Is>{}... );
    };
}

template<std::size_t N>
auto indexer() { return indexer( std::make_index_sequence<N>{} ); }

namespace my_ns {
    struct e_times_t : named_operator::make_operator<e_times_t> {};
    constexpr e_times_t e_times{};

    template<class L, class R, std::size_t N,
             class Out=std::decay_t<decltype( std::declval<L const&>()*std::declval<R const&>() )>
             >
    std::array<Out, N> named_invoke( std::array<L, N> const& lhs, e_times_t, std::array<R, N>
    const& rhs ) {
        using result_type = std::array<Out, N>;
        auto index_over_N = indexer<N>();
        return index_over_N([&](auto...is)->result_type {
            return {{
                (lhs[is] * rhs[is])...
            }};
        });
    }
}
```

живой пример .

Этот элементный код массива может быть расширен для работы с кортежами или парами или массивами С-стиля или даже с контейнерами переменной длины, если вы решите, что делать, если длины не совпадают.

Вы также можете использовать тип элементарного типа и получить `lhs *element_wise<'+'>* rhs` .

Написание `*dot*` и `*cross*` продуктов также очевидны.

Использование `*` может быть расширено для поддержки других разделителей, таких как `+` . Прецизионность метрики определяет точность упомянутого оператора, что может быть важным при переводе физических уравнений на C++ с минимальным использованием дополнительных `()` S.

С небольшим изменением в библиотеке выше мы можем поддерживать операторы `->*then*` и расширять `std::function` до обновляемого стандарта или писать `monadic ->*bind*` . У него также может быть оператор с именем `stateful`, где мы тщательно передаем `Op` до последней функции `invoke`, разрешая:

```
named_operator<'**> append = [] (auto lhs, auto&& rhs) {
    using std::begin; using std::end;
```



```
lhs.insert( end(lhs), begin(rhs), end(rhs) );  
return std::move(lhs);  
};
```

создание именованного оператора контейнера-добавления в C ++ 17.

Прочитайте Перегрузка оператора онлайн: <https://riptutorial.com/ru/cplusplus/topic/562/перегрузка-оператора>

глава 76: Перегрузка функции

Вступление

См. Также отдельную тему о [разрешении перегрузки](#)

замечания

Неоднозначность может возникать, когда один тип может быть неявно преобразован в несколько типов, и для этого конкретного типа нет соответствующей функции.

Например:

```
void foo(double, double);
void foo(long, long);

//Call foo with 2 ints
foo(1, 2); //Function call is ambiguous - int can be converted into a double/long at the same
time
```

Examples

Что такое перегрузка функций?

Перегрузка функций имеет несколько функций, объявленных в одной и той же области с тем же именем, существующим в одном и том же месте (известном как *область действия*), отличающимся только их *сигнатурой*, что означает аргументы, которые они принимают.

Предположим, вы пишете ряд функций для обобщенных возможностей печати, начиная с `std::string`:

```
void print(const std::string &str)
{
    std::cout << "This is a string: " << str << std::endl;
}
```

Это прекрасно работает, но предположим, что вам нужна функция, которая также принимает `int` и печатает это тоже. Вы могли бы написать:

```
void print_int(int num)
{
    std::cout << "This is an int:  " << num << std::endl;
}
```

Но поскольку две функции принимают разные параметры, вы можете просто написать:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Теперь у вас есть 2 функции, оба с именем `print`, но с разными сигнатурами. Один принимает `std::string`, другой - `int`. Теперь вы можете позвонить им, не беспокоясь о разных именах:

```
print("Hello world!"); //prints "This is a string: Hello world!"
print(1337);           //prints "This is an int: 1337"
```

Вместо:

```
print("Hello world!");
print_int(1337);
```

Когда у вас есть перегруженные функции, компилятор указывает, какую из функций вызывать из параметров, которые вы ему предоставляете. Необходимо соблюдать осторожность при записи перегрузок функций. Например, с неявными преобразованиями типов:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
void print(double num)
{
    std::cout << "This is a double: " << num << std::endl;
}
```

Теперь не сразу понятно, какая перегрузка `print` вызывается при написании:

```
print(5);
```

И вам может понадобиться дать вашему компилятору некоторые подсказки, например:

```
print(static_cast<double>(5));
print(static_cast<int>(5));
print(5.0);
```

Некоторая забота также должна быть предпринята при написании перегрузок, которые принимают необязательные параметры:

```
// WRONG CODE
void print(int num1, int num2 = 0)    //num2 defaults to 0 if not included
{
    std::cout << "These are ints: << num1 << " and " << num2 << std::endl;
}
void print(int num)
```

```
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Поскольку компилятор не может определить, предназначен ли вызов типа `print(17)` для первой или второй функции из-за необязательного второго параметра, это не скомпилируется.

Тип возврата в функции Перегрузка

Обратите внимание, что вы не можете перегружать функцию на основе ее возвращаемого типа. Например:

```
// WRONG CODE
std::string getValue()
{
    return "hello";
}

int getValue()
{
    return 0;
}

int x = getValue();
```

Это вызовет ошибку компиляции, поскольку компилятор не сможет определить, какую версию `getValue` вызывать, даже если тип возвращаемого значения присваивается `int`.

Функция члена `cv-qualifier` Перегрузка

Функции внутри класса могут быть перегружены, когда они доступны через `cv`-квалифицированную ссылку на этот класс; это чаще всего используется для перегрузки для `const`, но также может использоваться для перегрузки для `volatile` и `const volatile`. Это связано с тем, что все нестатические функции-члены принимают `this` как скрытый параметр, к которому применяются `cv`-квалификаторы. Это чаще всего используется для перегрузки для `const`, но также может использоваться для `volatile` и `const volatile`.

Это необходимо, потому что функция-член может быть вызвана только в том случае, если она, по меньшей мере, `cv`-квалифицирована как экземпляр, на который он вызван. В то время как экземпляр `non-const` может вызывать как `const` и не `const` члены, экземпляр `const` может вызывать только `const` члены. Это позволяет функции иметь различное поведение в зависимости от `cv`-квалификаторов вызывающего экземпляра и позволяет программисту запрещать функции для нежелательного `cv`-квалификатора (ов), не предоставляя версию с этим квалификатором.

Класс с некоторым основным методом `print` может быть `const` перегружен следующим образом:

```

#include <iostream>

class Integer
{
public:
    Integer(int i_) : i{i_}{}

    void print()
    {
        std::cout << "int: " << i << std::endl;
    }

    void print() const
    {
        std::cout << "const int: " << i << std::endl;
    }

protected:
    int i;
};

int main()
{
    Integer i{5};
    const Integer &ic = i;

    i.print(); // prints "int: 5"
    ic.print(); // prints "const int: 5"
}

```

Это ключевой принцип корректности `const` : помечая функции-члены как `const` , они могут быть вызваны в экземплярах `const` , что, в свою очередь, позволяет функциям принимать экземпляры как указатели / ссылки `const` если им не нужно их изменять. Это позволяет коду определять, изменяет ли он состояние, принимая немодифицированные параметры как `const` и измененные параметры без св-квалификаторов, делая код более безопасным и читаемым.

```

class ConstCorrect
{
public:
    void good_func() const
    {
        std::cout << "I care not whether the instance is const." << std::endl;
    }

    void bad_func()
    {
        std::cout << "I can only be called on non-const, non-volatile instances." <<
std::endl;
    }
};

void i_change_no_state(const ConstCorrect& cc)
{
    std::cout << "I can take either a const or a non-const ConstCorrect." << std::endl;
    cc.good_func(); // Good. Can be called from const or non-const instance.
    cc.bad_func(); // Error. Can only be called from non-const instance.
}

```

```
void const_incorrect_func(ConstCorrect& cc)
{
    cc.good_func(); // Good. Can be called from const or non-const instance.
    cc.bad_func();  // Good. Can only be called from non-const instance.
}
```

Общим применением этого является объявление аксессоров как `const` , а мутаторы - как `const` .

Никакие члены класса не могут быть изменены внутри функции `const member`. Если есть какой-то член, который вам действительно нужно изменить, например блокировку `std::mutex` , вы можете объявить его как `mutable` :

```
class Integer
{
public:
    Integer(int i_): i{i_}{}

    int get() const
    {
        std::lock_guard<std::mutex> lock{mut};
        return i;
    }

    void set(int i_)
    {
        std::lock_guard<std::mutex> lock{mut};
        i = i_;
    }

protected:
    int i;
    mutable std::mutex mut;
};
```

Прочитайте Перегрузка функции онлайн: <https://riptutorial.com/ru/cplusplus/topic/510/перегрузка-функции>

глава 77: Перегрузка функциональных шаблонов

замечания

- Нормальная функция никогда не связана с шаблоном функции, несмотря на то же имя, тот же тип.
- Обычный вызов функции и вызов шаблона сгенерированной функции различны, даже если они имеют одно и то же имя, одинаковый тип возвращаемого значения и один список аргументов

Examples

Что такое допустимая перегрузка шаблона функции?

Шаблон функции может быть перегружен по правилам для перегрузки без шаблонов (одно имя, но разные типы параметров), и в дополнение к этому перегрузка действительна, если

- Тип возврата отличается, или
- Список параметров шаблона отличается, за исключением названий параметров и наличия аргументов по умолчанию (они не являются частью подписи)

Для нормальной функции сравнение двух типов параметров легко для компилятора, так как оно имеет всю информацию. Но тип внутри шаблона еще не определен. Таким образом, правило, когда два типа параметров равны, здесь аппроксимативно и говорит, что типы и значения, не относящиеся к депенденду, должны совпадать, а написание зависимых типов и выражений должно быть одинаковым (точнее, они должны соответствовать так называемые ODR-правила), за исключением того, что параметры шаблона могут быть переименованы. Однако, если при таких разных написаниях два значения внутри типов считаются разными, но всегда будут создавать экземпляры с одинаковыми значениями, перегрузка недействительна, но никакой диагностики не требуется от компилятора.

```
template<typename T>
void f(T*) { }

template<typename T>
void f(T) { }
```

Это допустимая перегрузка, так как «Т» и «Т *» - это разные варианты написания. Но следующее недопустимо, без необходимости диагностики

```
template<typename T>
```

```
void f(T (*x)[sizeof(T) + sizeof(T)]) { }
```



```
template<typename T>  
void f(T (*x)[2 * sizeof(T)]) { }
```

Прочитайте Перегрузка функциональных шаблонов онлайн:

<https://riptutorial.com/ru/cplusplus/topic/4164/перегрузка-функциональных-шаблонов>

глава 78: Переместить семантику

Examples

Переместить семантику

Перенос семантики - это способ перемещения одного объекта на другой в C++. Для этого мы очищаем старый объект и помещаем все, что было в новом объекте.

Для этого мы должны понять, что такое ссылка rvalue. Ссылка rvalue (`T&&` где `T` - тип объекта) не сильно отличается от обычной ссылки (`T&`, теперь называемой ссылкой lvalue). Но они действуют как два разных типа, и поэтому мы можем создавать конструкторы или функции, которые принимают один тип или другой, что необходимо при использовании семантики перемещения.

Причина, по которой нам нужны два разных типа, - это указать два разных поведения. Конструкторы ссылок lvalue связаны с копированием, а конструкторы ссылок rvalue связаны с перемещением.

Чтобы переместить объект, мы будем использовать `std::move(obj)`. Эта функция возвращает ссылку на объект rvalue, чтобы мы могли украсть данные из этого объекта в новый. Существует несколько способов сделать это, которые обсуждаются ниже.

Важно отметить, что использование `std::move` создает только ссылку rvalue. Другими словами, оператор `std::move(obj)` не изменяет содержимое `obj`, тогда как `auto obj2 = std::move(obj)` (возможно).

Переместить конструктор

Скажем, у нас есть этот фрагмент кода.

```
class A {
public:
    int a;
    int b;

    A(const A &other) {
        this->a = other.a;
        this->b = other.b;
    }
};
```

Чтобы создать конструктор копирования, то есть создать функцию, которая копирует объект и создает новую, мы обычно выбираем синтаксис, показанный выше, у нас будет конструктор для `A`, который ссылается на другой объект типа `A`, и мы будем копировать объект вручную внутри метода.

В качестве альтернативы мы могли бы написать `A(const A &) = default;` который автоматически копирует все элементы, используя его конструктор копирования.

Однако для создания конструктора перемещения мы будем использовать ссылку `rvalue` вместо ссылки `lvalue`, как здесь.

```
class Wallet {
public:
    int nrOfDollars;

    Wallet() = default; //default ctor

    Wallet(Wallet &&other) {
        this->nrOfDollars = other.nrOfDollars;
        other.nrOfDollars = 0;
    }
};
```

Обратите внимание, что мы установили старые значения в `zero`. Конструктор перемещения по умолчанию (`Wallet(Wallet&&) = default;`) копирует значение `nrOfDollars`, так как это POD.

Поскольку семантика перемещения предназначена для того, чтобы разрешить «кражи» состояния из исходного экземпляра, важно рассмотреть, как должен выглядеть исходный экземпляр после этого кражи. В этом случае, если бы мы не изменили значение до нуля, мы бы удвоили сумму долларов в игре.

```
Wallet a;
a.nrOfDollars = 1;
Wallet b (std::move(a)); //calling B(B&& other);
std::cout << a.nrOfDollars << std::endl; //0
std::cout << b.nrOfDollars << std::endl; //1
```

Таким образом, мы перемещаем объект из старого.

Хотя приведенный выше пример является простым, он показывает, что должен сделать конструктор перемещения. Он становится более полезным в более сложных случаях, например, при управлении ресурсами.

```
// Manages operations involving a specified type.
// Owns a helper on the heap, and one in its memory (presumably on the stack).
// Both helpers are DefaultConstructible, CopyConstructible, and MoveConstructible.
template<typename T,
        template<typename> typename HeapHelper,
        template<typename> typename StackHelper>
class OperationsManager {
    using MyType = OperationsManager<T, HeapHelper, StackHelper>;

    HeapHelper<T>* h_helper;
    StackHelper<T> s_helper;
    // ...

public:
```

```

// Default constructor & Rule of Five.
OperationsManager() : h_helper(new HeapHelper<T>) {}
OperationsManager(const MyType& other)
    : h_helper(new HeapHelper<T>(*other.h_helper)), s_helper(other.s_helper) {}
MyType& operator=(MyType copy) {
    swap(*this, copy);
    return *this;
}
~OperationsManager() {
    if (h_helper) { delete h_helper; }
}

// Move constructor (without swap()).
// Takes other's HeapHelper<T>*.
// Takes other's StackHelper<T>, by forcing the use of StackHelper<T>'s move
constructor.
// Replaces other's HeapHelper<T>* with nullptr, to keep other from deleting our shiny
// new helper when it's destroyed.
OperationsManager(MyType&& other) noexcept
    : h_helper(other.h_helper),
      s_helper(std::move(other.s_helper)) {
    other.h_helper = nullptr;
}

// Move constructor (with swap()).
// Places our members in the condition we want other's to be in, then switches members
// with other.
// OperationsManager(MyType&& other) noexcept : h_helper(nullptr) {
//     swap(*this, other);
// }

// Copy/move helper.
friend void swap(MyType& left, MyType& right) noexcept {
    std::swap(left.h_helper, right.h_helper);
    std::swap(left.s_helper, right.s_helper);
}
};

```

Переместить назначение

Подобно тому, как мы можем присвоить значение объекту с ссылкой lvalue, скопировав его, мы также можем перемещать значения из объекта в другой, не создавая новый. Мы называем это назначение перемещения. Мы перемещаем значения из одного объекта в другой существующий объект.

Для этого нам придется перегружать `operator =`, а не так, чтобы он ссылался на ссылку lvalue, как и при копировании, но так, чтобы он принимал ссылку rvalue.

```

class A {
    int a;
    A& operator= (A&& other) {
        this->a = other.a;
        other.a = 0;
        return *this;
    }
};

```

Это типичный синтаксис для определения перемещения. Мы перегружаем `operator =` так, чтобы мы могли подать ему ссылку на `rvalue` и присвоить ее другому объекту.

```
A a;
a.a = 1;
A b;
b = std::move(a); //calling A& operator= (A&& other)
std::cout << a.a << std::endl; //0
std::cout << b.a << std::endl; //1
```

Таким образом, мы можем переместить назначение объекта другому.

Использование `std::move` для уменьшения сложности от $O(n^2)$ до $O(n)$

C++ 11 представил базовый язык и стандартную библиотечную поддержку для **перемещения** объекта. Идея заключается в том, что, когда объект `o` является временным, и один хочет логическую копию, то ее безопасно только стянуть `o` «с» ресурсы, такие как динамически выделенный буфер, оставляя `o` логически пустым, но по-прежнему разрушаемость и копируемой.

Поддержка основного языка в основном

- конструктор **ссылочного** типа **`rvalue`** `&&`, например `std::string&&` является ссылкой `rvalue` на `std::string`, указывая на то, что упомянутый объект является временным, ресурсы которого могут быть просто украдены (т.е. перемещены)
- специальную поддержку для **конструктора перемещения** `T(T&&)`, который должен эффективно перемещать ресурсы из указанного другого объекта, а не фактически копировать эти ресурсы, и
- специальная поддержка **оператора** `auto operator=(T&&) -> T&` **присваивания** **оператора** `auto operator=(T&&) -> T&`, который также должен перемещаться из источника.

Стандартная поддержка библиотеки - это главным образом шаблон `std::move` из заголовка `<utility>`. Эта функция создает ссылку `rvalue` для указанного объекта, указывая на то, что она может быть перемещена, как если бы она была временной.

Для контейнера фактическое копирование обычно имеет сложность $O(n)$, где n - количество элементов в контейнере, а перемещение - $O(1)$, постоянное время. А для алгоритма, который логически копирует этот контейнер n раз, это может уменьшить сложность от обычно непрактичного $O(n^2)$ до линейного $O(n)$.

В своей статье «[Контейнеры, которые никогда не меняются](#)» в *Dr. Dobbs Journal* в сентябре 19 2013 года Эндрю Кениг представил интересный пример алгоритмической неэффективности при использовании стиля программирования, где переменные

неизменны после инициализации. С помощью этого стиля петли обычно выражаются с помощью рекурсии. И для некоторых алгоритмов, таких как создание последовательности Collatz, рекурсия требует логического копирования контейнера:

```
// Based on an example by Andrew Koenig in his Dr. Dobbs Journal article
// "Containers That Never Change" September 19, 2013, available at
// <url: http://www.drdobbs.com/cpp/containers-that-never-change/240161543>

// Includes here, e.g. <vector>

namespace my {
    template< class Item >
    using Vector_ = /* E.g. std::vector<Item> */;

    auto concat( Vector_<int> const& v, int const x )
        -> Vector_<int>
    {
        auto result{ v };
        result.push_back( x );
        return result;
    }

    auto collatz_aux( int const n, Vector_<int> const& result )
        -> Vector_<int>
    {
        if( n == 1 )
        {
            return result;
        }
        auto const new_result = concat( result, n );
        if( n % 2 == 0 )
        {
            return collatz_aux( n/2, new_result );
        }
        else
        {
            return collatz_aux( 3*n + 1, new_result );
        }
    }

    auto collatz( int const n )
        -> Vector_<int>
    {
        assert( n != 0 );
        return collatz_aux( n, Vector_<int>() );
    }
} // namespace my

#include <iostream>
using namespace std;
auto main() -> int
{
    for( int const x : my::collatz( 42 ) )
    {
        cout << x << ' ';
    }
    cout << '\n';
}
```

Выход:

```
42 21 64 32 16 8 4 2
```

Количество операций копирования элементов из-за копирования векторов здесь примерно равно $O(n^2)$, так как это сумма $1 + 2 + 3 + \dots + n$.

В конкретных числах, с компиляторами g++ и Visual C++, вышеупомянутый вызов `collatz(42)` привел к последовательности Collatz из 8 элементов и 36 операций копирования экземпляров ($8 * \text{collatz}(42) = 28$, плюс некоторые) в вызовах конструктора вектора.

Все эти операции копирования элементов можно удалить простым переносом векторов, значения которых больше не нужны. Для этого необходимо удалить `const` и ссылку для аргументов типа вектора, передав векторы *по значению*. Возврат функции автоматически оптимизирован. Для вызовов, где векторы передаются и не используются снова в функции, просто примените `std::move` чтобы *переместить* эти буферы, а не копировать их:

```
using std::move;

auto concat( Vector<int> v, int const x )
    -> Vector<int>
{
    v.push_back( x );
    // warning: moving a local object in a return statement prevents copy elision [-Wpessimizing-move]
    // See https://stackoverflow.com/documentation/c%2b%2b/2489/copy-elision
    // return move( v );
    return v;
}

auto collatz_aux( int const n, Vector<int> result )
    -> Vector<int>
{
    if( n == 1 )
    {
        return result;
    }
    auto new_result = concat( move( result ), n );
    struct result; // Make absolutely sure no use of `result` after this.
    if( n % 2 == 0 )
    {
        return collatz_aux( n/2, move( new_result ) );
    }
    else
    {
        return collatz_aux( 3*n + 1, move( new_result ) );
    }
}

auto collatz( int const n )
    -> Vector<int>
{
    assert( n != 0 );
    return collatz_aux( n, Vector<int>() );
}
```

Здесь, с компиляторами g ++ и Visual C ++, количество операций копирования элементов из-за вызовов конструктора векторных копий было равно 0.

Алгоритм обязательно должен быть $O(n)$ в длине последовательности Collatz, но это довольно значительное улучшение: $O(n^2) \rightarrow O(n)$.

При некоторой поддержке языка можно было бы использовать перемещение и все еще выражать и обеспечивать неизменность переменной *между ее инициализацией и конечным ходом*, после чего любое использование этой переменной должно быть ошибкой. Увы, с C ++ 14 C ++ этого не поддерживает. Для кода без цикла без использования после перемещения может быть принудительно введено повторное объявление соответствующего имени как неполной `struct`, как и для `struct result`; выше, но это уродливо и вряд ли будет понято другими программистами; также диагностика может быть довольно вводящей в заблуждение.

Подводя итоги, поддержка языка C ++ и библиотеки для перемещения позволяет радикально улучшить сложность алгоритма, но из-за неполноты поддержки за счет отказа от гарантий правильности кода и ясности кода, которые может обеспечить `const`.

Для полноты, инструментальный векторный класс, используемый для измерения количества операций копирования элементов из-за вызовов конструктора копирования:

```
template< class Item >
class Copy_tracking_vector
{
private:
    static auto n_copy_ops()
        -> int&
    {
        static int value;
        return value;
    }

    vector<Item>    items_;

public:
    static auto n() -> int { return n_copy_ops(); }

    void push_back( Item const& o ) { items_.push_back( o ); }
    auto begin() const { return items_.begin(); }
    auto end() const { return items_.end(); }

    Copy_tracking_vector(){}

    Copy_tracking_vector( Copy_tracking_vector const& other )
        : items_( other.items_ )
    { n_copy_ops() += items_.size(); }

    Copy_tracking_vector( Copy_tracking_vector&& other )
        : items_( move( other.items_ ) )
    {}
};
```

Использование семантики перемещения на контейнерах

Вы можете переместить контейнер вместо копирования:

```
void print(const std::vector<int>& vec) {
    for (auto& val : vec) {
        std::cout << val << ", ";
    }
    std::cout << std::endl;
}

int main() {
    // initialize vec1 with 1, 2, 3, 4 and vec2 as an empty vector
    std::vector<int> vec1{1, 2, 3, 4};
    std::vector<int> vec2;

    // The following line will print 1, 2, 3, 4
    print(vec1);

    // The following line will print a new line
    print(vec2);

    // The vector vec2 is assigned with move assingment.
    // This will "steal" the value of vec1 without copying it.
    vec2 = std::move(vec1);

    // Here the vec1 object is in an indeterminate state, but still valid.
    // The object vec1 is not destroyed,
    // but there's is no guarantees about what it contains.

    // The following line will print 1, 2, 3, 4
    print(vec2);
}
```

Повторное использование перемещенного объекта

Вы можете повторно использовать перемещенный объект:

```
void consumingFunction(std::vector<int> vec) {
    // Some operations
}

int main() {
    // initialize vec with 1, 2, 3, 4
    std::vector<int> vec{1, 2, 3, 4};

    // Send the vector by move
    consumingFunction(std::move(vec));

    // Here the vec object is in an indeterminate state.
    // Since the object is not destroyed, we can assign it a new content.
    // We will, in this case, assign an empty value to the vector,
    // making it effectively empty
    vec = {};

    // Since the vector as gained a determinate value, we can use it normally.
    vec.push_back(42);
}
```



```
// Send the vector by move again.  
consumingFunction(std::move(vec));  
}
```

Прочитайте Переместить семантику онлайн: <https://riptutorial.com/ru/cplusplus/topic/2129/переместить-семантику>

глава 79: перечисление

Examples

Основная декларация перечисления

Стандартные перечисления позволяют пользователям объявлять полезное имя для набора целых чисел. Названия в совокупности называются счетчиками. Перечисление и связанные с ним счетчики определяются следующим образом:

```
enum myEnum
{
    enumName1,
    enumName2,
};
```

Перечисление - это *тип*, который отличается от всех других типов. В этом случае имя этого типа - `myEnum`. Ожидается, что объекты этого типа будут принимать значение перечислителя в перечислении.

Перечислители, объявленные в перечислении, являются постоянными значениями типа перечисления. Хотя перечисления объявляются внутри типа, оператор области видимости `::` не нужен для доступа к имени. Итак, имя первого `enumName1` - `enumName1`.

C ++ 11

Оператор области может быть необязательно использован для доступа к перечислителю в перечислении. Таким образом, `enumName1` также может быть записано `myEnum::enumName1`.

Перечислителям присваиваются целочисленные значения, начиная с 0 и возрастая на 1 для каждого перечислителя в перечислении. Таким образом, в приведенном выше случае имя `enumName1` имеет значение 0, а значение `enumName2` имеет значение 1.

Пользователям также может быть присвоено определенное значение; это значение должно быть интегральным постоянным выражением. Перечисления, значения которых не указаны явно, будут иметь значение, соответствующее значению предыдущего счетчика + 1.

```
enum myEnum
{
    enumName1 = 1, // value will be 1
    enumName2 = 2, // value will be 2
    enumName3,    // value will be 3, previous value + 1
    enumName4 = 7, // value will be 7
    enumName5,    // value will be 8
    enumName6 = 5, // value will be 5, legal to go backwards
    enumName7 = 3, // value will be 3, legal to reuse numbers
};
```

```
enumName8 = enumName4 + 2, // value will be 9, legal to take prior enums and adjust them
};
```

Перечисление в операторах switch

Обычное использование для счетчиков - это для операторов switch, и поэтому они обычно появляются в государственных машинах. На самом деле полезной особенностью операторов switch с перечислениями является то, что если для коммутатора не включен оператор по умолчанию, и не все значения перечисления были использованы, компилятор выдаст предупреждение.

```
enum State {
    start,
    middle,
    end
};

...

switch(myState) {
    case start:
        ...
    case middle:
        ...
} // warning: enumeration value 'end' not handled in switch [-Wswitch]
```

Итерация над перечислением

Для перебора не существует встроенного метода.

Но есть несколько способов

- для enum с только последовательными значениями:

```
enum E {
    Begin,
    E1 = Begin,
    E2,
    // ..
    En,
    End
};

for (E e = E::Begin; e != E::End; ++e) {
    // Do job with e
}
```

C++ 11

С enum class, operator ++ должен быть реализован:

```
E& operator ++ (E& e)
```

```

{
    if (e == E::End) {
        throw std::out_of_range("for E& operator ++ (E&)");
    }
    e = E(static_cast<std::underlying_type<E>::type>(e) + 1);
    return e;
}

```

- использование контейнера в виде `std::vector`

```

enum E {
    E1 = 4,
    E2 = 8,
    // ..
    En
};

std::vector<E> build_all_E()
{
    const E all[] = {E1, E2, /*...*/ En};

    return std::vector<E>(all, all + sizeof(all) / sizeof(E));
}

std::vector<E> all_E = build_all_E();

```

а ПОТОМ

```

for (std::vector<E>::const_iterator it = all_E.begin(); it != all_E.end(); ++it) {
    E e = *it;
    // Do job with e;
}

```

C++ 11

- или `std::initializer_list` и более простой синтаксис:

```

enum E {
    E1 = 4,
    E2 = 8,
    // ..
    En
};

constexpr std::initializer_list<E> all_E = {E1, E2, /*...*/ En};

```

а ПОТОМ

```

for (auto e : all_E) {
    // Do job with e
}

```

Скопированные перечисления

С ++ 11 вводит так называемые *скопированные перечисления* . Это перечисления, чьи члены должны быть квалифицированы с `enumname::membername` . Выделенные области перечислены с использованием синтаксиса `enum class` . Например, чтобы сохранить цвета в радуге:

```
enum class rainbow {  
    RED,  
    ORANGE,  
    YELLOW,  
    GREEN,  
    BLUE,  
    INDIGO,  
    VIOLET  
};
```

Чтобы получить доступ к определенному цвету:

```
rainbow r = rainbow::INDIGO;
```

`enum class` **es** не может быть неявно преобразован в `int` **s** без приведения. Поэтому `int x = rainbow::RED` **недействителен**.

Скопированные перечисления также позволяют указать *базовый тип* , который является типом, используемым для представления члена. По умолчанию это `int` . В игре Тис-Тас-Тое вы можете хранить кусок как

```
enum class piece : char {  
    EMPTY = '\0',  
    X = 'X',  
    O = 'O',  
};
```

Как вы можете заметить, `enum` может иметь конечную запятую после последнего члена.

Передать декларацию в С ++ 11

Области:

```
...  
enum class Status; // Forward declaration  
Status doWork(); // Use the forward declaration  
...  
enum class Status { Invalid, Success, Fail };  
Status doWork() // Full declaration required for implementation  
{  
    return Status::Success;  
}
```

Неперечисленные перечисления:

```
...
enum Status: int; // Forward declaration, explicit type required
Status doWork(); // Use the forward declaration
...
enum Status: int{ Invalid=0, Success, Fail }; // Must match forward declare type
static_assert( Success == 1 );
```

Подробный пример с несколькими файлами можно найти здесь: [Пример торговца плотными фруктами](#)

Прочитайте перечисление онлайн: <https://riptutorial.com/ru/cplusplus/topic/2796/перечисление>

глава 80: Поведение, определяемое реализацией

Examples

Char может быть неподписанным или подписанным

В стандарте не указывается, должен ли `char` быть подписан или без знака. Различные компиляторы реализуют его по-другому или могут изменить его с помощью командной строки.

Размер интегральных типов

Следующие типы определяются как *интегральные типы* :

- `char`
- Подписанные целые типы
- Беззнаковые целые типы
- `char16_t` и `char32_t`
- `bool`
- `wchar_t`

За исключением `sizeof(char)` / `sizeof(signed char)` / `sizeof(unsigned char)` , который разделен между § 3.9.1.1 [basic.fundamental / 1] и § 5.3.3.1 [expr.sizeof] и `sizeof(bool)` , который полностью определяется реализацией и не имеет минимального размера, требования минимального размера этих типов приведены в разделе § 3.9.1 [basic.fundamental] стандарта и будут подробно описаны ниже.

Размер `char`

Все версии стандарта C ++ указывают в п. 5.3.3.1, что `sizeof` дает 1 для `unsigned char` , `signed char` и `char` (это реализация определяет, `signed` ли тип `char` или `unsigned`).

C ++ 14

`char` достаточно большой, чтобы представлять 256 различных значений, чтобы быть пригодным для хранения кодовых блоков UTF-8.

Размер подписанных и неподписанных

ЦЕЛЫХ ТИПОВ

Стандарт указывает в п. 3.9.1.2, что в списке *стандартных стандартных целочисленных типов*, состоящих из `signed char`, `short int`, `int`, `long int` и `long long int`, каждый тип будет содержать как минимум столько же памяти, что и предыдущие это в списке. Кроме того, как указано в п. 3.9.1.3, каждый из этих типов имеет соответствующий *стандартный беззнаковый целочисленный тип*, `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int` и `unsigned long long int`, которые имеют одинаковый размер и выравнивание, как его соответствующий подписанный тип. Кроме того, как указано в п. 3.9.1.1, `char` имеет те же требования к размеру и выравниванию, что и как `signed char` и `unsigned char`.

C ++ 11

До C ++ 11 `long long` и `unsigned long long` официально не были частью стандарта C ++. Однако после их введения в C на C99 многие компиляторы поддерживали `long long` *расширенный целочисленный тип со* `unsigned long long`, а `unsigned long long` как *расширенный целочисленный тип без знака*, с теми же правилами, что и типы C.

Таким образом, стандарт гарантирует, что:

```
1 == sizeof(char)    == sizeof(signed char) == sizeof(unsigned char)
<= sizeof(short)    == sizeof(unsigned short)
<= sizeof(int)      == sizeof(unsigned int)
<= sizeof(long)     == sizeof(unsigned long)
```

C ++ 11

```
<= sizeof(long long) == sizeof(unsigned long long)
```

Конкретные минимальные размеры для каждого типа не указаны стандартом. Вместо этого каждый тип имеет минимальный диапазон значений, которые он может поддерживать, который, как указано в § 3.9.1.3, унаследован от стандарта C, в п. 5.2.4.2.1. Минимальный размер каждого типа может быть грубо выведен из этого диапазона, определяя минимальное количество требуемых бит; обратите внимание, что для любой данной платформы фактический поддерживаемый диапазон любого типа может быть больше минимального. Обратите внимание, что для подписанных типов диапазоны соответствуют своему дополнению, а не более часто используемому двум дополнениям; это позволяет более широкому диапазону платформ соответствовать стандарту.

Тип	Минимальный диапазон	Требуются минимальные бит
<code>signed char</code>	От -127 до 127 (- (от 2^7 до 1) до $(2^7 - 1)$)	8
<code>unsigned char</code>	От 0 до 255 (от 0 до $2^8 - 1$)	8

Тип	Минимальный диапазон	Требуются минимальные бит
signed short	-32,767 до 32,767 (- (2 ¹⁵ - 1) до (2 ¹⁵ - 1))	16
unsigned short	0 до 65535 (от 0 до 2 ¹⁶ - 1)	16
signed int	-32,767 до 32,767 (- (2 ¹⁵ - 1) до (2 ¹⁵ - 1))	16
unsigned int	0 до 65535 (от 0 до 2 ¹⁶ - 1)	16
signed long	-2,147,483,647 до 2,147,483,647 (- (2 ³¹ - 1) до (2 ³¹ - 1))	32
unsigned long	0 - 4 294 967 295 (от 0 до 2 ³² - 1)	32

C ++ 11

Тип	Минимальный диапазон	Требуются минимальные бит
signed long long	-9,223,372,036,854,775,807 - 9,223,372,036,854,775,807 (- (от 2 ⁶³ до 1) до (2 ⁶³ - 1))	64
unsigned long long	От 0 до 18 446 744 073 709 551 615 (от 0 до 2 ⁶⁴ - 1)	64

Поскольку каждому типу разрешено превышать требования к минимальному размеру, типы могут отличаться по размеру между реализациями. Наиболее примечательный пример этого - с 64-битными моделями данных LP64 и LLP64, где системы LLP64 (например, 64-разрядные Windows) имеют 32-битные `ints` и `long s`, а системы LP64 (например, 64-разрядные Linux) имеют 32-битный `int` с и 64-разрядные `long` с. Из-за этого нельзя считать, что целочисленные типы имеют фиксированную ширину на всех платформах.

C ++ 11

Если требуются целые типы с фиксированной шириной, используйте типы из заголовка `<stdint.h>`, но обратите внимание, что стандарт делает его необязательным для реализаций для поддержки типов точной ширины `int8_t`, `int16_t`, `int32_t`, `int64_t`, `intptr_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` и `uintptr_t`.

C ++ 11

Размер `char16_t` и `char32_t`

Размеры `char16_t` и `char32_t` определяются реализацией, как указано в п. 5.3.3.1, с положениями, изложенными в п. 3.9.1.5:

- `char16_t` достаточно велик, чтобы представлять любой модуль кода UTF-16 и имеет тот же размер, подпись и выравнивание как `uint_least16_t`; он должен иметь размер не менее 16 бит.
- `char32_t` достаточно велик, чтобы представлять любой кодовый блок UTF-32 и имеет тот же размер, подпись и выравнивание как `uint_least32_t`; он должен иметь размер не менее 32 бит.

Размер `bool`

Размер `bool` определяется реализацией и может быть или не быть 1.

Размер `wchar_t`

`wchar_t`, как указано в п. 3.9.1.5, представляет собой отдельный тип, диапазон значений которого может представлять каждый отдельный блок кода самого большого расширенного набора символов среди поддерживаемых локалей. Он имеет тот же размер, подпись и выравнивание как один из других интегральных типов, который известен как его *базовый тип*. Размер этого типа определяется реализацией, как указано в п. 5.3.3.1, и может быть, например, не менее 8, 16 или 32 бит; если система поддерживает Unicode, например, `wchar_t` требуется как минимум 32 бита (исключение из этого правила - Windows, где `wchar_t` - 16 бит для целей совместимости). Он наследуется от стандарта C90, ISO 9899:1990 § 4.1.5, с незначительной переписью.

В зависимости от реализации размер `wchar_t` часто, но не всегда, 8, 16 или 32 бит. Наиболее распространенными примерами этого являются:

- В Unix и Unix-подобных системах `wchar_t` является 32-разрядным и обычно используется для UTF-32.
- В Windows `wchar_t` является 16-разрядным и используется для UTF-16.
- В системе, которая имеет только 8-битную поддержку, `wchar_t` имеет 8 бит.

C ++ 11

Если требуется поддержка Unicode, рекомендуется использовать `char` для UTF-8, `char16_t` для UTF-16 или `char32_t` для UTF-32 вместо использования `wchar_t`.

Модели данных

Как упоминалось выше, ширина целых типов может различаться между платформами. Наиболее распространенными являются следующие модели с размерами, указанными в битах:

модель	int	long	указатель
LP32 (2/4/4)	16	32	32
ILP32 (4/4/4)	32	32	32
LLP64 (4/4/8)	32	32	64
LP64 (4/8/8)	32	64	64

Из этих моделей:

- 16-битная Windows использовала LP32.
- 32-битные * nix-системы (Unix, Linux, Mac OSX и другие Unix-подобные ОС) и Windows используют ILP32.
- 64-разрядная версия Windows использует LLP64.
- 64-битные * nix-системы используют LP64.

Обратите внимание, однако, что эти модели конкретно не упоминаются в самом стандарте.

Количество бит в байте

В C++ *байт* - это пространство, занимаемое объектом `char`. Количество бит в байте задается `CHAR_BIT`, которое определяется в `climits` и должно быть не менее 8. Хотя большинство современных систем имеют 8-битные байты, а POSIX требует, чтобы `CHAR_BIT` был ровно 8, существуют некоторые системы, где `CHAR_BIT` больше 8, т. е. один байт может состоять из 8, 16, 32 или 64 бит.

Числовое значение указателя

Результат литья указателя на целое число с использованием `reinterpret_cast` определяется реализацией, но «... не может быть неудивительным для тех, кто знает структуру адресации лежащей в основе машины».

```
int x = 42;
int* p = &x;
long addr = reinterpret_cast<long>(p);
std::cout << addr << "\n"; // prints some numeric address,
                             // probably in the architecture's native address format
```

Аналогично, указатель, полученный преобразованием из целого числа, также определяется реализацией.

Правильный способ хранения указателя как целого - использовать `uintptr_t` или `intptr_t` :

```
// `uintptr_t` was not in C++03. It's in C99, in <stdint.h>, as an optional type
#include <stdint.h>

uintptr_t uip;
```

C++ 11

```
// There is an optional `std::uintptr_t` in C++11
#include <cstdint>

std::uintptr_t uip;
```

C++ 11 относится к C99 для определения `uintptr_t` (стандарт C99, 6.3.2.3):

целочисленный тип без знака с тем свойством, что любой действительный указатель на `void` может быть преобразован в этот тип, а затем преобразован обратно в указатель на `void` , и результат будет сравниваться с исходным указателем.

Хотя для большинства современных платформ вы можете принять плоское адресное пространство и что арифметика на `uintptr_t` эквивалентна арифметике на `char *` , вполне возможно, что реализация выполнит любое преобразование при литье `void *` в `uintptr_t` как только трансформация может быть отменено при отбрасывании от `uintptr_t` до `void *` .

Технические детали

- В системах, `intptr_t` XSI (X / Open System Interfaces), `uintptr_t` типы `intptr_t` и `uintptr_t` , в противном случае они являются **необязательными** .
- В значении стандарта C функции не являются объектами; в стандарте C не гарантируется, что `uintptr_t` может содержать указатель на функцию. В любом случае соответствие POSIX (2.12.3) требует:

Все типы указателей функций должны иметь то же представление, что и указатель типа на `void`. Преобразование указателя функции в `void *` не должно изменять представление. Значение `void *`, возникающее в результате такого преобразования, может быть преобразовано обратно в исходный тип указателя функции с использованием явного приведения без потери информации.

- C99 §7.18.1:

Когда определены имена `typedef`, отличающиеся только отсутствием или наличием начального `u`, они должны обозначать соответствующие типы подписей и без знака, как описано в 6.2.5; реализация, обеспечивающая один из этих соответствующих типов, также должна обеспечивать другую.

`uintptr_t` может иметь смысл, если вы хотите сделать что-то с битами указателя, который вы не можете сделать так же разумно со знаком целого числа.

Диапазоны числовых типов

Диапазоны целочисленных типов определены реализацией. Заголовок `<limits>` содержит шаблон `std::numeric_limits<T>` который предоставляет минимальные и максимальные значения всех основных типов. Значения удовлетворяют гарантиям, предоставляемым стандартом C, через `<climits>` и (`>= C ++ 11`) `<cinttypes>`.

- `std::numeric_limits<signed char>::min()` равно `SCHAR_MIN`, которое меньше или равно -127.
- `std::numeric_limits<signed char>::max()` равно `SCHAR_MAX`, которое больше или равно 127.
- `std::numeric_limits<unsigned char>::max()` равно `UCHAR_MAX`, который больше или равен 255.
- `std::numeric_limits<short>::min()` равно `SHRT_MIN`, которое меньше или равно -32767.
- `std::numeric_limits<short>::max()` равно `SHRT_MAX`, которое больше или равно 32767.
- `std::numeric_limits<unsigned short>::max()` равно `USHRT_MAX`, который больше или равен 65535.
- `std::numeric_limits<int>::min()` равно `INT_MIN`, которое меньше или равно -32767.
- `std::numeric_limits<int>::max()` равно `INT_MAX`, которое больше или равно 32767.
- `std::numeric_limits<unsigned int>::max()` равно `UINT_MAX`, которое больше или равно 65535.
- `std::numeric_limits<long>::min()` равно `LONG_MIN`, которое меньше или равно -2147483647.
- `std::numeric_limits<long>::max()` равно `LONG_MAX`, которое больше или равно 2147483647.
- `std::numeric_limits<unsigned long>::max()` равно `ULONG_MAX`, которое больше или равно 4294967295.

C ++ 11

- `std::numeric_limits<long long>::min()` равно `LLONG_MIN`, которое меньше или равно -9223372036854775807.
- `std::numeric_limits<long long>::max()` равно `LLONG_MAX`, который больше или равен 9223372036854775807.
- `std::numeric_limits<unsigned long long>::max()` равно `ULLONG_MAX`, которое больше или равно 18446744073709551615.

Для типов с плавающей запятой `T::max()` - максимальное конечное значение, а `min()` - минимальное положительное нормированное значение. Дополнительные члены предоставляются для типов с плавающей точкой, которые также определены для реализации, но удовлетворяют определенным гарантиям, предоставляемым стандартом C, через заголовок `<cmath>`.

- `digits10` дают число десятичных цифр точности.

- `std::numeric_limits<float>::digits10` равно `FLT_DIG` , который не менее 6.
- `std::numeric_limits<double>::digits10` равно `DBL_DIG` , который не менее 10.
- `std::numeric_limits<long double>::digits10` равно `LDBL_DIG` , который не менее 10.
- Член `min_exponent10` является минимальным отрицательным `E` , так что `10` к мощности `E` является нормальным.
 - `std::numeric_limits<float>::min_exponent10` равно `FLT_MIN_10_EXP` , что не более -37.
 - `std::numeric_limits<double>::min_exponent10` равно `DBL_MIN_10_EXP` , что не более -37.
 - `std::numeric_limits<long double>::min_exponent10` равно `LDBL_MIN_10_EXP` , что не более -37.
- Член `max_exponent10` - это максимум `E` такой, что `10` до степени `E` конечен.
 - `std::numeric_limits<float>::max_exponent10` равно `FLT_MAX_10_EXP` , который не менее 37.
 - `std::numeric_limits<double>::max_exponent10` равно `DBL_MAX_10_EXP` , который не менее 37.
 - `std::numeric_limits<long double>::max_exponent10` равно `LDBL_MAX_10_EXP` , который составляет не менее 37.
- Если элемент `is_iec559` имеет значение `true` , тип соответствует IEC 559 / IEEE 754 , и поэтому его диапазон определяется этим стандартом.

Представление значений типов с плавающей запятой

Стандарт требует, чтобы `long double` обеспечивал как минимум такую же точность, как `double` , которая обеспечивает как минимум такую же точность, как `float` ; и что `long double` может представлять любое значение, которое может представлять `double` , а `double` может представлять любое значение, которое может представлять `float` . Однако детали представления, однако, определены в реализации.

Для типа с плавающей точкой `T` `std::numeric_limits<T>::radix` указывает базовый знак, используемый представлением `T`

Если `std::numeric_limits<T>::is_iec559` истинно, то представление `T` соответствует одному из форматов, определенных IEC 559 / IEEE 754.

Переполнение при преобразовании из целого числа в целое число со знаком

Когда целое число под знаком или без знака преобразуется в целочисленный тип со знаком, а его значение не представляется в целевом типе, полученное значение определяется реализацией. Пример:

```
// Suppose that on this implementation, the range of signed char is -128 to +127 and
// the range of unsigned char is 0 to 255
int x = 12345;
signed char sc = x;    // sc has an implementation-defined value
unsigned char uc = x;  // uc is initialized to 57 (i.e., 12345 modulo 256)
```

Основной тип (и, следовательно, размер) перечисления

Если базовый тип явно не указан для типа нумерации без регистрации, он определяется определенным образом.

```
enum E {  
    RED,  
    GREEN,  
    BLUE,  
};  
using T = std::underlying_type<E>::type; // implementation-defined
```

Однако стандарт требует, чтобы базовый тип перечисления был не больше, чем `int` если только `int` и `unsigned int` не могут представлять все значения перечисления. Поэтому в вышеприведенном коде `T` может быть `int`, `unsigned int` или `short`, но не `long long`, чтобы дать несколько примеров.

Обратите внимание, что перечисление имеет тот же размер (который возвращается `sizeof`) как его базовый тип.

Прочитайте Поведение, определяемое реализацией онлайн:

<https://riptutorial.com/ru/cplusplus/topic/1363/поведение--определяемое-реализацией>

глава 81: Поиск зависимого имени аргумента

Examples

Какие функции найдены

Функции обнаруживаются путем первого сбора набора «*связанных классов*» и «*связанных пространств имен*», которые включают в себя одну или несколько из следующих, в зависимости от типа аргумента *T*. Во-первых, давайте покажем правила для имен классов, перечислений и классов.

- Если *T* - вложенный класс, перечисление участника, то его окружающий класс.
- Если *T* - перечисление (оно *также* может быть членом класса!), Самое внутреннее пространство имён.
- Если *T* - класс (он *также* может быть вложен!), Все его базовые классы и сам класс. Самое внутреннее пространство имен всех связанных классов.
- Если *T* является `ClassTemplate<TemplateArguments>` (это *также* класс!), Классы и пространства имен, связанные с аргументами типа шаблона, пространство имен любого аргумента шаблона шаблона и окружающий класс любого аргумента шаблона шаблона, если аргумент шаблона шаблон участника.

Теперь есть несколько правил для встроенных типов

- Если *T* - указатель на *U* или массив *U*, классы и пространства имен, связанные с *U*. Пример: `void (*fptr)(A); f(fptr);`, включает пространства имен и классы, связанные с `void(A)` (см. следующее правило).
- Если *T* - это тип функции, классы и пространства имен связаны с параметрами и типами возврата. Пример: `void(A)` будет включать пространства имен и классы, связанные с *A*.
- Если *T* является указателем на член, классы и пространства имен, связанные с типом члена (могут применяться как к указателю на функции-члену, так и указателю на элемент данных!). Пример: `BA::*p; void (A::*pf)(B); f(p); f(pf);` включает пространства имен и классы, связанные с *A*, *B*, `void(B)` (который применяет маркер выше для типов функций).

Все функции и шаблоны во всех связанных пространствах имен находятся путем поиска зависимых от аргументов. Кроме того, найдены функции имени пространства-области видимости в ассоциированных классах, которые обычно не видны. Однако использование директив игнорируется.

Все следующие примерные вызовы действительны, не определяя *f* именем имени в вызове.


```
namespace A {
    struct Z { };
    namespace I { void g(Z); }
    using namespace I;

    struct X { struct Y { }; friend void f(Y) { } };
    void f(X p) { }
    void f(std::shared_ptr<X> p) { }
}

// example calls
f(A::X());
f(A::X::Y());
f(std::make_shared<A::X>());

g(A::Z()); // invalid: "using namespace I;" is ignored!
```

Прочитайте Поиск зависимого имени аргумента онлайн:

<https://riptutorial.com/ru/cplusplus/topic/5163/поиск-зависимого-имени-аргумента>

глава 82: Полиморфизм

Examples

Определение полиморфных классов

Типичным примером является абстрактный класс формы, который затем может быть выведен на квадраты, круги и другие конкретные формы.

Родительский класс:

Начнем с полиморфного класса:

```
class Shape {
public:
    virtual ~Shape() = default;
    virtual double get_surface() const = 0;
    virtual void describe_object() const { std::cout << "this is a shape" << std::endl; }

    double get_doubled_surface() const { return 2 * get_surface(); }
};
```

Как читать это определение?

- Вы можете определить полиморфное поведение, введя функции-члены с ключевым словом `virtual`. Здесь `get_surface()` и `describe_object()`, очевидно, будут выполняться по-разному для квадрата, чем для круга. Когда функция вызывается на объект, функция, соответствующая реальному классу объекта, будет определена во время выполнения.
- Нет смысла определять `get_surface()` для абстрактной формы. Вот почему за функцией следует `= 0`. Это означает, что функция является *чистой виртуальной функцией*.
- Полиморфный класс должен всегда определять виртуальный деструктор.
- Вы можете определить не виртуальные функции-члены. Когда эта функция будет вызываться для объекта, функция будет выбрана в зависимости от класса, используемого во время компиляции. Здесь `get_double_surface()` определяется таким образом.
- Класс, содержащий хотя бы одну чистую виртуальную функцию, является абстрактным классом. Абстрактные классы не могут быть созданы. У вас могут быть только указатели или ссылки абстрактного типа класса.

Производные классы

Как только полиморфный базовый класс определен, вы можете его получить. Например:

```
class Square : public Shape {
    Point top_left;
    double side_length;
public:
    Square (const Point& top_left, double side)
        : top_left(top_left), side_length(side_length) {}

    double get_surface() override { return side_length * side_length; }
    void describe_object() override {
        std::cout << "this is a square starting at " << top_left.x << ", " << top_left.y
            << " with a length of " << side_length << std::endl;
    }
};
```

Некоторые объяснения:

- Вы можете определить или переопределить любую из виртуальных функций родительского класса. Тот факт, что функция была виртуальной в родительском классе, делает ее виртуальной в производном классе. Не нужно снова указывать компилятору ключевое слово `virtual`. Но рекомендуется добавить ключевое слово `override` в конце объявления функции, чтобы предотвратить тонкие ошибки, вызванные незаметными изменениями в сигнатуре функции.
- Если все чистые виртуальные функции родительского класса определены, вы можете создавать объекты для этого класса, иначе он также станет абстрактным классом.
- Вы не обязаны отменять все виртуальные функции. Вы можете сохранить версию родителя, если это соответствует вашим потребностям.

Пример создания экземпляра

```
int main() {

    Square square(Point(10.0, 0.0), 6); // we know it's a square, the compiler also
    square.describe_object();
    std::cout << "Surface: " << square.get_surface() << std::endl;

    Circle circle(Point(0.0, 0.0), 5);

    Shape *ps = nullptr; // we don't know yet the real type of the object
    ps = &circle;         // it's a circle, but it could as well be a square
    ps->describe_object();
    std::cout << "Surface: " << ps->get_surface() << std::endl;
}
```

Безопасное понижение

Предположим, что у вас есть указатель на объект полиморфного класса:

```
Shape *ps; // see example on defining a polymorphic class
ps = get_a_new_random_shape(); // if you don't have such a function yet, you
                                // could just write ps = new Square(0.0,0.0, 5);
```

нисходящим было бы бросить из общей полиморфной `Shape` до одной из ее производной и более конкретной формы, такой как `Square` или `Circle`.

Почему сваливать?

В большинстве случаев вам не нужно будет знать, каков реальный тип объекта, так как виртуальные функции позволяют вам манипулировать вашим объектом независимо от его типа:

```
std::cout << "Surface: " << ps->get_surface() << std::endl;
```

Если вам не нужен нисходящий, ваш дизайн будет идеальным.

Однако иногда вам может потребоваться понижение. Типичным примером является то, что вы хотите вызывать не виртуальную функцию, которая существует только для дочернего класса.

Рассмотрим, например, круги. Только круги имеют диаметр. Таким образом, класс будет определяться как:

```
class Circle: public Shape { // for Shape, see example on defining a polymorphic class
    Point center;
    double radius;
public:
    Circle (const Point& center, double radius)
        : center(center), radius(radius) {}

    double get_surface() const override { return r * r * M_PI; }

    // this is only for circles. Makes no sense for other shapes
    double get_diameter() const { return 2 * r; }
};
```

Функция `get_diameter()` существует только для кругов. Он не был определен для объекта `Shape`:

```
Shape* ps = get_any_shape();
ps->get_diameter(); // OUCH !!! Compilation error
```

Как сбить?

Если вы точно знаете, что `ps` указывает на круг, вы можете выбрать `static_cast`:

```
std::cout << "Diameter: " << static_cast<Circle*>(ps)->get_diameter() << std::endl;
```

Это сделает трюк. Но это очень рискованно: если `ps` появляется ничем иным, как `Circle` поведение вашего кода будет неопределенным.

Поэтому вместо того, чтобы играть в русскую рулетку, вы должны безопасно использовать `dynamic_cast`

. Это специально для полиморфных классов:

```
int main() {
    Circle circle(Point(0.0, 0.0), 10);
    Shape &shape = circle;

    std::cout << "The shape has a surface of " << shape.get_surface() << std::endl;

    //shape.get_diameter();    // OUCH !!! Compilation error

    Circle *pc = dynamic_cast<Circle*>(&shape); // will be nullptr if ps wasn't a circle
    if (pc)
        std::cout << "The shape is a circle of diameter " << pc->get_diameter() << std::endl;
    else
        std::cout << "The shape isn't a circle !" << std::endl;
}
```

Обратите внимание, что `dynamic_cast` невозможен для класса, который не является полиморфным. Вам понадобится хотя бы одна виртуальная функция в классе или ее родителях, чтобы иметь возможность использовать ее.

Полиморфизм и деструкторы

Если класс предназначен для использования полиморфно, при этом производные экземпляры хранятся в качестве базовых указателей / ссылок, его деструктор его базового класса должен быть либо `virtual` либо `protected`. В первом случае это приведет к уничтожению объекта, чтобы проверить `vtable`, автоматически вызывая правильный деструктор на основе динамического типа. В последнем случае уничтожение объекта с помощью указателя / ссылки базового класса отключено, и объект может быть удален только при явном трактовке его фактического типа.

```
struct VirtualDestructor {
    virtual ~VirtualDestructor() = default;
};

struct VirtualDerived : VirtualDestructor {};

struct ProtectedDestructor {
    protected:
    ~ProtectedDestructor() = default;
};

struct ProtectedDerived : ProtectedDestructor {
    ~ProtectedDerived() = default;
};

// ...

VirtualDestructor* vd = new VirtualDerived;
delete vd; // Looks up VirtualDestructor::~~VirtualDestructor() in vtable, sees it's
           // VirtualDerived::~~VirtualDerived(), calls that.

ProtectedDestructor* pd = new ProtectedDerived;
delete pd; // Error: ProtectedDestructor::~~ProtectedDestructor() is protected.
```

```
delete static_cast<ProtectedDerived*>(pd); // Good.
```

Обе эти практики гарантируют, что деструктор производного класса всегда будет вызываться на экземплярах производного класса, предотвращая утечку памяти.

Прочитайте Полиморфизм онлайн: <https://riptutorial.com/ru/cplusplus/topic/1717/полиморфизм>

глава 83: Пользовательские литералы

Examples

Пользовательские литералы с длинными двойными значениями

```
#include <iostream>

long double operator"" _km(long double val)
{
    return val * 1000.0;
}

long double operator"" _mi(long double val)
{
    return val * 1609.344;
}

int main()
{
    std::cout << "3 km = " << 3.0_km << " m\n";
    std::cout << "3 mi = " << 3.0_mi << " m\n";
    return 0;
}
```

Результат этой программы следующий:

```
3 km = 3000 m
3 mi = 4828.03 m
```

Стандартные пользовательские литералы для продолжительности

C++ 14

Те последующие пользовательские литералы объявляются в namespace

`std::literals::chrono_literals`, где оба `literals` и `chrono_literals` являются **внутренними пространствами имен**. Доступ к этим операторам можно получить с `using namespace std::literals using namespace std::chrono_literals` и `using namespace std::literals::chrono_literals`.

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::literals::chrono_literals;

    std::chrono::nanoseconds t1 = 600ns;
    std::chrono::microseconds t2 = 42us;
}
```

```

std::chrono::milliseconds t3 = 51ms;
std::chrono::seconds t4 = 61s;
std::chrono::minutes t5 = 88min;
auto t6 = 2 * 0.5h;

auto total = t1 + t2 + t3 + t4 + t5 + t6;

std::cout.precision(13);
std::cout << total.count() << " nanoseconds" << std::endl; // 8941051042600 nanoseconds
std::cout << std::chrono::duration_cast<std::chrono::hours>(total).count()
    << " hours" << std::endl; // 2 hours
}

```

Стандартные пользовательские литералы для строк

C++ 14

Те, кто соответствует строковым пользовательским литералам, объявляются в `namespace std::literals::string_literals`, где оба `literals` и `string_literals` являются **внутренними пространствами имен**. Доступ к этим операторам можно получить с `using namespace std::literals, using namespace std::string_literals` и `using namespace std::literals::string_literals`.

```

#include <codecvt>
#include <iostream>
#include <locale>
#include <string>

int main()
{
    using namespace std::literals::string_literals;

    std::string s = "hello world"s;
    std::u16string s16 = u"hello world"s;
    std::u32string s32 = U"hello world"s;
    std::wstring ws = L"hello world"s;

    std::cout << s << std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> utf16conv;
    std::cout << utf16conv.to_bytes(s16) << std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char32_t>, char32_t> utf32conv;
    std::cout << utf32conv.to_bytes(s32) << std::endl;

    std::wcout << ws << std::endl;
}

```

Замечания:

Литеральная строка может содержать `\0`

```

std::string s1 = "foo\0\0bar"; // constructor from C-string: results in "foo"s
std::string s2 = "foo\0\0bar"s; // That string contains 2 '\0' in its middle

```


Стандартные пользовательские литералы для сложных

C++ 14

Те, кто следуют за сложными пользовательскими литералами, объявляются в `namespace std::literals::complex_literals`, где оба `literals` и `complex_literals` являются **встроенными пространствами имен**. Доступ к этим операторам можно получить с `using namespace std::literals using namespace std::complex_literals`, `using namespace std::literals::complex_literals using namespace std::complex_literals` и `using namespace std::literals::complex_literals`.

```
#include <complex>
#include <iostream>

int main()
{
    using namespace std::literals::complex_literals;

    std::complex<double> c = 2.0 + 1i;          // {2.0, 1.}
    std::complex<float> cf = 2.0f + 1if;       // {2.0f, 1.f}
    std::complex<long double> cl = 2.0L + 1il; // {2.0L, 1.L}

    std::cout << "abs" << c << " = " << abs(c) << std::endl; // abs(2,1) = 2.23607
    std::cout << "abs" << cf << " = " << abs(cf) << std::endl; // abs(2,1) = 2.23607
    std::cout << "abs" << cl << " = " << abs(cl) << std::endl; // abs(2,1) = 2.23607
}
```

Самодельный пользовательский литерал для двоичного

Несмотря на то, что вы можете записать двоичное число в C++ 14, например:

```
int number = 0b0001'0101; // ==21
```

здесь идет знаменитый пример с самодельной реализацией для двоичных чисел:

Примечание. Вся программа расширения шаблонов запускается во время компиляции.

```
template< char FIRST, char... REST > struct binary
{
    static_assert( FIRST == '0' || FIRST == '1', "invalid binary digit" );
    enum { value = ( ( FIRST - '0' ) << sizeof...(REST) ) + binary<REST...>::value };
};

template<> struct binary<'0'> { enum { value = 0 }; };
template<> struct binary<'1'> { enum { value = 1 }; };

// raw literal operator
template< char... LITERAL > inline
constexpr unsigned int operator "" _b() { return binary<LITERAL...>::value ; }

// raw literal operator
template< char... LITERAL > inline
constexpr unsigned int operator "" _B() { return binary<LITERAL...>::value ; }
```

```
#include <iostream>

int main()
{
    std::cout << 10101_B << ", " << 011011000111_b << '\n' ; // prints 21, 1735
}
```

Прочитайте Пользовательские литералы онлайн:

<https://riptutorial.com/ru/cplusplus/topic/2745/пользовательские-литералы>

глава 84: Поля бит

Вступление

Бит-поля плотно упаковывают структуры C и C++ для уменьшения размера. Это кажется безболезненным: укажите количество бит для членов, а компилятор выполняет работу со-смешивающих битов. Ограничением является невозможность принять адрес члена битового поля, поскольку он хранится совместно. `sizeof()` также не разрешено.

Стоимость битных полей - это более медленный доступ, так как память должна быть восстановлена и побитовые операции применяются для извлечения или изменения значений элементов. Эти операции также добавляют к размеру исполняемого файла.

замечания

Насколько дороги побитовые операции? Предположим, что простая небитовая структура поля:

```
struct foo {
    unsigned x;
    unsigned y;
}
static struct foo my_var;
```

В более позднем коде:

```
my_var.y = 5;
```

Если `sizeof (unsigned) == 4`, то `x` сохраняется в начале структуры, а `y` хранится в 4 байта. Сгенерированный код сборки может напоминать:

```
loada register1,#myvar      ; get the address of the structure
storei register1[4],#0x05    ; put the value '5' at offset 4, e.g., set y=5
```

Это просто, потому что `x` не смешивается с `y`. Но представьте себе переопределение структуры с битовыми полями:

```
struct foo {
    unsigned x : 4; /* Range 0-0x0f, or 0 through 15 */
    unsigned y : 4;
}
```

И `x` и `y` будут выделены 4 бита, разделяя один байт. Таким образом, структура занимает 1 байт, а не 8. Рассмотрим сборку, чтобы установить `y` теперь, предполагая, что она заканчивается в верхнем полубайте:

```

loada  register1,#myvar      ; get the address of the structure
loadb  register2,register1[0] ; get the byte from memory
andb   register2,#0x0f      ; zero out y in the byte, leaving x alone
orb    register2,#0x50      ; put the 5 into the 'y' portion of the byte
stb    register1[0],register2 ; put the modified byte back into memory

```

Это может быть хорошим компромиссом, если у нас есть тысячи или миллионы этих структур, и это помогает хранить память в кеше или препятствует обмену - или может раздуть исполняемый файл, чтобы ухудшить эти проблемы и медленную обработку. Как и все, используйте здравый смысл.

Использование драйвера устройства: избегайте бит-полей как умной стратегии реализации драйверов устройств. Битовые поля хранения не обязательно согласованы между компиляторами, что делает такие реализации не переносимыми. Значение read-modify-write для установки значений может не делать то, что ожидают устройства, вызывая неожиданное поведение.

Examples

Декларация и использование

```

struct FileAttributes
{
    unsigned int ReadOnly: 1;
    unsigned int Hidden: 1;
};

```

Здесь каждое из этих двух полей будет занимать 1 бит в памяти. Он определяется выражением : 1 после имен переменных. Базовый тип битового поля может быть любым интегральным типом (8-битный int до 64-битного int). Рекомендуется использовать `unsigned` тип, иначе могут возникнуть сюрпризы.

Если требуется больше бит, замените «1» на количество бит. Например:

```

struct Date
{
    unsigned int Year : 13; // 2^13 = 8192, enough for "year" representation for long time
    unsigned int Month: 4;  // 2^4 = 16, enough to represent 1-12 month values.
    unsigned int Day: 5;    // 32
};

```

Вся структура использует всего 22 бита, и с обычными настройками компилятора `sizeof` этой структуры будет 4 байта.

Использование довольно простое. Просто объявите переменную и используйте ее как обычную структуру.

```
Date d;
```

```
d.Year = 2016;  
d.Month = 7;  
d.Day = 22;  
  
std::cout << "Year:" << d.Year << std::endl <<  
    "Month:" << d.Month << std::endl <<  
    "Day:" << d.Day << std::endl;
```

Прочитайте Поля бит онлайн: <https://riptutorial.com/ru/cplusplus/topic/2710/поля-бит>

глава 85: Потоки C ++

замечания

Конструктор по умолчанию `std::istream_iterator` создает итератор, который представляет конец потока. Таким образом, `std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(), ...)` означает копирование с текущей позиции в `ifs` до конца.

Examples

Строковые потоки

`std::ostringstream` - это класс, объекты которого выглядят как поток вывода (то есть вы можете писать им через `operator<<`), но на самом деле сохраняете результаты записи и предоставляете их в виде потока.

Рассмотрим следующий короткий код:

```
#include <sstream>
#include <string>

using namespace std;

int main()
{
    ostringstream ss;
    ss << "the answer to everything is " << 42;
    const string result = ss.str();
}
```

Линия

```
ostringstream ss;
```

создает такой объект. Этот объект сначала обрабатывается как обычный поток:

```
ss << "the answer to everything is " << 42;
```

После этого, однако, полученный поток можно получить следующим образом:

```
const string result = ss.str();
```

(`result` строки будет равен `"the answer to everything is 42"`).

Это в основном полезно, когда у нас есть класс, для которого была определена

сериализация потока, и для которого мы хотим получить строчную форму. Например, предположим, что у нас есть класс

```
class foo
{
    // All sort of stuff here.
};

ostream &operator<<(ostream &os, const foo &f);
```

Чтобы получить строковое представление объекта `foo`,

```
foo f;
```

мы могли бы использовать

```
ostringstream ss;
ss << f;
const string result = ss.str();
```

Тогда `result` содержит строковое представление объекта `foo`.

Чтение файла до конца

Чтение текстового файла по очереди

Правильный способ чтения текстового файла по очереди до конца обычно не ясен из документации `ifstream`. Давайте рассмотрим некоторые распространенные ошибки, сделанные начинающими программистами на C++, и правильный способ прочитать файл.

Строки без пробельных символов

Для простоты предположим, что каждая строка в файле не содержит пробельных символов.

`ifstream` имеет `operator bool()`, который возвращает `true`, когда поток не имеет ошибок и готов к чтению. Более того, `ifstream::operator >>` возвращает ссылку на сам поток, поэтому мы можем читать и проверять EOF (а также на ошибки) в одной строке с очень элегантным синтаксисом:

```
std::ifstream ifs("1.txt");
std::string s;
while(ifs >> s) {
    std::cout << s << std::endl;
}
```

Строки с пробельными символами

`ifstream::operator >>` считывает поток до появления любого символа пробела, поэтому приведенный выше код будет печатать слова из строки на отдельных строках. Чтобы прочитать все до конца строки, используйте `std::getline` вместо `ifstream::operator >>`. `getline` возвращает ссылку на поток, с которым он работал, поэтому доступен такой же синтаксис:

```
while(std::getline(ifs, s)) {
    std::cout << s << std::endl;
}
```

Очевидно, `std::getline` также следует использовать для чтения однострочного файла до конца.

Чтение файла в буфер сразу

Наконец, давайте прочитаем файл с самого начала и до конца, не останавливаясь ни на каком персонаже, включая пробелы и символы новой строки. Если мы знаем, что точный размер файла или верхняя граница длины допустимы, мы можем изменить размер строки, а затем читать:

```
s.resize(100);
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    s.begin());
```

В противном случае нам нужно вставить каждый символ в конец строки, поэтому

`std::back_inserter` - это то, что нам нужно:

```
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    std::back_inserter(s));
```

В качестве альтернативы, можно инициализировать коллекцию данными потока, используя конструктор с аргументами диапазона итератора:

```
std::vector v(std::istreambuf_iterator<char>(ifs),
    std::istreambuf_iterator<char>());
```

Обратите внимание, что эти примеры также применимы, если `ifs` открыт как двоичный файл:

```
std::ifstream ifs("1.txt", std::ios::binary);
```


Копирование потоков

Файл может быть скопирован в другой файл с потоками и итераторами:

```
std::ofstream ofs("out.file");
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
          std::ostream_iterator<char>(ofs));
ofs.close();
```

или перенаправляется на любой другой тип потока с совместимым интерфейсом. Например, сетевой поток Boost.Asio:

```
boost::asio::ip::tcp::iostream stream;
stream.connect("example.com", "http");
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
          std::ostream_iterator<char>(stream));
stream.close();
```

Массивы

Поскольку итераторы могут рассматриваться как обобщение указателей, контейнеры STL в приведенных выше примерах могут быть заменены на собственные массивы. Вот как разбирать числа в массиве:

```
int arr[100];
std::copy(std::istream_iterator<char>(ifs), std::istream_iterator<char>(), arr);
```

Остерегайтесь переполнения буфера, поскольку массивы не могут быть изменены «на лету» после их выделения. Например, если вышеприведенный код будет содержать файл, содержащий более 100 целых чисел, он попытается написать за пределами массива и выполнить неопределенное поведение.

Печать коллекций с помощью `iostream`

Основная печать

`std::ostream_iterator` позволяет печатать содержимое контейнера STL в любой выходной поток без явных циклов. Второй аргумент конструктора `std::ostream_iterator` устанавливает разделитель. Например, следующий код:

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " ! "));
```

распечатает

```
1 ! 2 ! 3 ! 4 !
```

Неявный тип

`std::ostream_iterator` позволяет `std::ostream_iterator` тип содержимого контейнера.

Например, давайте настроим `std::cout` для печати значений с плавающей запятой с 3 цифрами после десятичной точки:

```
std::cout << std::setprecision(3);  
std::fixed(std::cout);
```

и создайте экземпляр `std::ostream_iterator` с `float`, а содержащиеся значения останутся `int`:

```
std::vector<int> v = {1,2,3,4};  
std::copy(v.begin(), v.end(), std::ostream_iterator<float>(std::cout, " ! "));
```

поэтому приведенный выше код дает

```
1.000 ! 2.000 ! 3.000 ! 4.000 !
```

несмотря на то, что `std::vector` имеет `int` S.

Поколение и трансформация

`std::generate`, `std::generate_n` и `std::transform` - это очень мощный инструмент для манипуляции данными на лету. Например, имея вектор:

```
std::vector<int> v = {1,2,3,4,8,16};
```

мы можем легко распечатать логическое значение оператора «x is even» для каждого элемента:

```
std::boolalpha(std::cout); // print booleans alphabetically  
std::transform(v.begin(), v.end(), std::ostream_iterator<bool>(std::cout, " "),  
[](int val) {  
    return (val % 2) == 0;  
});
```

или напечатать квадрат элемента:

```
std::transform(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "),
```

```
[](int val) {  
    return val * val;  
});
```

Печать N случайных чисел, разделенных пробелами:

```
const int N = 10;  
std::generate_n(std::ostream_iterator<int>(std::cout, " "), N, std::rand);
```

Массивы

Как и в разделе о чтении текстовых файлов, почти все эти соображения могут быть применены к встроенным массивам. Например, давайте напечатаем квадратные значения из собственного массива:

```
int v[] = {1,2,3,4,8,16};  
std::transform(v, std::end(v), std::ostream_iterator<int>(std::cout, " "),  
[](int val) {  
    return val * val;  
});
```

Разбор файлов

Разбор файлов в контейнеры STL

`istream_iterator` S очень полезны для чтения последовательностей чисел или других анализируемых данных в контейнеры STL без явных циклов кода.

Использование явного размера контейнера:

```
std::vector<int> v(100);  
std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),  
v.begin());
```

или с вставкой итератора:

```
std::vector<int> v;  
std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),  
std::back_inserter(v));
```

Обратите внимание, что числа во входном файле могут быть разделены на любое количество символов пробела и символов новой строки.

Разбор гетерогенных текстовых таблиц

Поскольку `istream::operator>>` считывает текст до символа пробела, он может использоваться во `while` условия для обработки сложных таблиц данных. Например, если у нас есть файл с двумя действительными числами, за которым следует строка (без пробелов) в каждой строке:

```
1.12 3.14 foo
2.1 2.2 barr
```

он может быть проанализирован следующим образом:

```
std::string s;
double a, b;
while(ifs >> a >> b >> s) {
    std::cout << a << " " << b << " " << s << std::endl;
}
```

преобразование

Любая функция управления диапазоном может использоваться с диапазонами `std::istream_iterator`. Одним из них является `std::transform`, который позволяет обрабатывать данные «на лету». Например, давайте прочитаем целочисленные значения, умножим их на 3.14 и сохраним результат в контейнер с плавающей запятой:

```
std::vector<double> v(100);
std::transform(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
v.begin(),
[](int val) {
    return val * 3.14;
});
```

Прочитайте Потоки C++ онлайн: <https://riptutorial.com/ru/cplusplus/topic/7660/потоки-cplusplus>

глава 86: Потолочные манипуляторы

Вступление

Манипуляторы - это специальные вспомогательные функции, которые помогают управлять входными и выходными потоками с помощью `operator >>` или `operator <<`.

Все они могут быть включены в `#include <iomanip>`.

замечания

Манипуляторы могут использоваться другим способом. Например:

1. `os.width(n);` равно `os << std::setw(n);`
`is.width(n);` равно `to is >> std::setw(n);`
2. `os.precision(n);` равно `os << std::setprecision(n);`
`is.precision(n);` равно `to is >> std::setprecision(n);`
3. `os.setfill(c);` равно `os << std::setfill(c);`
4. `str >> std::setbase(base);` или `str << std::setbase(base);` равно

```
str.setf(base == 8 ? std::ios_base::oct :  
        base == 10 ? std::ios_base::dec :  
        base == 16 ? std::ios_base::hex :  
        std::ios_base::fmtflags(0),  
        std::ios_base::basefield);
```

5. `os.setf(std::ios_base::flag);` равно `os << std::flag;`
`is.setf(std::ios_base::flag);` равно `to is >> std::flag;`
`os.unsetf(std::ios_base::flag);` равно `os << std::no ## flag;`
`is.unsetf(std::ios_base::flag);` equals `to is >> std::no ## flag;`
(где `##` - оператор конкатенации)
на следующий `flag` `C: boolalpha, showbase, showpoint, showpos, skipws, uppercase`.

6. `std::ios_base::basefield`

.

Для `flag` `S`: `dec` , `hex` И `oct` :

- `os.setf(std::ios_base::flag, std::ios_base::basefield);` равно `os << std::flag;`
`is.setf(std::ios_base::flag, std::ios_base::basefield);` равно `is >> std::flag;`
(1)
- `str.unsetf(std::ios_base::flag, std::ios_base::basefield);` равно
`str.setf(std::ios_base::fmtflags(0), std::ios_base::basefield);`
(2)

7. `std::ios_base::adjustfield` .

Для `flag` `S`: `left` , `right` И `internal` :

- `os.setf(std::ios_base::flag, std::ios_base::adjustfield);` равно `os << std::flag;`
`is.setf(std::ios_base::flag, std::ios_base::adjustfield);` равно `is >> std::flag;`
(1)
- `str.unsetf(std::ios_base::flag, std::ios_base::adjustfield);` равно
`str.setf(std::ios_base::fmtflags(0), std::ios_base::adjustfield);`
(2)

(1) Если флаг соответствующего поля, ранее установленного, уже отключен `unsetf` .

(2) Если установлен `flag` .

8. `std::ios_base::floatfield` .

- `os.setf(std::ios_base::flag, std::ios_base::floatfield);` равно `os << std::flag;`
`is.setf(std::ios_base::flag, std::ios_base::floatfield);` равно `is >> std::flag;`
Для `flag` `S`: `fixed` И `scientific` .
- `os.setf(std::ios_base::fmtflags(0), std::ios_base::floatfield);` равно `os << std::defaultfloat;`
`is.setf(std::ios_base::fmtflags(0), std::ios_base::floatfield);` равно `is >> std::defaultfloat;`

9. `str.setf(std::ios_base::fmtflags(0), std::ios_base::flag);` равно `str.unsetf(std::ios_base::flag)`

Для `flag` `S`: `basefield` , `adjustfield` , `floatfield` .

- ## 10.
- `os.setf(mask)` равна `os << setiosflags(mask);`
`is.setf(mask)` равна `is >> setiosflags(mask);`
`os.unsetf(mask)` равна `os << resetiosflags(mask);`
`is.unsetf(mask)` равно `is >> resetiosflags(mask);`

Для почти всей `mask` типа `std::ios_base::fmtflags` .

Examples

Потолочные манипуляторы

`std::boolalpha` и `std::noboolalpha` - переключаться между текстовым и числовым представлением булевых.

```
std::cout << std::boolalpha << 1;
// Output: true

std::cout << std::noboolalpha << false;
// Output: 0

bool boolValue;
std::cin >> std::boolalpha >> boolValue;
std::cout << "Value \"" << std::boolalpha << boolValue
          << "\" was parsed as " << std::noboolalpha << boolValue;
// Input: true
// Output: Value "true" was parsed as 0
```

`std::showbase` и `std::noshowbase` - контролирует, используется ли префикс, указывающий числовую базу.

`std::dec` (десятичный), `std::hex` (шестнадцатеричный) и `std::oct` (восьмеричный) - используются для изменения базы для целых чисел.

```
#include <sstream>

std::cout << std::dec << 29 << ' - '
          << std::hex << 29 << ' - '
          << std::showbase << std::oct << 29 << ' - '
          << std::noshowbase << 29 << '\n';

int number;
std::istringstream("3B") >> std::hex >> number;
std::cout << std::dec << 10;
// Output: 22 - 1D - 35 - 035
// 59
```

Значения по умолчанию: `std::ios_base::noshowbase` и `std::ios_base::dec` .

Если вы хотите больше узнать о `std::istringstream` просмотрите заголовок `< sstream >` .

`std::uppercase` `std::nouppercase` и `std::nouppercase` - управляют ли символы верхнего регистра в выводе с плавающей запятой и шестнадцатеричным целым числом. Не влияют на входные потоки.

```
std::cout << std::hex << std::showbase
          << "0x2a with nouppercase: " << std::nouppercase << 0x2a << '\n'
          << "1e-10 with uppercase: " << std::uppercase << 1e-10 << '\n'
}
// Output: 0x2a with nouppercase: 0x2a
// 1e-10 with uppercase: 1E-10
```

По умолчанию используется `std::nouppercase` .

`std::setw(n)` - изменяет ширину следующего поля ввода / вывода на ровно `n` .

Свойство `width` `n` сбрасывается до 0 когда вызываются некоторые функции (полный список [здесь](#)).

```
std::cout << "no setw:" << 51 << '\n'
          << "setw(7): " << std::setw(7) << 51 << '\n'
          << "setw(7), more output: " << 13
          << std::setw(7) << std::setfill('*') << 67 << ' ' << 94 << '\n';

char* input = "Hello, world!";
char arr[10];
std::cin >> std::setw(6) >> arr;
std::cout << "Input from \"Hello, world!\" with setw(6) gave \"" << arr << "\"\n";

// Output: 51
// setw(7):      51
// setw(7), more output: 13*****67 94

// Input: Hello, world!
// Output: Input from "Hello, world!" with setw(6) gave "Hello"
```

По умолчанию используется `std::setw(0)` .

`std::left` , `std::right` и `std::internal` - изменение позиции по умолчанию для символов заполнения путем установки `std::ios_base::adjustfield` в `std::ios_base::left` , `std::ios_base::right` и `std::ios_base::internal` соответственно. `std::left` и `std::right` применяются к любому выходу, `std::internal` - для целых чисел, с плавающей запятой и денежных средств. Не влияют на входные потоки.

```
#include <locale>
...

std::cout.imbue(std::locale("en_US.utf8"));

std::cout << std::left << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87 << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << " $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
```



```

        << std::setfill(' ') << std::put_money(367, false) << '\n';
// Output:
// flt: -9.87*****
// hex: 41*****
// $: $3.67*****
// usd: USD *3.67*****
// usd: $3.67

std::cout << std::internal << std::showbase << std::setfill('*')
        << "flt: " << std::setw(15) << -9.87 << '\n'
        << "hex: " << std::setw(15) << 41 << '\n'
        << " $: " << std::setw(15) << std::put_money(367, false) << '\n'
        << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
        << "usd: " << std::setw(15)
        << std::setfill(' ') << std::put_money(367, true) << '\n';
// Output:
// flt: -*****9.87
// hex: *****41
// $: $3.67*****
// usd: USD *****3.67
// usd: USD      3.67

std::cout << std::right << std::showbase << std::setfill('*')
        << "flt: " << std::setw(15) << -9.87 << '\n'
        << "hex: " << std::setw(15) << 41 << '\n'
        << " $: " << std::setw(15) << std::put_money(367, false) << '\n'
        << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
        << "usd: " << std::setw(15)
        << std::setfill(' ') << std::put_money(367, true) << '\n';
// Output:
// flt: *****-9.87
// hex: *****41
// $: *****$3.67
// usd: *****USD *3.67
// usd:      USD  3.67

```

По умолчанию `std::left`.

`std::fixed`, `std::scientific`, `std::hexfloat` [C ++ 11] и `std::defaultfloat` [C ++ 11] - изменение форматирования для ввода / вывода с плавающей запятой.

`std::fixed` устанавливает `std::ios_base::floatfield` в `std::ios_base::fixed`,
`std::scientific` - to `std::ios_base::scientific`,
`std::hexfloat` - to `std::ios_base::fixed` | `std::ios_base::scientific` и
`std::defaultfloat` - to `std::ios_base::fmtflags(0)`.

`fmtflags`

```

#include <sstream>
...

std::cout << '\n'
        << "The number 0.07 in fixed:      " << std::fixed << 0.01 << '\n'
        << "The number 0.07 in scientific: " << std::scientific << 0.01 << '\n'
        << "The number 0.07 in hexfloat:   " << std::hexfloat << 0.01 << '\n'

```

```

    << "The number 0.07 in default: " << std::defaultfloat << 0.01 << '\n';

double f;
std::istringstream is("0x1P-1022");
double f = std::strtod(is.str().c_str(), NULL);
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';

// Output:
// The number 0.01 in fixed:      0.070000
// The number 0.01 in scientific: 7.000000e-02
// The number 0.01 in hexfloat:   0x1.1eb851eb851ecp-4
// The number 0.01 in default:    0.07
// Parsing 0x1P-1022 as hex gives 2.22507e-308

```

По умолчанию используется `std::ios_base::fmtflags(0)` .

В некоторых компиляторах есть **ошибка**, которая вызывает

```

double f;
std::istringstream("0x1P-1022") >> std::hexfloat >> f;
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';
// Output: Parsing 0x1P-1022 as hex gives 0

```

`std::showpoint` и `std::noshowpoint` - контролировать, всегда ли десятичная точка включена в представление с плавающей запятой. Не влияют на входные потоки.

```

std::cout << "7.0 with showpoint: " << std::showpoint << 7.0 << '\n'
    << "7.0 with noshowpoint: " << std::noshowpoint << 7.0 << '\n';
// Output: 1.0 with showpoint: 7.00000
// 1.0 with noshowpoint: 7

```

По умолчанию используется `std::showpoint` .

`std::showpos` и `std::noshowpos` - контроль отображения знака + в неотрицательном выходе. Не влияют на входные потоки.

```

std::cout << "With showpos: " << std::showpos
    << 0 << ' ' << -2.718 << ' ' << 17 << '\n'
    << "Without showpos: " << std::noshowpos
    << 0 << ' ' << -2.718 << ' ' << 17 << '\n';
// Output: With showpos: +0 -2.718 +17
// Without showpos: 0 -2.718 17

```

По умолчанию, если `std::noshowpos` .

`std::unitbuf` , `std::nounitbuf` - контроль потока промывки после каждой операции. Не влияют на входной поток. `std::unitbuf` вызывает промывку.

`std::setbase(base)` - устанавливает числовую базу потока.

`std::setbase(8)` равно `std::ios_base::basefield` для `std::ios_base::oct` ,
`std::setbase(16)` - то `std::ios_base::hex` ,
`std::setbase(10)` - то `std::ios_base::dec` .

Если `base` другая, то 8 , 10 или 16 тогда `std::ios_base::basefield` устанавливает значение `std::ios_base::fmtflags(0)` . Это означает десятичный выход и вход, зависящий от префикса.

По умолчанию `std::ios_base::basefield` - `std::ios_base::dec` затем по умолчанию `std::setbase(10)` .

`std::setprecision(n)` - изменяет точность с плавающей запятой.

```
#include <cmath>
#include <limits>
...

typedef std::numeric_limits<long double> ld;
const long double pi = std::acos(-1.L);

std::cout << '\n'
    << "default precision (6):   pi: " << pi << '\n'
    << "                        10pi: " << 10 * pi << '\n'
    << "std::setprecision(4): 10pi: " << std::setprecision(4) << 10 * pi << '\n'
    << "                        10000pi: " << 10000 * pi << '\n'
    << "std::fixed:          10000pi: " << std::fixed << 10000 * pi << std::defaultfloat
<< '\n'
    << "std::setprecision(10):   pi: " << std::setprecision(10) << pi << '\n'
    << "max-1 radix precicion:   pi: " << std::setprecision(ld::digits - 1) << pi <<
'\n'
    << "max+1 radix precision:   pi: " << std::setprecision(ld::digits + 1) << pi <<
'\n'
    << "significant digits prec: pi: " << std::setprecision(ld::digits10) << pi << '\n';

// Output:
// default precision (6):   pi: 3.14159
//                        10pi: 31.4159
// std::setprecision(4): 10pi: 31.42
//                        10000pi: 3.142e+04
// std::fixed:          10000pi: 31415.9265
// std::setprecision(10):   pi: 3.141592654
// max-1 radix precicion:   pi:
3.14159265358979323851280895940618620443274267017841339111328125
// max+1 radix precision:   pi:
3.14159265358979323851280895940618620443274267017841339111328125
// significant digits prec: pi: 3.14159265358979324
```

По умолчанию используется `std::setprecision(6)` .

`std::setiosflags(mask)` И `std::resetiosflags(mask)` - установить и очистить флаги, указанные в `mask` `std::ios_base::fmtflags` .

```
#include <sstream>
...

std::istringstream in("10 010 10 010 10 010");
int num1, num2;

in >> std::oct >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::oct gives:  " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   8 8

in >> std::dec >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::dec gives:  " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:  10 10

in >> std::resetiosflags(std::ios_base::basefield) >> num1 >> num2;
std::cout << "Parsing \"10 010\" with autodetect gives: " << num1 << ' ' << num2 << '\n';
// Parsing "10 010" with autodetect gives: 10 8

std::cout << std::setiosflags(std::ios_base::hex |
                             std::ios_base::uppercase |
                             std::ios_base::showbase) << 42 << '\n';

// Output: 0X2A
```

`std::skipws` И `std::noskipws` - контроль пропуска пропущенных пробелов форматированными входными функциями. Не влияют на выходные потоки.

```
#include <sstream>
...

char c1, c2, c3;
std::istringstream("a b c") >> c1 >> c2 >> c3;
std::cout << "Default behavior: c1 = " << c1 << " c2 = " << c2 << " c3 = " << c3 << '\n';

std::istringstream("a b c") >> std::noskipws >> c1 >> c2 >> c3;
std::cout << "noskipws behavior: c1 = " << c1 << " c2 = " << c2 << " c3 = " << c3 << '\n';
// Output: Default behavior: c1 = a c2 = b c3 = c
// noskipws behavior: c1 = a c2 = c3 = b
```

По умолчанию используется `std::ios_base::skipws` .

`std::quoted(s[, delim[, escape]])` [C ++ 14] - вставляет или извлекает цитируемые строки со встроенными пространствами.

`s` - строка для вставки или извлечения.

`delim` - символ для использования в качестве разделителя " по умолчанию.

`escape` - символ, который будет использоваться как escape-символ, \ по умолчанию.

```
#include <sstream>
...

std::stringstream ss;
std::string in = "String with spaces, and embedded \"quotes\" too";
std::string out;

ss << std::quoted(in);
std::cout << "read in      [" << in << "]\n"
          << "stored as    [" << ss.str() << "]\n";

ss >> std::quoted(out);
std::cout << "written out [" << out << "]\n";
// Output:
// read in      [String with spaces, and embedded "quotes" too]
// stored as    ["String with spaces, and embedded \"quotes\" too"]
// written out  [String with spaces, and embedded "quotes" too]
```

Для получения дополнительной информации см. Ссылку выше.

Манипуляторы выходного потока

`std::ends` - вставляет нулевой символ `'\0'` для вывода потока. Более формально это объявление манипулятора выглядит так:

```
template <class charT, class traits>
std::basic_ostream<charT, traits>& ends(std::basic_ostream<charT, traits>& os);
```

и этот манипулятор помещает символ, вызывая `os.put(charT())` при использовании в выражении

```
os << std::ends;
```

`std::endl` и `std::flush` поток потока `flush` `out` вызывая `out.flush()` . Это приводит к немедленному производству продукции. Но `std::endl` вставляет конец строки `'\n'` символ перед промывкой.

```
std::cout << "First line." << std::endl << "Second line. " << std::flush
          << "Still second line.";
// Output: First line.
// Second line. Still second line.
```

`std::setfill(c)` - изменяет символ заполнения на `c` . Часто используется с `std::setw` .

```
std::cout << "\nDefault fill: " << std::setw(10) << 79 << '\n'
          << "setfill('#'): " << std::setfill('#')
          << std::setw(10) << 42 << '\n';
// Output:
// Default fill:          79
// setfill('#'): #####79
```

`std::put_money(mon[, intl])` [C ++ 11]. В выражении `out << std::put_money(mon, intl)` преобразуется денежное значение `mon` (`long double` или `std::basic_string` type) в его представление символа, как определено гранью `std::money_put` языка, в настоящее время проникнутого в `out` . Используйте международные валютные строки, если `intl true` , иначе используйте символы валюты.

```
long double money = 123.45;
// or std::string money = "123.45";

std::cout.imbue(std::locale("en_US.utf8"));
std::cout << std::showbase << "en_US: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: en_US: $1.23 or USD 1.23

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "ru_RU: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ru_RU: 1.23 руб or 1.23 RUB

std::cout.imbue(std::locale("ja_JP.utf8"));
std::cout << "ja_JP: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ja_JP: ¥123 or JPY 123
```

`std::put_time(tmb, fmt)` [C ++ 11] - форматирует и выводит значение даты / времени в `std::tm` соответствии с указанным форматом `fmt` .

`tmb` - указатель на структуру календаря `const std::tm*` полученную из `localtime()` или `gmtime()` .

`fmt` - указатель на строку с нулевым символом `const CharT*` определяющую формат преобразования.

```
#include <ctime>
...

std::time_t t = std::time(nullptr);
std::tm tm = *std::localtime(&t);

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "\nru_RU: " << std::put_time(&tm, "%c %Z") << '\n';
// Possible output:
// ru_RU: Вт 04 июл 2017 15:08:35 UTC
```

Для получения дополнительной информации см. Ссылку выше.

Манипуляторы входного потока

`std::ws` - потребляет ведущие пробелы во входном потоке. Он отличается от `std::skipws` .

```
#include <sstream>
...

std::string str;
std::istringstream(" \v\n\r\t   Wow!There   is no whitespaces!") >> std::ws >> str;
std::cout << str;
// Output: Wow!There   is no whitespaces!
```

`std::get_money(mon[, intl])` [C ++ 11]. В выражении `in >> std::get_money(mon, intl)` анализирует ввод символа как денежное значение, как указано в `std::money_get` аспекте локали, в настоящее время `std::money_get in` , и сохраняет значение в `mon` (`long double` или `std::basic_string`). Манипулятор ожидает, что *требуемые* международные валютные строки, если `intl true` , ожидает *дополнительных* символов валюты в противном случае.

```
#include <sstream>
#include <locale>
...

std::istringstream in("$1,234.56 2.22 USD  3.33");
long double v1, v2;
std::string v3;

in.imbue(std::locale("en_US.UTF-8"));
in >> std::get_money(v1) >> std::get_money(v2) >> std::get_money(v3, true);
if (in) {
    std::cout << std::quoted(in.str()) << " parsed as: "
              << v1 << ", " << v2 << ", " << v3 << '\n';
}
// Output:
// "$1,234.56 2.22 USD  3.33" parsed as: 123456, 222, 333
```

`std::get_time(tmb, fmt)` [C ++ 11] - анализирует значение даты / времени, хранящееся в `tmb` указанного формата `fmt` .

`tmb` - действительный указатель на объект `const std::tm*` где будет сохранен результат.
`fmt` - указатель на строку с нулевым символом `const CharT*` определяющую формат преобразования.

```
#include <sstream>
#include <locale>
...

std::tm t = {};
std::istringstream ss("2011-Februar-18 23:12:34");

ss.imbue(std::locale("de_DE.utf-8"));
ss >> std::get_time(&t, "%Y-%b-%d %H:%M:%S");
if (ss.fail()) {
    std::cout << "Parse failed\n";
}
```

```
else {  
    std::cout << std::put_time(&t, "%c") << '\n';  
}  
// Possible output:  
// Sun Feb 18 23:12:34 2011
```

Для получения дополнительной информации см. Ссылку выше.

Прочитайте Потолочные манипуляторы онлайн: <https://riptutorial.com/ru/cplusplus/topic/10699/потолочные-манипуляторы>

глава 87: Правило три, пять и ноль

Examples

Правило пяти

C++ 11

C++ 11 вводит две новые специальные функции-члены: конструктор перемещения и оператор назначения перемещения. По всем тем же причинам, по которым вы хотите следовать [правилу 3](#) в C++ 03, вы обычно хотите следовать правилу 5 в C++ 11: если класс требует ОДИН из пяти специальных функций-членов и если семантика перемещения желательны, то это, скорее всего, требует ВСЕ ПЯТЬ из них.

Обратите внимание, однако, что несоблюдение правила пяти обычно не считается ошибкой, но упущена возможность оптимизации, если до сих пор соблюдается правило трех. Если в случае, если компилятор обычно использует один оператор, если оператор-конструктор перемещения или оператор перемещения не доступны, вместо этого он будет использовать семантику копирования, что приведет к менее эффективной операции из-за ненужных операций копирования. Если семантика перемещения не нужна для класса, тогда нет необходимости объявлять конструктор перемещения или оператор присваивания.

Тот же пример, что и для правила трех:

```
class Person
{
    char* name;
    int age;

public:
    // Destructor
    ~Person() { delete [] name; }

    // Implement Copy Semantics
    Person(Person const& other)
        : name(new char[std::strlen(other.name) + 1])
        , age(other.age)
    {
        std::strcpy(name, other.name);
    }

    Person &operator=(Person const& other)
    {
        // Use copy and swap idiom to implement assignment.
        Person copy(other);
        swap(*this, copy);
        return *this;
    }
}
```

```

// Implement Move Semantics
// Note: It is usually best to mark move operators as noexcept
//       This allows certain optimizations in the standard library
//       when the class is used in a container.

Person(Person&& that) noexcept
    : name(nullptr)           // Set the state so we know it is undefined
    , age(0)
{
    swap(*this, that);
}

Person& operator=(Person&& that) noexcept
{
    swap(*this, that);
    return *this;
}

friend void swap(Person& lhs, Person& rhs) noexcept
{
    std::swap(lhs.name, rhs.name);
    std::swap(lhs.age, rhs.age);
}
};

```

В качестве альтернативы оператор копирования и перемещения может быть заменен одним оператором присваивания, который принимает экземпляр по значению вместо ссылки или ссылки на `rvalue`, чтобы облегчить использование идиомы копирования и свопинга.

```

Person& operator=(Person copy)
{
    swap(*this, copy);
    return *this;
}

```

Переход от правила трех к правилу пяти имеет важное значение для служебных целей, но в большинстве случаев это не является строго необходимым. Добавление конструктора копирования и оператора присваивания гарантирует, что перемещение этого типа не будет утечкой памяти (в этом случае перемещение-построение просто вернется к копированию), но будет выполнять копии, которые, по-видимому, не ожидали вызывающий.

Правило нуля

C ++ 11

Мы можем объединить принципы Правил пяти и [RAII](#), чтобы получить более компактный интерфейс: Правило нуля: любой ресурс, который нужно управлять, должен быть в своем собственном типе. Этот тип должен следовать правилу 5, но всем пользователям этого ресурса не нужно писать *ни одну* из пяти специальных функций-членов, и они могут просто по `default` все из них.

Используя класс `Person` представленный в [примере правила 3](#) , мы можем создать объект управления ресурсами для `cstrings` :

```
class cstring {
private:
    char* p;

public:
    ~cstring() { delete [] p; }
    cstring(cstring const& );
    cstring(cstring&& );
    cstring& operator=(cstring const& );
    cstring& operator=(cstring&& );

    /* other members as appropriate */
};
```

И как только это отдельно, наш класс `Person` становится намного проще:

```
class Person {
    cstring name;
    int arg;

public:
    ~Person() = default;
    Person(Person const& ) = default;
    Person(Person&& ) = default;
    Person& operator=(Person const& ) = default;
    Person& operator=(Person&& ) = default;

    /* other members as appropriate */
};
```

Особых членов в `Person` даже не нужно заявлять явно; компилятор по умолчанию или удалит их соответствующим образом на основе содержимого `Person` . Следовательно, следующее правило также является примером правила нуля.

```
struct Person {
    cstring name;
    int arg;
};
```

Если `cstring` должен быть типом только для перемещения, с оператором-конструктором / присваиванием `delete d` со ру, то `Person` автоматически будет также перемещаться.

Термин «правило нуля» был [введен Р. Мартиньо Фернандесом](#)

Правило трех

C ++ 03

Правило трех утверждает, что если тип когда-либо должен иметь пользовательский конструктор копирования, оператор присваивания копии или деструктор, тогда он должен

иметь *все три* .

Причиной этого правила является то, что для управления этим ресурсом необходим класс, который требует, чтобы любой из трех управлял некоторым ресурсом (файловые дескрипторы, динамически распределенная память и т. Д.), И все три. Функции копирования связаны с тем, как ресурс копируется между объектами, а деструктор уничтожает ресурс в соответствии с [принципами RAI](#) .

Рассмотрим тип, управляющий строковым ресурсом:

```
class Person
{
    char* name;
    int age;

public:
    Person(char const* new_name, int new_age)
        : name(new char[std::strlen(new_name) + 1])
        , age(new_age)
    {
        std::strcpy(name, new_name);
    }

    ~Person() {
        delete [] name;
    }
};
```

Поскольку `name` было назначено в конструкторе, деструктор освобождает его, чтобы избежать утечки памяти. Но что произойдет, если такой объект будет скопирован?

```
int main()
{
    Person p1("foo", 11);
    Person p2 = p1;
}
```

Сначала будет построено `p1` . Тогда `p2` будет скопирован из `p1` . Тем не менее, созданный C++ конструктор копии будет копировать каждый компонент типа *as-is*. Это означает, что `p1.name` и `p2.name` указывают на **одну и ту же** строку.

Когда `main` концы, будут вызваны деструкторы. `p2` первый деструктор `p2` ; он удалит строку. Тогда будет вызван деструктор `p1` . Однако строка *уже удалена* . Вызов `delete` в уже удаленной памяти дает неопределенное поведение.

Чтобы этого избежать, необходимо предоставить подходящий конструктор копирования. Один из подходов заключается в реализации системы подсчета ссылок, где разные экземпляры `Person` имеют одни и те же строковые данные. Каждый раз, когда выполняется копия, счетчик общих ссылок увеличивается. Деструктор затем уменьшает счетчик ссылок, освобождая только память, если счетчик равен нулю.

Или мы можем реализовать **семантику значений и глубокое копирование** :

```
Person(Person const& other)
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
{
    std::strcpy(name, other.name);
}

Person &operator=(Person const& other)
{
    // Use copy and swap idiom to implement assignment
    Person copy(other);
    swap(copy);          // assume swap() exchanges contents of *this and copy
    return *this;
}
```

Реализация оператора присваивания копии осложняется необходимостью выпуска существующего буфера. Метод `copy` and `swap` создает временный объект, который содержит новый буфер. Перестановка содержимого `*this` и `copy` дает право собственности на `copy` исходного буфера. Уничтожение `copy` , когда функция возвращает, освобождает буфер, ранее принадлежавший `*this` .

Защита от самостоятельного назначения

При написании оператора присваивания копии *очень* важно, чтобы он мог работать в случае самонаведения. То есть, это должно позволить:

```
SomeType t = ...;
t = t;
```

Самоназвание обычно не происходит столь очевидным образом. Обычно это происходит через обходной маршрут через различные системы кода, где местоположение назначения просто имеет два указателя на `Person` или ссылки и не имеет представления, что они являются одним и тем же объектом.

Любой оператор копирования, который вы пишете, должен учесть это.

Типичный способ сделать это - обернуть всю логику назначения в таком состоянии, как это:

```
SomeType &operator=(const SomeType &other)
{
    if(this != &other)
    {
        //Do assignment logic.
    }
    return *this;
}
```

Примечание. Важно подумать о самоопределении и убедиться, что ваш код ведет себя

правильно, когда это происходит. Однако самоназвание является очень редким явлением, и оптимизация для предотвращения его может фактически pessимизировать нормальный случай. Так как нормальный случай гораздо более распространен, pessимизация для самостоятельного назначения может значительно снизить эффективность кода (поэтому будьте осторожны с ним).

Например, обычным методом реализации оператора присваивания является `copy and swap idiom`. Обычная реализация этой методики не утруждает себя проверкой на самоназвание (хотя самозадача является дорогостоящей, потому что делается копия). Причина в том, что pessимизация нормального случая оказалась намного более дорогостоящей (как это происходит чаще).

C ++ 11

Операторы переадресации также должны быть защищены от самонаправления. Однако логика для многих таких операторов основана на `std::swap`, которая может обрабатывать `std::swap` из / в одну и ту же память. Поэтому, если ваша логика назначения перемещения - это не что иное, как серия операций свопинга, тогда вам не нужна защита самоназвания.

Если это не так, вы *должны* принять аналогичные меры, как указано выше.

Прочитайте Правило три, пять и ноль онлайн: <https://riptutorial.com/ru/cplusplus/topic/1206/правило-три--пять-и-ноль>

глава 88: препроцессор

Вступление

Препроцессор C представляет собой простой текстовый парсер / заменитель, который запускается до фактической компиляции кода. Используется для расширения и облегчения использования языка C (и более поздней версии C ++), его можно использовать для:

- а. **Включение других файлов** с помощью `#include`
- б. **Определите макрос замены текста** с помощью `#define`
- с. **Условная компиляция** с использованием `#if #ifdef`
- д. **Спецификация платформы / компилятора** (как расширение условной компиляции)

замечания

Операторы препроцессора выполняются до того, как исходные файлы передаются компилятору. Они способны к очень низкой условной логике. Поскольку конструкции препроцессора (например, макросы, подобные объектам) не типизируются, как обычные функции (шаг предварительной обработки происходит до компиляции), компилятор не может применять проверки типов, поэтому их следует тщательно использовать.

Examples

Включить охранников

Заголовочный файл может быть включен в другие файлы заголовков. Исходный файл (блок компиляции), который включает несколько заголовков, может поэтому косвенно включать некоторые заголовки более одного раза. Если такой заголовочный файл, который включен более одного раза, содержит определения, компилятор (после предварительной обработки) обнаруживает нарушение правила Единого определения (например, §3.2 стандарта 2003 C ++) и поэтому выдает сбой диагностики и компиляции.

Множественное включение предотвращается с помощью «include guard», которые иногда также называются защитами заголовков или защитой от макросов. Они реализованы с использованием препроцессора `#define` , `#ifndef` , директив `#endif` .

например

```
// Foo.h
```

```
#ifndef FOO_H_INCLUDED
#define FOO_H_INCLUDED

class Foo    // a class definition
{
};

#endif
```

Ключевым преимуществом использования охранников является то, что они будут работать со всеми стандартными компиляторами и препроцессорами.

Однако включение охранников также создает проблемы для разработчиков, так как необходимо гарантировать, что макросы уникальны во всех заголовках, используемых в проекте. В частности, если два (или более) заголовка используют `FOO_H_INCLUDED` качестве `FOO_H_INCLUDED include`, первый из этих заголовков, включенных в блок компиляции, эффективно предотвратит включение других. Особые вызовы возникают, если в проекте используется ряд сторонних библиотек с файлами заголовков, которые используются для использования включенных охранников.

Также необходимо убедиться, что макросы, используемые в включении охранников, не конфликтуют с другими макросами, определенными в файлах заголовков.

Большинство реализаций на C++ также поддерживают директиву `#pragma once` которая гарантирует, что файл включен только один раз в рамках одной компиляции. Это стандартная директива *de facto*, но она не является частью стандарта ISO C++. Например:

```
// Foo.h
#pragma once

class Foo
{
};
```

В `#pragma once` время как `#pragma once` избегает некоторых проблем, связанных с включением охранников, `#pragma` - по определению в стандартах - по сути является компилятором, и он будет игнорироваться компиляторами, которые его не поддерживают. Проекты, которые используют `#pragma once`, сложнее переносить на компиляторы, которые его не поддерживают.

Ряд руководств по кодированию и стандартов достоверности для C++ специально препятствует использованию препроцессора, отличному от `#include` заголовочных файлов, или для целей размещения включают в себя защитные устройства в заголовках.

Условная логика и кросс-платформенная обработка

В двух словах условная логика предварительной обработки заключается в том, чтобы

сделать кодовую логику доступной или недоступной для компиляции с использованием макроопределений.

Три видных варианта использования:

- различные **профили приложений** (например, отладка, выпуск, тестирование, оптимизация), которые могут быть кандидатами одного и того же приложения (например, с дополнительным протоколированием).
- **кросс-платформенные компиляции** - единая кодовая база, множественные платформы компиляции.
- используя общую кодовую базу для нескольких **версий приложений** (например, базовые **версии** , Premium и Pro версии программного обеспечения) - с немного отличающимися функциями.

Пример а: кросс-платформенный подход для удаления файлов (иллюстративный):

```
#ifdef _WIN32
#include <windows.h> // and other windows system files
#endif
#include <cstdio>

bool remove_file(const std::string &path)
{
#ifdef _WIN32
    return DeleteFile(path.c_str());
#elif defined(_POSIX_VERSION) || defined(__unix__)
    return (0 == remove(path.c_str()));
#elif defined(__APPLE__)
    //TODO: check if NSAPI has a more specific function with permission dialog
    return (0 == remove(path.c_str()));
#else
#error "This platform is not supported"
#endif
}
```

Макросы, такие как `_WIN32` , `__APPLE__` или `__unix__` , обычно predefinedены соответствующими реализациями.

Пример b: Включение дополнительного ведения журнала для сборки отладки:

```
void s_PrintAppStateOnUserPrompt()
{
    std::cout << "-----BEGIN-DUMP-----\n"
               << AppState::Instance()->Settings().ToString() << "\n"
    #if ( 1 == TESTING_MODE ) //privacy: we want user details only when testing
        << ListToString(AppState::UndoStack()->GetActionNames())
        << AppState::Instance()->CrntDocument().Name()
        << AppState::Instance()->CrntDocument().SignatureSHA() << "\n"
    #endif
        << "-----END-DUMP-----\n"
}
```

Пример с: Включите премиум-функцию в отдельную сборку продукта (обратите внимание:

это иллюстративно. Часто лучше дать возможность разблокировать функцию без необходимости переустановки приложения)

```
void MainWindow::OnProcessButtonClick()
{
#ifdef _PREMIUM
    CreatePurchaseDialog("Buy App Premium", "This feature is available for our App Premium
users. Click the Buy button to purchase the Premium version at our website");
    return;
#endif
    //...actual feature logic here
}
```

Некоторые общие трюки:

Определение символов во время вызова:

Препроцессор можно вызывать с предопределенными символами (с необязательной инициализацией). Например, эта команда (`gcc -E` запускает только препроцессор)

```
gcc -E -DOPTIMISE_FOR_OS_X -DTESTING_MODE=1 Sample.cpp
```

процессов `Sample.cpp` так же, как если бы `#define OPTIMISE_FOR_OS_X` и `#define TESTING_MODE 1` были добавлены в начало `Sample.cpp`.

Определение макроса:

Если макрос не определен и его значение сравнивается или проверяется, препроцессор почти всегда молча принимает значение 0. Есть несколько способов работать с этим. Один из подходов состоит в том, чтобы предположить, что параметры по умолчанию представлены как 0, и любые изменения (например, профиль профиля приложения) должны быть явно выполнены (например, `ENABLE_EXTRA_DEBUGGING = 0` по умолчанию, установите `-DENABLE_EXTRA_DEBUGGING = 1` для переопределения). Другой подход - сделать все определения и значения по умолчанию явными. Этого можно достичь с помощью комбинации директив `#ifndef` и `#error`:

```
#ifndef (ENABLE_EXTRA_DEBUGGING)
// please include DefaultDefines.h if not already included.
#    error "ENABLE_EXTRA_DEBUGGING is not defined"
#else
#    if ( 1 == ENABLE_EXTRA_DEBUGGING )
        //code
#    endif
#endif
```

макрос

Макросы подразделяются на две основные группы: макросы, подобные объектам, и макросы, подобные функциям. Макросы рассматриваются как замещение токенов в начале

процесса компиляции. Это означает, что большие (или повторяющиеся) разделы кода могут быть абстрагированы в макрос препроцессора.

```
// This is an object-like macro
#define PI 3.14159265358979

// This is a function-like macro.
// Note that we can use previously defined macros
// in other macro definitions (object-like or function-like)
// But watch out, its quite useful if you know what you're doing, but the
// Compiler doesnt know which type to handle, so using inline functions instead
// is quite recommended (But e.g. for Minimum/Maximum functions it is quite useful)
#define AREA(r) (PI*(r)*(r))

// They can be used like this:
double pi_macro = PI;
double area_macro = AREA(4.6);
```

Библиотека Qt использует эту технику для создания системы метаобъектов путем объявления пользователем макроса `Q_OBJECT` во главе пользовательского класса, расширяющего `QObject`.

Имена макросов обычно записываются во всех шапках, чтобы упростить их отличать от обычного кода. Это не является обязательным требованием, но многие программисты считаются хорошим стилем.

Когда встречается объект-подобный макрос, он расширяется как простая операция копирования-вставки, при этом имя макроса заменяется его определением. Когда встречается функционально-подобный макрос, расширяются его имя и его параметры.

```
double pi_squared = PI * PI;
// Compiler sees:
double pi_squared = 3.14159265358979 * 3.14159265358979;

double area = AREA(5);
// Compiler sees:
double area = (3.14159265358979*(5)*(5))
```

В связи с этим функциональные макропараметры часто заключаются в круглые скобки, как в `AREA()` выше. Это предотвращает любые ошибки, которые могут возникать при расширении макросов, в частности ошибки, вызванные одним макропараметром, состоящим из нескольких фактических значений.

```
#define BAD_AREA(r) PI * r * r

double bad_area = BAD_AREA(5 + 1.6);
// Compiler sees:
double bad_area = 3.14159265358979 * 5 + 1.6 * 5 + 1.6;

double good_area = AREA(5 + 1.6);
// Compiler sees:
```

```
double good_area = (3.14159265358979*(5 + 1.6)*(5 + 1.6));
```

Также обратите внимание, что из-за этого простого расширения необходимо соблюдать параметры, переданные макросам, чтобы предотвратить неожиданные побочные эффекты. Если параметр изменен во время оценки, он будет изменяться каждый раз, когда он используется в расширенном макросе, который обычно не является тем, что мы хотим. Это справедливо даже в том случае, если макрос заключает в круглые скобки параметры, чтобы предотвратить разрыв в чем-либо.

```
int oops = 5;
double incremental_damage = AREA(oops++);
// Compiler sees:
double incremental_damage = (3.14159265358979*(oops++)*(oops++));
```

Кроме того, макросы не обеспечивают безопасности типов, что приводит к трудно понятным ошибкам в отношении несоответствия типов.

Поскольку программисты обычно заканчивают строки точкой с запятой, макросы, предназначенные для использования в качестве автономных линий, часто предназначены для «проглатывания» точки с запятой; это предотвращает появление непредвиденных ошибок из-за дополнительной точки с запятой.

```
#define IF_BREAKER(Func) Func();

if (some_condition)
    // Oops.
    IF_BREAKER(some_func);
else
    std::cout << "I am accidentally an orphan." << std::endl;
```

В этом примере непреднамеренная двойная точка с запятой разбивает блок `if...else`, не позволяя компилятору сопоставить `else` с `if`. Чтобы предотвратить это, точка с запятой не указана в определении макроса, что приведет к «проглатыванию» точки с запятой сразу после ее использования.

```
#define IF_FIXER(Func) Func()

if (some_condition)
    IF_FIXER(some_func);
else
    std::cout << "Hooray! I work again!" << std::endl;
```

Оставляя конечную точку с запятой, также можно использовать макрос без окончания текущего утверждения, что может быть полезным.

```
#define DO_SOMETHING(Func, Param) Func(Param, 2)

// ...
```

```
some_function(DO_SOMETHING(some_func, 3), DO_SOMETHING(some_func, 42));
```

Обычно определение макроса заканчивается в конце строки. Однако, если макрос должен покрывать несколько строк, в конце строки может использоваться обратная косая черта, чтобы указать это. Эта обратная косая черта должна быть последним символом в строке, которая указывает препроцессору, что следующая строка должна быть конкатенирована в текущей строке, рассматривая их как одну строку. Это можно использовать несколько раз подряд.

```
#define TEXT "I \
am \
many \
lines."

// ...

std::cout << TEXT << std::endl; // Output:  I am many lines.
```

Это особенно полезно для сложных макросов, подобных функциям, которые, возможно, должны охватывать несколько строк.

```
#define CREATE_OUTPUT_AND_DELETE(Str) \
    std::string* tmp = new std::string(Str); \
    std::cout << *tmp << std::endl; \
    delete tmp;

// ...

CREATE_OUTPUT_AND_DELETE("There's no real need for this to use 'new'.")
```

В случае более сложных макросов, подобных функциям, может оказаться полезным предоставить им собственную область действия, чтобы предотвратить возможные столкновения имен или вызвать уничтожение объектов в конце макроса, аналогично фактической функции. Обычная идиома для этого - *делать в то время как 0*, где макрос заключен в блок *do-while*. Этот блок обычно не следует с точкой с запятой, позволяя ему проглотить точку с запятой.

```
#define DO_STUFF(Type, Param, ReturnVar) do { \
    Type temp(some_setup_values); \
    ReturnVar = temp.process(Param); \
} while (0)

int x;
DO_STUFF(MyClass, 41153.7, x);

// Compiler sees:

int x;
do {
```

```
MyClass temp(some_setup_values);
x = temp.process(41153.7);
} while (0);
```

Существуют также переменные макросы; аналогично вариационным функциям, они принимают переменное количество аргументов, а затем расширяют их все вместо специального параметра «Varargs», `__VA_ARGS__` .

```
#define VARIADIC(Param, ...) Param(__VA_ARGS__)

VARIADIC(sprintf, "%d", 8);
// Compiler sees:
sprintf("%d", 8);
```

Обратите внимание, что во время расширения `__VA_ARGS__` может быть помещен в любом месте определения и будет правильно `__VA_ARGS__` .

```
#define VARIADIC2(POne, PTwo, PThree, ...) POne(PThree, __VA_ARGS__, PTwo)

VARIADIC2(some_func, 3, 8, 6, 9);
// Compiler sees:
some_func(8, 6, 9, 3);
```

В случае вариационного параметра с нулевым аргументом разные компиляторы будут обрабатывать конечную запятую по-разному. Некоторые компиляторы, такие как Visual Studio, беззвучно проглатывают запятую без специального синтаксиса. Другие компиляторы, такие как GCC, требуют, чтобы вы разместили `##` непосредственно перед `__VA_ARGS__` . Из-за этого разумно условно определять переменные макросы, когда переносимость вызывает беспокойство.

```
// In this example, COMPILER is a user-defined macro specifying the compiler being used.

#if COMPILER == "VS"
#define VARIADIC3(Name, Param, ...) Name(Param, __VA_ARGS__)
#elif COMPILER == "GCC"
#define VARIADIC3(Name, Param, ...) Name(Param, ##__VA_ARGS__)
#endif /* COMPILER */
```

Сообщения об ошибках препроцессора

Ошибки компиляции могут быть сгенерированы с использованием препроцессора. Это полезно по ряду причин, некоторые из которых включают в себя уведомление пользователей, если они находятся на неподдерживаемой платформе или неподдерживаемый компилятор.

например, Return Error, если версия gcc 3.0.0 или более ранняя.

```
#if __GNUC__ < 3
```

```
#error "This code requires gcc > 3.0.0"
#endif
```

например, «Ошибка возврата» при компиляции на компьютере Apple.

```
#ifdef __APPLE__
#error "Apple products are not supported in this release"
#endif
```

Предопределенные макросы

Предопределенные макросы - это те, которые определяет компилятор (в отличие от тех, которые пользователь определяет в исходном файле). Эти макросы не должны переопределяться или неопределяться пользователем.

Следующие макросы предопределены стандартом C ++:

- `__LINE__` содержит номер строки строки, на которой используется этот макрос, и может быть изменен директивой `#line`.
- `__FILE__` содержит имя файла, в котором этот макрос используется, и может быть изменен директивой `#line`.
- `__DATE__` содержит дату (в формате "Mmm dd yyyy") компиляции файла, где *Mmm* отформатирован так, как если бы был получен вызов `std::asctime()`.
- `__TIME__` содержит время (в формате "hh:mm:ss") компиляции файла.
- `__cplusplus` определяется (совместимыми) компиляторами C ++ при компиляции файлов C ++. Его значение - стандартная версия, которая **полностью** соответствует компилятору, то есть `199711L` для C ++ 98 и C ++ 03, `201103L` для C ++ 11 и `201402L` для C ++ 14.

C ++ 11

- `__STDC_HOSTED__` определяется как `1` если реализация *размещена*, или `0` если она является *автономной*.

C ++ 17

- `__STDCPP_DEFAULT_NEW_ALIGNMENT__` содержит литерал `size_t`, который является выравниванием, используемым для вызова `operator new` выравнивания-незнания `operator new`.

Кроме того, следующие макросы могут быть предопределены реализациями и могут присутствовать или отсутствовать:

- `__STDC__` имеет значение, зависящее от реализации, и обычно определяется только при компиляции файла как `C`, чтобы обозначить полное соответствие стандарту C. (Или никогда, если компилятор решает не поддерживать этот макрос.)

C ++ 11

- `__STDC_VERSION__` имеет значение, зависящее от реализации, и его значение обычно является версией C, аналогично тому, как `__cplusplus` является `__cplusplus` на C++. (Или даже не определено, если компилятор решает не поддерживать этот макрос.)
- `__STDC_MB_MIGHT_NEQ_WC__` определяется как 1, если значения узкой кодировки базового набора символов могут быть не равны значениям их широких копий (например, если `(uintmax_t)'x' != (uintmax_t)L'x'`)
- `__STDC_ISO_10646__` определяется, если `wchar_t` закодирован как Unicode и расширяется до целочисленной константы в форме `yyyyymmL`, указывая, что поддерживается последняя версия Unicode.
- `__STDCPP_STRICT_POINTER_SAFETY__` определяется как 1, если реализация имеет *строгую безопасность указателя* (в противном случае она *ослабляет безопасность указателя*)
- `__STDCPP_THREADS__` определяется как 1, если программа может иметь более одного потока выполнения (применяемая к автономной реализации - реализация хоста всегда может иметь более одного потока)

Также стоит упомянуть `__func__`, который не является макросом, а предопределенной функцией-локальной переменной. Он содержит имя функции, в которой он используется, в виде массива статических символов в формате, определяемом реализацией.

Помимо этих стандартных предопределенных макросов, компиляторы могут иметь собственный набор предопределенных макросов. Для их изучения нужно обратиться к документации компилятора. Например:

- [НКУ](#)
- [Microsoft Visual C++](#)
- [Лязг](#)
- [Компилятор Intel C++](#)

Некоторые макросы предназначены только для поддержки некоторой функции:

```
#ifdef __cplusplus // if compiled by C++ compiler
extern "C"{ // C code has to be decorated
    // C library header declarations here
}
#endif
```

Другие очень полезны для отладки:

C++ 11

```
bool success = doSomething( /*some arguments*/ );
if( !success ){
    std::cerr << "ERROR: doSomething() failed on line " << __LINE__ - 2
               << " in function " << __func__ << "()"
               << " in file " << __FILE__
               << std::endl;
}
```


И другие для тривиального контроля версий:

```
int main( int argc, char *argv[] ){
    if( argc == 2 && std::string( argv[1] ) == "-v" ){
        std::cout << "Hello World program\n"
                    << "v 1.1\n" // I have to remember to update this manually
                    << "compiled: " << __DATE__ << ' ' << __TIME__ // this updates automagically
                    << std::endl;
    }
    else{
        std::cout << "Hello World!\n";
    }
}
```

X-макросы

Идиоматический метод генерации повторяющихся структур кода во время компиляции.

X-масро состоит из двух частей: списка и выполнения списка.

Пример:

```
#define LIST \
    X(dog)    \
    X(cat)    \
    X(racoon)

// class Animal {
//     public:
//         void say();
// };

#define X(name) Animal name;
LIST
#undef X

int main() {
#define X(name) name.say();
    LIST
#undef X

    return 0;
}
```

который расширяется препроцессором до следующего:

```
Animal dog;
Animal cat;
Animal racoon;

int main() {
    dog.say();
    cat.say();
    racoon.say();

    return 0;
}
```

```
}
```

Поскольку списки становятся больше (скажем, более 100 элементов), этот метод помогает избежать чрезмерного копирования.

Источник: https://en.wikipedia.org/wiki/X_Macro

См. Также: [X-макросы](#)

Если определение скелетно неуместного `x` перед использованием `LIST` не по своему усмотрению, вы можете передать имя макроса в качестве аргумента:

```
#define LIST(MACRO) \
    MACRO(dog) \
    MACRO(cat) \
    MACRO(racoon)
```

Теперь вы явно указываете, какой макрос следует использовать при расширении списка, например

```
#define FORWARD_DECLARE_ANIMAL(name) Animal name;
LIST(FORWARD_DECLARE_ANIMAL)
```

Если каждый вызов `MACRO` должен принимать дополнительные параметры - постоянный по отношению к списку, могут использоваться переменные макросы

```
//a walkaround for Visual studio
#define EXPAND(x) x

#define LIST(MACRO, ...) \
    EXPAND(MACRO(dog, __VA_ARGS__)) \
    EXPAND(MACRO(cat, __VA_ARGS__)) \
    EXPAND(MACRO(racoon, __VA_ARGS__))
```

Первый аргумент предоставляется `LIST`, а остальное предоставляется пользователем в вызове `LIST`. Например:

```
#define FORWARD_DECLARE(name, type, prefix) type prefix##name;
LIST(FORWARD_DECLARE, Animal, anim_)
LIST(FORWARD_DECLARE, Object, obj_)
```

будет расширяться до

```
Animal anim_dog;
Animal anim_cat;
Animal anim_racoon;
Object obj_dog;
Object obj_cat;
Object obj_racoon;
```

pragma один раз

Большинство, но не все, реализации C ++ поддерживают директиву `#pragma once` которая гарантирует, что файл включен только один раз в рамках одной компиляции. Он не является частью стандарта ISO C ++. Например:

```
// Foo.h
#pragma once

class Foo
{
};
```

В `#pragma once` время как `#pragma once` избегает некоторых проблем, связанных с [включением охранников](#), `#pragma` - по определению в стандартах - по сути является компилятором, и он будет игнорироваться компиляторами, которые его не поддерживают. Проекты, которые используют `#pragma once` должны быть изменены, чтобы быть стандартными.

С некоторыми компиляторами, особенно с теми, которые используют [предварительно скомпилированные заголовки](#), `#pragma once` может привести к значительному ускорению процесса компиляции. Аналогичным образом, некоторые препроцессоры добиваются ускорения компиляции, отслеживая используемые заголовки, включая защитные устройства. Чистая выгода, когда используются как `#pragma once` и включенные охранники, зависит от реализации и может быть либо увеличением, либо уменьшением времени компиляции.

`#pragma once` объединенный с [включенными охранниками](#), был рекомендуемым макетом для файлов заголовков при написании приложений на основе MFC в Windows и был сгенерирован `add class add dialog Visual Studio, add dialog, add windows` мастеров. Поэтому очень часто их объединяют в C ++ Windows Applicants.

Операторы препроцессора

`#` оператор или оператор стробирования используется для преобразования параметра макроса в строковый литерал. Его можно использовать только с макросами с аргументами.

```
// preprocessor will convert the parameter x to the string literal x
#define PRINT(x) printf(#x "\n")

PRINT(This line will be converted to string by preprocessor);
// Compiler sees
printf("This line will be converted to string by preprocessor"\n");
```

Компилятор объединяет две строки, а окончательный аргумент `printf()` будет строковым литералом с символом новой строки в конце.

Препроцессор будет игнорировать пробелы до или после аргумента макроса. Таким

образом, описание печати будет иметь тот же результат.

```
PRINT(    This line will be converted to string by preprocessor );
```

Если для параметра строкового литерала требуется escape-последовательность, как перед двойной кавычкой (), она будет автоматически вставлена препроцессором.

```
PRINT(This "line" will be converted to "string" by preprocessor);  
// Compiler sees  
printf("This \"line\" will be converted to \"string\" by preprocessor\"\"\n");
```

Оператор ## или оператор сопоставления токенов используется для конкатенации двух параметров или жетонов макроса.

```
// preprocessor will combine the variable and the x  
#define PRINT(x) printf("variable" #x " = %d", variable##x)  
  
int variableY = 15;  
PRINT(Y);  
//compiler sees  
printf("variable""Y" = %d", variableY);
```

и конечный результат будет

```
variableY = 15
```

Прочитайте препроцессор онлайн: <https://riptutorial.com/ru/cplusplus/topic/1098/препроцессор>

глава 89: Примеры клиентского сервера

Examples

Привет, TCP-сервер

Позвольте мне начать с того, что вы должны сначала посетить [Руководство Веей по программированию в сети](#) и дать ему быстрое чтение, что объясняет большую часть этого материала немного более подробно. Здесь мы создадим простой TCP-сервер, который скажет «Hello World» всем входящим соединениям, а затем закроет их. Другое дело, что сервер будет обмениваться данными с клиентами итеративно, что означает один клиент за раз. Обязательно проверьте соответствующие справочные страницы, так как они могут содержать ценную информацию о каждом вызове функций и структурах сокетов.

Мы запустим сервер с портом, поэтому мы также примем аргумент для номера порта. Давайте начнем с кода -

```
#include <cstring>      // sizeof()
#include <iostream>
#include <string>

// headers for socket(), getaddrinfo() and friends
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>

#include <unistd.h>      // close()

int main(int argc, char *argv[])
{
    // Let's check if port number is supplied or not..
    if (argc != 2) {
        std::cerr << "Run program as 'program <port>'\n";
        return -1;
    }

    auto &portNum = argv[1];
    const unsigned int backlog = 8; // number of connections allowed on the incoming queue

    addrinfo hints, *res, *p;      // we need 2 pointers, res to hold and p to iterate over
    memset(&hints, 0, sizeof(hints));

    // for more explanation, man socket
    hints.ai_family   = AF_UNSPEC;    // don't specify which IP version to use yet
    hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM refers to TCP, SOCK_DGRAM will be?
    hints.ai_flags    = AI_PASSIVE;

    // man getaddrinfo
```

```

int gAddRes = getaddrinfo(NULL, portNum, &hints, &res);
if (gAddRes != 0) {
    std::cerr << gai_strerror(gAddRes) << "\n";
    return -2;
}

std::cout << "Detecting addresses" << std::endl;

unsigned int numOfAddr = 0;
char ipStr[INET6_ADDRSTRLEN];    // ipv6 length makes sure both ipv4/6 addresses can be
stored in this variable

// Now since getaddrinfo() has given us a list of addresses
// we're going to iterate over them and ask user to choose one
// address for program to bind to
for (p = res; p != NULL; p = p->ai_next) {
    void *addr;
    std::string ipVer;

    // if address is ipv4 address
    if (p->ai_family == AF_INET) {
        ipVer          = "IPv4";
        sockaddr_in *ipv4 = reinterpret_cast<sockaddr_in *>(p->ai_addr);
        addr           = &(ipv4->sin_addr);
        ++numOfAddr;
    }

    // if address is ipv6 address
    else {
        ipVer          = "IPv6";
        sockaddr_in6 *ipv6 = reinterpret_cast<sockaddr_in6 *>(p->ai_addr);
        addr           = &(ipv6->sin6_addr);
        ++numOfAddr;
    }

    // convert IPv4 and IPv6 addresses from binary to text form
    inet_ntop(p->ai_family, addr, ipStr, sizeof(ipStr));
    std::cout << "(" << numOfAddr << ") " << ipVer << " : " << ipStr
        << std::endl;
}

// if no addresses found :(
if (!numOfAddr) {
    std::cerr << "Found no host address to use\n";
    return -3;
}

// ask user to choose an address
std::cout << "Enter the number of host address to bind with: ";
unsigned int choice = 0;
bool madeChoice    = false;
do {
    std::cin >> choice;
    if (choice > (numOfAddr + 1) || choice < 1) {
        madeChoice = false;
        std::cout << "Wrong choice, try again!" << std::endl;
    } else
        madeChoice = true;
} while (!madeChoice);

```

```

p = res;

// let's create a new socket, sockfd is returned as descriptor
// man socket for more information
// these calls usually return -1 as result of some error
int sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
if (sockfd == -1) {
    std::cerr << "Error while creating socket\n";
    freeaddrinfo(res);
    return -4;
}

// Let's bind address to our socket we've just created
int bindR = bind(sockfd, p->ai_addr, p->ai_addrlen);
if (bindR == -1) {
    std::cerr << "Error while binding socket\n";

    // if some error occurs, make sure to close socket and free resources
    close(sockfd);
    freeaddrinfo(res);
    return -5;
}

// finally start listening for connections on our socket
int listenR = listen(sockfd, backlog);
if (listenR == -1) {
    std::cerr << "Error while Listening on socket\n";

    // if some error occurs, make sure to close socket and free resources
    close(sockfd);
    freeaddrinfo(res);
    return -6;
}

// structure large enough to hold client's address
sockaddr_storage client_addr;
socklen_t client_addr_size = sizeof(client_addr);

const std::string response = "Hello World";

// a fresh infinite loop to communicate with incoming connections
// this will take client connections one at a time
// in further examples, we're going to use fork() call for each client connection
while (1) {

    // accept call will give us a new socket descriptor
    int newFD
        = accept(sockfd, (sockaddr *) &client_addr, &client_addr_size);
    if (newFD == -1) {
        std::cerr << "Error while Accepting on socket\n";
        continue;
    }

    // send call sends the data you specify as second param and it's length as 3rd param,

```

```

also returns how many bytes were actually sent
    auto bytes_sent = send(newFD, response.data(), response.length(), 0);
    close(newFD);
}

close(sockFD);
freeaddrinfo(res);

return 0;
}

```

Следующая программа работает как -

```

Detecting addresses
(1) IPv4 : 0.0.0.0
(2) IPv6 : ::
Enter the number of host address to bind with: 1

```

Привет, TCP-клиент

Эта программа бесплатна для программы Hello TCP Server, вы можете запустить любой из них, чтобы проверить правильность друг друга. Поток программы довольно распространен с сервером Hello TCP, поэтому не забудьте взглянуть на это тоже.

Вот код -

```

#include <cstring>
#include <iostream>
#include <string>

#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    // Now we're taking an ipaddress and a port number as arguments to our program
    if (argc != 3) {
        std::cerr << "Run program as 'program <ipaddress> <port>'\n";
        return -1;
    }

    auto &ipAddress = argv[1];
    auto &portNum    = argv[2];

    addrinfo hints, *p;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags    = AI_PASSIVE;

    int gAddRes = getaddrinfo(ipAddress, portNum, &hints, &p);

```



```

if (gAddRes != 0) {
    std::cerr << gai_strerror(gAddRes) << "\n";
    return -2;
}

if (p == NULL) {
    std::cerr << "No addresses found\n";
    return -3;
}

// socket() call creates a new socket and returns it's descriptor
int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
if (sockFD == -1) {
    std::cerr << "Error while creating socket\n";
    return -4;
}

// Note: there is no bind() call as there was in Hello TCP Server
// why? well you could call it though it's not necessary
// because client doesn't necessarily has to have a fixed port number
// so next call will bind it to a random available port number

// connect() call tries to establish a TCP connection to the specified server
int connectR = connect(sockFD, p->ai_addr, p->ai_addrlen);
if (connectR == -1) {
    close(sockFD);
    std::cerr << "Error while connecting socket\n";
    return -5;
}

std::string reply(15, ' ');

// recv() call tries to get the response from server
// BUT there's a catch here, the response might take multiple calls
// to recv() before it is completely received
// will be demonstrated in another example to keep this minimal
auto bytes_rcv = recv(sockFD, &reply.front(), reply.size(), 0);
if (bytes_rcv == -1) {
    std::cerr << "Error while receiving bytes\n";
    return -6;
}

std::cout << "\nClient recieved: " << reply << std::endl;
close(sockFD);
freeaddrinfo(p);

return 0;
}

```

Прочитайте Примеры клиентского сервера онлайн:

<https://riptutorial.com/ru/cplusplus/topic/7177/примеры-клиентского-сервера>

глава 90: приоритет оператора

замечания

Операторы перечислены сверху вниз, в нисходящем приоритете. Операторы с одинаковым числом имеют одинаковый приоритет и одну и ту же ассоциативность.

1. `::`
2. Постфиксные операторы: `[] () T(...) . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid`
3. Унарные префиксные операторы: `++ -- * & + - ! ~ sizeof new delete delete[]`; нотация C-стиля, `(T)...`; (C ++ 11 и выше) `sizeof... alignof noexcept`
4. `.` * и `->*`
5. `*`, `/` и `%`, бинарные арифметические операторы
6. `+` и `-`, двоичные арифметические операторы
7. `<<` и `>>`
8. `<`, `>`, `<=`, `>=`
9. `==` и `!=`
10. `&`, побитовый оператор И
11. `^`
12. `|`
13. `&&`
14. `||`
15. `?:` (тернарный условный оператор)
16. `=`, `*`, `/=`, `%=`, `+=`, `-=`, `>>=`, `<<=`, `&=`, `^=`, `|=`
17. `throw`
18. `,` (оператор запятой)

Назначение, составное назначение и тернарные условные операторы являются право-ассоциативными. Все остальные бинарные операторы лево-ассоциативны.

Правила для тернарного условного оператора немного сложнее, чем могут выразить простые правила приоритета.

- Операнд связывается менее жестко с `?` слева или справа : справа от любого другого оператора. Фактически, второй операнд условного оператора анализируется так, как будто он заключен в скобки. Это позволяет выразить такое выражение, как `a ? b , c : d` чтобы быть синтаксически действительными.
- Операнд более тесно связан с `?` справа от оператора назначения или `throw` слева, так что `a = b ? c : d` эквивалентно `a = (b ? c : d)` и `throw a ? b : c` эквивалентно `throw (a ? b : c)`.
- Операнд более тесно связан с оператором присваивания справа от него : слева от него, `a ? b : c = d` эквивалентно `a ? b : (c = d)`.

Examples

Арифметические операторы

Арифметические операторы в C ++ имеют тот же приоритет, что и в математике:

Умножение и деление оставили ассоциативность (что означает, что они будут оцениваться слева направо), и они имеют более высокий приоритет, чем сложение и вычитание, которые также оставят ассоциативность.

Мы также можем форсировать приоритет выражения с помощью круглых скобок (). Точно так же, как и в обычной математике.

```
// volume of a spherical shell = 4 pi R^3 - 4 pi r^3
double vol = 4.0*pi*R*R*R/3.0 - 4.0*pi*r*r*r/3.0;

//Addition:

int a = 2+4/2;           // equal to: 2+(4/2)           result: 4
int b = (3+3)/2;         // equal to: (3+3)/2           result: 3

//With Multiplication

int c = 3+4/2*6;          // equal to: 3+((4/2)*6)       result: 15
int d = 3*(3+6)/9;        // equal to: (3*(3+6))/9       result: 3

//Division and Modulo

int g = 3-3%1;            // equal to: 3 % 1 = 0   3 - 0 = 3
int h = 3-(3%1);          // equal to: 3 % 1 = 0   3 - 0 = 3
int i = 3-3/1%3;          // equal to: 3 / 1 = 3   3 % 3 = 0   3 - 0 = 3
int l = 3-(3/1)%3;        // equal to: 3 / 1 = 3   3 % 3 = 0   3 - 0 = 3
int m = 3-(3/(1%3));       // equal to: 1 % 3 = 1   3 / 1 = 3   3 - 3 = 0
```

Логические операторы AND и OR

Эти операторы имеют обычный приоритет в C ++: И до OR.

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || has_foreign_license && num_days <= 60;
```

Этот код эквивалентен следующему:

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || (has_foreign_license && num_days <= 60);
```

Добавление скобок не меняет поведения, однако облегчает ее чтение. Добавляя эти круглые скобки, не существует путаницы в намерениях писателя.

Логические && и || операторы: короткое замыкание

&& имеет приоритет над ||, это означает, что круглые скобки помещаются для оценки того, что будет оцениваться вместе.

с ++ использует оценку короткого замыкания в && и || чтобы не делать ненужных казней. Если левая часть || возвращает true, правая сторона больше не нуждается в оценке.

```
#include <iostream>
#include <string>

using namespace std;

bool True(string id){
    cout << "True" << id << endl;
    return true;
}

bool False(string id){
    cout << "False" << id << endl;
    return false;
}

int main(){
    bool result;
    //let's evaluate 3 booleans with || and && to illustrate operator precedence
    //precedence does not mean that && will be evaluated first but rather where
    //parentheses would be added
    //example 1
    result =
        False("A") || False("B") && False("C");
        // eq. False("A") || (False("B") && False("C"))
    //FalseA
    //FalseB
    //"Short-circuit evaluation skip of C"
    //A is false so we have to evaluate the right of ||,
    //B being false we do not have to evaluate C to know that the result is false

    result =
        True("A") || False("B") && False("C");
        // eq. True("A") || (False("B") && False("C"))
    cout << result << " :===== " << endl;
    //TrueA
    //"Short-circuit evaluation skip of B"
    //"Short-circuit evaluation skip of C"
    //A is true so we do not have to evaluate
    //      the right of || to know that the result is true
    //If || had precedence over && the equivalent evaluation would be:
    // (True("A") || False("B")) && False("C")
    //What would print
    //TrueA
    //"Short-circuit evaluation skip of B"
    //FalseC
    //Because the parentheses are placed differently
    //the parts that get evaluated are differently
    //which makes that the end result in this case would be False because C is false
}
```

Унарные операторы

Унарные операторы действуют на объект, на который они вызваны, и имеют высокий приоритет. (См. Примечания)

При использовании postfix действие происходит только после того, как вся операция будет оценена, что приведет к некоторой интересной арифметике:

```
int a = 1;
++a;           // result: 2
a--;           // result: 1
int minusa=-a; // result: -1

bool b = true;
!b; // result: true

a=4;
int c = a++/2; // equal to: (a==4) 4 / 2 result: 2 ('a' incremented postfix)
cout << a << endl; // prints 5!
int d = ++a/2; // equal to: (a+1) == 6 / 2 result: 3

int arr[4] = {1,2,3,4};

int *ptr1 = &arr[0]; // points to arr[0] which is 1
int *ptr2 = ptr1++;   // ptr2 points to arr[0] which is still 1; ptr1 incremented
std::cout << *ptr1++ << std::endl; // prints 2

int e = arr[0]++;     // receives the value of arr[0] before it is incremented
std::cout << e << std::endl; // prints 1
std::cout << *ptr2 << std::endl; // prints arr[0] which is now 2
```

Прочитайте приоритет оператора онлайн: <https://riptutorial.com/ru/cplusplus/topic/3895/приоритет-оператора>

глава 91: Пространства имен

Вступление

Используется для предотвращения конфликтов имен при использовании нескольких библиотек, пространство имен является декларативным префиксом для функций, классов, типов и т. Д.

Синтаксис

- *Идентификатор* пространства имен (*opt*) { *declaration-seq* }
- `inline namespace identifier ( opt ) { declaration-seq } / *`, поскольку C ++ 11 * /
- `inline ( opt ) namespace attribute-specifier-seq identifier ( opt ) { declaration-seq } / *`, поскольку C ++ 17 * /
- `namespace enclosing-namespace-specifier :: identifier { declaration-seq } / *`, поскольку C ++ 17 * /
- *Идентификатор* пространства имен = *квалифицированный-namespace-specifier* ;
- использование `namespace -name-specifier ( opt ) namespace-name ;`
- `attribute-specifier-seq using namespace имя-имя-спецификатор ( opt ) namespace-name ;` / *, поскольку C ++ 11 * /

замечания

Пространство `namespace` **ключевых слов** имеет три разных значения в зависимости от контекста:

1. Когда за ним следует необязательное имя и последовательность объявлений, заключенная в фигурной скобке, он **определяет новое пространство имен** или **расширяет существующее пространство имен** с этими объявлениями. Если имя опущено, пространство имен является **безымянным пространством имен** .
2. Когда за ним следует имя и знак равенства, он объявляет **псевдоним пространства имен** .
3. Когда ему предшествует `using` и имя имени пространства имен, оно формирует **директиву using** , которая позволяет найти имена в данном пространстве имен путем поиска неквалифицированного имени (но не обновляет эти имена в текущей области). *Директива-директива* не может возникать в классе.

`using namespace std;` не рекомендуется. Зачем? Потому что `namespace std` огромно! Это означает, что существует вероятность того, что имена будут сталкиваться:

```
//Really bad!
using namespace std;

//Calculates p^e and outputs it to std::cout
void pow(double p, double e) { /*...*/ }

//Calls pow
pow(5.0, 2.0); //Error! There is already a pow function in namespace std with the same
signature,
                //so the call is ambiguous
```

Examples

Что такое пространства имен?

Пространство имен C ++ представляет собой набор объектов C ++ (функции, классы, переменные), имена которых имеют префикс имени пространства имен. При написании кода в пространстве имен именованные объекты, принадлежащие к этому пространству имен, не обязательно должны иметь префикс имени пространства имен, но сущности вне него должны использовать полное имя. Полноценное имя имеет формат

<namespace>::<entity> . Пример:

```
namespace Example
{
    const int test = 5;

    const int test2 = test + 12; //Works within `Example` namespace
}

const int test3 = test + 3; //Fails; `test` not found outside of namespace.

const int test3 = Example::test + 3; //Works; fully qualified name used.
```

Пространства имен полезны для группировки связанных определений вместе. Пройдите аналогию с торговым центром. Как правило, торговый центр делится на несколько магазинов, каждый магазин продает предметы из определенной категории. Один магазин может продавать электронику, а другой магазин может продавать обувь. Эти логические разделения в типах магазинов помогают покупателям находить предметы, которые они ищут. Пространства имен помогают программистам на C ++, например покупателям, находить функции, классы и переменные, которые они ищут, организуя их логически.

Пример:

```
namespace Electronics
{
    int TotalStock;
    class Headphones
    {
        // Description of a Headphone (color, brand, model number, etc.)
    };
    class Television
    {
```

```

        // Description of a Television (color, brand, model number, etc.)
    };
}

namespace Shoes
{
    int TotalStock;
    class Sandal
    {
        // Description of a Sandal (color, brand, model number, etc.)
    };
    class Slipper
    {
        // Description of a Slipper (color, brand, model number, etc.)
    };
}

```

Существует одно пространство имен предопределен, что глобальное пространство имен, не имеет имени, но можно обозначить `::`. Пример:

```

void bar() {
    // defined in global namespace
}
namespace foo {
    void bar() {
        // defined in namespace foo
    }
    void barbar() {
        bar(); // calls foo::bar()
        ::bar(); // calls bar() defined in global namespace
    }
}

```

Создание пространств имен

Создание пространства имен очень просто:

```

//Creates namespace foo
namespace Foo
{
    //Declares function bar in namespace foo
    void bar() {}
}

```

Чтобы вызвать `bar`, вы должны сначала указать пространство имен, а затем оператор разрешения области `::`:

```

Foo::bar();

```

Разрешено создавать одно пространство имен в другом, например:

```

namespace A
{
    namespace B

```



```

{
    namespace C
    {
        void bar() {}
    }
}

```

C ++ 17

Вышеприведенный код можно упростить до следующего:

```

namespace A::B::C
{
    void bar() {}
}

```

Расширение пространств имен

Полезной функцией `namespace` с является то, что вы можете расширять их (добавлять к ней участников).

```

namespace Foo
{
    void bar() {}
}

//some other stuff

namespace Foo
{
    void bar2() {}
}

```

Использование директивы

Ключевое слово «**использование**» имеет три варианта. В сочетании с ключевым словом «`namespace`» вы пишете «`using directive`»:

Если вы не хотите писать `Foo::` перед каждым материалом в пространстве имен `Foo` , вы можете использовать `using namespace Foo;` импортировать каждую вещь из `Foo` .

```

namespace Foo
{
    void bar() {}
    void baz() {}
}

//Have to use Foo::bar()
Foo::bar();

//Import Foo
using namespace Foo;

```

```
bar(); //OK
baz(); //OK
```

Также можно импортировать выбранные объекты в пространстве имен, а не во всем пространстве имен:

```
using Foo::bar;
bar(); //OK, was specifically imported
baz(); // Not OK, was not imported
```

Предупреждение: `using namespace s` в файлах заголовков в большинстве случаев рассматривается как плохой стиль. Если это будет сделано, пространство имен будет импортировано в *каждый* файл, включающий заголовок. Поскольку нет способа «не `using`» пространство имен, это может привести к загрязнению пространства имен (больше или неожиданным символам в глобальном пространстве имен) или, что еще хуже, конфликтам. См. Этот пример для иллюстрации проблемы:

```
/****** foo.h *****/
namespace Foo
{
    class C;
}

/****** bar.h *****/
namespace Bar
{
    class C;
}

/****** baz.h *****/
#include "foo.h"
using namespace Foo;

/****** main.cpp *****/
#include "bar.h"
#include "baz.h"

using namespace Bar;
C c; // error: Ambiguity between Bar::C and Foo::C
```

Директива-директива не может возникать в классе.

Поиск зависимых аргументов

При вызове функции без явного определителя пространства имен компилятор может выбрать вызов функции в пространстве имен, если один из типов параметров этой функции также находится в этом пространстве имен. Это называется «зависимым от аргумента» или ADL:

```
namespace Test
{
    int call(int i);
}
```

```

class SomeClass {...};

int call_too(const SomeClass &data);
}

call(5); //Fails. Not a qualified function name.

Test::SomeClass data;

call_too(data); //Succeeds

```

`call` невозможен, потому что ни один из его типов параметров не приходит из пространства имен `Test`. `call_too` работает, потому что `SomeClass` является членом `Test` и поэтому он подходит для правил ADL.

Когда ADL не возникает

ADL не возникает, если обычный неквалифицированный поиск находит член класса, функцию, объявленную в области блока, или что-то, что не относится к типу функции. Например:

```

void foo();
namespace N {
    struct X {};
    void foo(X ) { std::cout << '1'; }
    void qux(X ) { std::cout << '2'; }
}

struct C {
    void foo() {}
    void bar() {
        foo(N::X{}); // error: ADL is disabled and C::foo() takes no arguments
    }
};

void bar() {
    extern void foo(); // redeclares ::foo
    foo(N::X{});      // error: ADL is disabled and ::foo() doesn't take any arguments
}

int qux;

void baz() {
    qux(N::X{}); // error: variable declaration disables ADL for "qux"
}

```

Встроенное пространство имен

C++ 11

`inline namespace` включает содержимое встроенного пространства имен в охватывающем пространстве имен, поэтому

```
namespace Outer
{
    inline namespace Inner
    {
        void foo();
    }
}
```

В ОСНОВНОМ ЭКВИВАЛЕНТЕН

```
namespace Outer
{
    namespace Inner
    {
        void foo();
    }

    using Inner::foo;
}
```

но элемент из `Outer::Inner::` и те, которые связаны с `Outer::` , идентичны.

Итак, следующее эквивалентно

```
Outer::foo();
Outer::Inner::foo();
```

Альтернатива `using namespace Inner;` не будет эквивалентен некоторым сложным частям в качестве типовой специализации:

За

```
#include <outer.h> // See below

class MyCustomType;
namespace Outer
{
    template <>
    void foo<MyCustomType>() { std::cout << "Specialization"; }
}
```

- Встроенное пространство имен позволяет специализировать `Outer::foo`

```
// outer.h
// include guard omitted for simplification

namespace Outer
{
    inline namespace Inner
    {
        template <typename T>
        void foo() { std::cout << "Generic"; }
    }
}
```

```
}
```

- В то время как `using namespace` не позволяет специализации `Outer::foo`

```
// outer.h
// include guard omitted for simplification

namespace Outer
{
    namespace Inner
    {
        template <typename T>
        void foo() { std::cout << "Generic"; }
    }
    using namespace Inner;
    // Specialization of `Outer::foo` is not possible
    // it should be `Outer::Inner::foo`.
}
```

Встроенное пространство имен - это способ позволить нескольким версиям сожительствовать и дефолтировать `inline`

```
namespace MyNamespace
{
    // Inline the last version
    inline namespace Version2
    {
        void foo(); // New version
        void bar();
    }

    namespace Version1 // The old one
    {
        void foo();
    }
}
```

И с использованием

```
MyNamespace::Version1::foo(); // old version
MyNamespace::Version2::foo(); // new version
MyNamespace::foo();           // default version : MyNamespace::Version1::foo();
```

Без имени / анонимных пространств имен

Неименованное пространство имен может использоваться, чтобы гарантировать, что имена имеют внутреннюю связь (на которые может ссылаться только текущая единица перевода). Такое пространство имен определяется так же, как и любое другое пространство имен, но без имени:

```
namespace {
```

```
int foo = 42;
}
```

`foo` отображается только в блоке перевода, в котором он отображается.

Рекомендуется никогда не использовать неназванные пространства имен в заголовочных файлах, так как это дает версию содержимого для каждой единицы перевода, в которую она включена. Это особенно важно, если вы определяете неконтекстные глобальные переменные.

```
// foo.h
namespace {
    std::string globalString;
}

// 1.cpp
#include "foo.h" //< Generates unnamed_namespace{1.cpp}::globalString ...

globalString = "Initialize";

// 2.cpp
#include "foo.h" //< Generates unnamed_namespace{2.cpp}::globalString ...

std::cout << globalString; //< Will always print the empty string
```

Компактные вложенные пространства имен

C++ 17

```
namespace a {
    namespace b {
        template<class T>
        struct qualifies : std::false_type {};
    }
}

namespace other {
    struct bob {};
}

namespace a::b {
    template<>
    struct qualifies<::other::bob> : std::true_type {};
}
```

Вы можете ввести оба пространства имен `a` и `b` за один шаг с `namespace a::b` начиная с C++ 17.

Слияние длинного пространства имен

Обычно это используется для переименования или сокращения длинных ссылок пространства имен, например, для ссылок на компоненты библиотеки.

```

namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace Name1 = boost::multiprecision;

//    Both Type declarations are equivalent
boost::multiprecision::Number X    //    Writing the full namespace path, longer
Name1::Number Y                    //    using the name alias, shorter

```

Область объявления псевдонима

Алиас Декларация затрагивает предшествующими *используя* операторы

```

namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

using namespace boost;

//    Both Namespace are equivalent
namespace Name1 = boost::multiprecision;
namespace Name2 = multiprecision;

```

Тем не менее, легче запутаться в том, какое пространство имен вы используете, когда у вас есть что-то вроде этого:

```

namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace numeric
{
    namespace multiprecision
    {
        class Number ...
    }
}

using namespace numeric;
using namespace boost;

//    Not recommended as

```

```
//    its not explicitly clear whether Name1 refers to
//    numeric::multiprecision or boost::multiprecision
namespace Name1 = multiprecision;

//    For clarity, its recommended to use absolute paths
//    instead
namespace Name2 = numeric::multiprecision;
namespace Name3 = boost::multiprecision;
```

Псевдоним пространства имен

Пространству имен может быть присвоен псевдоним (*то есть* другое имя для того же пространства имен) с использованием *идентификатора* `namespace = синтаксис`. К элементам пространства имен с псевдонимом можно получить доступ, присвоив им имя псевдонима. В следующем примере вложенное пространство имен `AReallyLongName::AnotherReallyLongName` неудобно вводить, поэтому функция `qux` локально объявляет псевдоним `N`. После этого члены этого пространства имен могут быть доступны просто с помощью `N::`.

```
namespace AReallyLongName {
    namespace AnotherReallyLongName {
        int foo();
        int bar();
        void baz(int x, int y);
    }
}

void qux() {
    namespace N = AReallyLongName::AnotherReallyLongName;
    N::baz(N::foo(), N::bar());
}
```

Прочитайте Пространства имен онлайн: <https://riptutorial.com/ru/cplusplus/topic/495/пространства-имен>

глава 92: профилирование

Examples

Профилирование gcc и gprof

Профайлер GNU gprof, [gprof](#), позволяет вам профилировать свой код. Чтобы использовать его, вам необходимо выполнить следующие шаги:

1. Создайте приложение с настройками для генерации профилирующей информации
2. Генерирование профилирующей информации путем запуска встроенного приложения
3. Просмотр сгенерированной профилирующей информации с помощью gprof

Чтобы создать приложение с настройками для генерации профилирующей информации, мы добавляем флаг `-pg`. Так, например, мы могли бы использовать

```
$ gcc -pg *.cpp -o app
```

или же

```
$ gcc -O2 -pg *.cpp -o app
```

и так далее.

Как только приложение, скажем, `app`, будет создано, выполните его как обычно:

```
$ ./app
```

Это должно создать файл `gmon.out`.

Чтобы просмотреть результаты профилирования, выполните

```
$ gprof app gmon.out
```

(обратите внимание, что мы предоставляем как приложение, так и сгенерированный вывод).

Конечно, вы можете также перенаправить или перенаправить:

```
$ gprof app gmon.out | less
```

и так далее.

Результатом последней команды должна быть таблица, строками которой являются функции, а в столбцах которой указано количество вызовов, общее время, затраченное на самостоятельное время (т. Е. Время, затрачиваемое на функцию, исключая вызовы для детей).

Генерирование диаграмм callgraph с помощью gperf2dot

Для более сложных приложений трудно выполнить плоские профили выполнения. Вот почему многие инструменты профилирования также генерируют некоторую форму аннотированной информации об обслуживании.

[gperf2dot](#) преобразует текстовый вывод из многих профилировщиков (Linux perf, callgrind, oprofile и т. д.) в диаграмму callgraph. Вы можете использовать его, запустив свой профилировщик (пример для `gprof`):

```
# compile with profiling flags
g++ *.cpp -pg
# run to generate profiling data
./main
# translate profiling data to text, create image
gprof ./main | gprof2dot -s | dot -Tpng -o output.png
```


Perf Tools

Google Perf Tools также предоставляет профилировщик процессора со слегка более дружелюбным интерфейсом. Чтобы использовать его:

1. **Установка Google Perf Tools**
2. Скомпилируйте свой код как обычно
3. Добавить библиотеку профилировщика `libprofiler` в путь загрузки библиотеки во время выполнения
4. Используйте `pprof` для создания плоского профиля выполнения или диаграммы вызова

Например:

```
# compile code
g++ -O3 -std=c++11 main.cpp -o main

# run with profiler
LD_PRELOAD=/usr/local/lib/libprofiler.so CPUPROFILE=main.prof CPUPROFILE_FREQUENCY=100000
./main
```

где:

- `CPUPROFILE` указывает выходной файл для профилирования данных
- `CPUPROFILE_FREQUENCY` указывает частоту выборки профайлера;

Используйте `pprof` для `pprof` обработки данных профилирования.

Вы можете создать профиль плоских вызовов в виде текста:

```
$ pprof --text ./main main.prof
PROFILE: interrupts/evictions/bytes = 67/15/2016
pprof --text --lines ./main main.prof
Using local file ./main.
Using local file main.prof.
Total: 67 samples
 22 32.8% 32.8%      67 100.0% longRunningFoo ??:0
 20 29.9% 62.7%      20 29.9% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/./sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1627
  4 6.0% 68.7%       4 6.0% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/./sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1619
  3 4.5% 73.1%       3 4.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:388
  3 4.5% 77.6%       3 4.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:401
  2 3.0% 80.6%       2 3.0% __munmap /build/eglibc-3GlaMS/eglibc-
2.19/misc/./sysdeps/unix/syscall-template.S:81
  2 3.0% 83.6%      12 17.9% __random /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random.c:298
  2 3.0% 86.6%       2 3.0% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:385
  2 3.0% 89.6%       2 3.0% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:26
  1 1.5% 91.0%       1 1.5% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
```

```

2.19/string/../../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1617
    1   1.5%  92.5%      1   1.5%  __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1623
    1   1.5%  94.0%      1   1.5%  __random /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random.c:293
    1   1.5%  95.5%      1   1.5%  __random /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random.c:296
    1   1.5%  97.0%      1   1.5%  __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:371
    1   1.5%  98.5%      1   1.5%  __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:381
    1   1.5% 100.0%      1   1.5%  rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:28
    0   0.0% 100.0%     67 100.0% __libc_start_main /build/eglibc-3GlaMS/eglibc-
2.19/csu/libc-start.c:287
    0   0.0% 100.0%     67 100.0% _start ??:0
    0   0.0% 100.0%     67 100.0% main ??:0
    0   0.0% 100.0%     14  20.9% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:27
    0   0.0% 100.0%     27  40.3% std::vector::_M_emplace_back_aux ??:0

```

... или вы можете создать аннотированный callgraph в pdf с:

```
pprof --pdf ./main main.prof > out.pdf
```

Прочитайте профилирование онлайн: <https://riptutorial.com/ru/cplusplus/topic/5347/>
 профилирование

глава 93: Разрешение перегрузки

замечания

Разрешение перегрузки происходит в нескольких разных ситуациях

- Вызовы для названных перегруженных функций. Кандидатами являются все функции, найденные при поиске по имени.
- Вызывает объект класса. Кандидатами обычно являются все перегруженные операторы вызова функции класса.
- Использование оператора. Кандидатами являются перегруженные функции оператора в области пространства имен, перегруженные функции оператора в левом классе (если есть) и встроенные операторы.
- Разрешение перегрузки, чтобы найти правильную функцию оператора преобразования или конструктор для вызова для инициализации
 - Для прямой инициализации без списка (`Class c(value)`) кандидаты являются конструкторами `Class`.
 - Для инициализации копии без списка (`Class c = value`) и для поиска функции, определяемой пользователем, для вызова в пользовательской последовательности преобразования. Кандидаты являются конструкторами `Class` и если источником является объект класса, его операторы преобразования.
 - Для инициализации `Nonclass c = classObject` из объекта класса (`Nonclass c = classObject`). Кандидаты - это функции оператора преобразования объекта инициализации.
 - Для инициализации ссылки с объектом класса (`R &r = classObject`), когда класс имеет функции оператора преобразования, которые дают значения, которые могут быть связаны непосредственно с `r`. Кандидатами являются такие функции оператора преобразования.
 - Для инициализации списка неагрегатного объекта класса (`Class c{1, 2, 3}`) кандидаты являются конструкторами списка инициализаторов для первого прохода через разрешение перегрузки. Если это не находит жизнеспособного кандидата, выполняется второй проход через разрешение перегрузки с конструкторами `Class` качестве кандидатов.

Examples

Полное совпадение

Перегрузка без конверсий, необходимых для типов параметров, или только преобразования, необходимые между типами, которые все еще считаются точными

совпадениями, предпочтительнее перегрузки, которая требует других преобразований для вызова.

```
void f(int x);
void f(double x);
f(42); // calls f(int)
```

Когда аргумент связывается с ссылкой на тот же тип, считается, что совпадение не требует преобразования, даже если ссылка больше cv-qualified.

```
void f(int& x);
void f(double x);
int x = 42;
f(x); // argument type is int; exact match with int&

void g(const int& x);
void g(int x);
g(x); // ambiguous; both overloads give exact match
```

Для целей разрешения перегрузки тип «массив T» считается точно совпадающим с типом «указатель на T», и тип функции T считается точно совпадающим с типом указателя функции T*, хотя оба требуют преобразования.

```
void f(int* p);
void f(void* p);

void g(int* p);
void g(int (&p)[100]);

int a[100];
f(a); // calls f(int*); exact match with array-to-pointer conversion
g(a); // ambiguous; both overloads give exact match
```

Категоризация аргумента по стоимости параметра

Разрешение перегрузки разделяет стоимость передачи аргумента параметру в одну из четырех различных категорий, называемых «последовательностями». Каждая последовательность может включать в себя нуль, одно или несколько преобразований

- Стандартная последовательность преобразования

```
void f(int a); f(42);
```

- Определенная пользователем последовательность преобразования

```
void f(std::string s); f("hello");
```

- Последовательность преобразования эллипсиса

```
void f(...); f(42);
```

- Последовательность инициализации списка

```
void f(std::vector<int> v); f({1, 2, 3});
```

Общий принцип заключается в том, что стандартные конверсионные последовательности являются самыми дешевыми, за ними следуют пользовательские последовательности преобразования, за которыми следуют последовательности преобразования многоточия.

Частным случаем является последовательность инициализации списка, которая не является преобразованием (список инициализаторов не является выражением с типом). Его стоимость определяется путем определения ее эквивалентности одной из трех других последовательностей преобразования в зависимости от типа параметра и формы списка инициализаторов.

Проверка имени и проверка доступа

Разрешение перегрузки происходит *после* поиска имени. Это означает, что функция лучшего соответствия не будет выбрана с помощью разрешения перегрузки, если она потеряет поиск по имени:

```
void f(int x);
struct S {
    void f(double x);
    void g() { f(42); } // calls S::f because global f is not visible here,
                        // even though it would be a better match
};
```

Разрешение перегрузки происходит *перед* проверкой доступа. Недопустимая функция может быть выбрана с помощью разрешения перегрузки, если она лучше соответствует доступной функции.

```
class C {
public:
    static void f(double x);
private:
    static void f(int x);
};
C::f(42); // Error! Calls private C::f(int) even though public C::f(double) is viable.
```

Аналогично, разрешение перегрузки происходит без проверки того, правильно ли сформирован результирующий вызов в отношении `explicit`:

```
struct X {
    explicit X(int );
    X(char );
};
```



```
void foo(X );
foo({4}); // X(int) is better much, but expression is
          // ill-formed because selected constructor is explicit
```

Перегрузка по ссылке

Вы должны быть очень осторожны при предоставлении ссылочной экспедиторской перегрузки, как это может соответствовать слишком хорошо:

```
struct A {
    A() = default;           // #1
    A(A const& ) = default;  // #2

    template <class T>
    A(T&& );                 // #3
};
```

Цель заключалась в том, что `A` является скопируемым и что у нас есть этот другой конструктор, который может инициализировать какой-либо другой элемент. Тем не менее:

```
A a;           // calls #1
A b(a);        // calls #3!
```

Для вызова строительства есть два жизнеспособных матча:

```
A(A const& ); // #2
A(A& );       // #3, with T = A&
```

Оба являются точными совпадениями, но `#3` ссылается на менее `cv`-qualified объект, чем `#2`, поэтому он имеет лучшую стандартную последовательность преобразования и является наилучшей жизнеспособной функцией.

Решение здесь состоит в том, чтобы всегда ограничивать эти конструкторы (например, используя `SFINAE`):

```
template <class T,
    class = std::enable_if_t<!std::is_convertible<std::decay_t<T>*, A*>::value>
    >
A(T&& );
```

Типичная черта здесь заключается в том, чтобы исключить любой `A` или класс, публично и однозначно полученный из `A` из рассмотрения, что сделало бы этот конструктор плохо сформированным в примере, описанном ранее (и, следовательно, удаляемом из набора перегрузки). В результате вызывается конструктор копирования - это то, что мы хотели.

Шаги разрешения перегрузки

Шаги разрешения перегрузки:

1. Найти функции кандидата с помощью поиска по имени. Неквалифицированные вызовы будут выполнять как регулярный неквалифицированный поиск, так и зависящий от аргументов поиск (если применимо).
2. Отфильтруйте набор функций-кандидатов до набора *жизнеспособных* функций. Жизнеспособная функция, для которой существует неявная последовательность преобразований между аргументами, вызываемыми функцией, и параметрами, которые выполняет функция.

```
void f(char);           // (1)
void f(int ) = delete; // (2)
void f();               // (3)
void f(int& );          // (4)

f(4); // 1,2 are viable (even though 2 is deleted!)
      // 3 is not viable because the argument lists don't match
      // 4 is not viable because we cannot bind a temporary to
      //      a non-const lvalue reference
```

3. Выберите лучшего жизнеспособного кандидата. Жизнеспособная функция F_1 является лучшей функцией, чем другая жизнеспособная функция F_2 если неявная последовательность преобразований для каждого аргумента в F_1 не хуже соответствующей имплицитной последовательности преобразований в F_2 и ...:

3.1. Для некоторого аргумента неявная последовательность преобразования для этого аргумента в F_1 является лучшей последовательностью преобразования, чем для этого аргумента в F_2 , или

```
void f(int ); // (1)
void f(char ); // (2)

f(4); // call (1), better conversion sequence
```

3.2. В пользовательском преобразовании стандартная последовательность преобразования от возврата F_1 к типу назначения является лучшей последовательностью преобразования, чем последовательность возвращаемого типа F_2 , или

```
struct A
{
    operator int();
    operator double();
} a;

int i = a; // a.operator int() is better than a.operator double() and a conversion
float f = a; // ambiguous
```

3.3. В прямой привязке ссылки F_1 имеет тот же тип ссылки, что и F_2 , или

```

struct A
{
    operator X&(); // #1
    operator X&&(); // #2
};
A a;
X& lx = a; // calls #1
X&& rx = a; // calls #2

```

3.4. F1 не является специализированным шаблоном, но F2 есть или

```

template <class T> void f(T ); // #1
void f(int ); // #2

f(42); // calls #2, the non-template

```

3.5. F1 и F2 являются специализациями шаблонов функций, но F1 более специализирован, чем F2 .

```

template <class T> void f(T ); // #1
template <class T> void f(T* ); // #2

int* p;
f(p); // calls #2, more specialized

```

Заказ здесь значителен. Лучшая проверка последовательности преобразований происходит до проверки шаблона и без шаблона. Это приводит к общей ошибке при перегрузке по ссылке пересылки:

```

struct A {
    A(A const& ); // #1

    template <class T>
    A(T&& ); // #2, not constrained
};

A a;
A b(a); // calls #2!
// #1 is not a template but #2 resolves to
// A(A& ), which is a less cv-qualified reference than #1
// which makes it a better implicit conversion sequence

```

Если в конце нет единственного лучшего жизнеспособного кандидата, вызов неоднозначен:

```

void f(double ) { }
void f(float ) { }

f(42); // error: ambiguous

```

Арифметические акции и конверсии

Преобразование целочисленного типа в соответствующий продвинутый тип лучше, чем преобразование его в какой-либо другой целочисленный тип.

```
void f(int x);
void f(short x);
signed char c = 42;
f(c); // calls f(int); promotion to int is better than conversion to short
short s = 42;
f(s); // calls f(short); exact match is better than promotion to int
```

Продвижение `float` в `double` лучше, чем преобразование его в другой тип с плавающей точкой.

```
void f(double x);
void f(long double x);
f(3.14f); // calls f(double); promotion to double is better than conversion to long double
```

Арифметические преобразования, кроме рекламных акций, не лучше и хуже друг друга.

```
void f(float x);
void f(long double x);
f(3.14); // ambiguous

void g(long x);
void g(long double x);
g(42); // ambiguous
g(3.14); // ambiguous
```

Поэтому для обеспечения отсутствия двусмысленности при вызове функции `f` с аргументами целочисленной или с плавающей запятой любого стандартного типа требуется всего восемь перегрузок, так что для каждого возможного типа аргумента либо перегрузка соответствует точно или будет выбрана уникальная перегрузка с указанным типом аргумента.

```
void f(int x);
void f(unsigned int x);
void f(long x);
void f(unsigned long x);
void f(long long x);
void f(unsigned long long x);
void f(double x);
void f(long double x);
```

Перегрузка в иерархии классов

В следующих примерах будет использоваться эта иерархия классов:

```
struct A { int m; };
struct B : A {};
struct C : B {};
```

Преобразование из типа производного класса в тип базового класса является предпочтительным для пользовательских преобразований. Это применяется при передаче по значению или по ссылке, а также при преобразовании указателя на производный в указатель на базу.

```
struct Unrelated {
    Unrelated(B b);
};
void f(A a);
void f(Unrelated u);
B b;
f(b); // calls f(A)
```

Преобразование указателя из производного класса в базовый класс также лучше, чем преобразование в `void*`.

```
void f(A* p);
void f(void* p);
B b;
f(&b); // calls f(A*)
```

Если в одной цепочке наследования имеется несколько перегрузок, предпочтительной является перегрузка основной производной базы. Это основано на аналогичном принципе, как виртуальная отправка: выбрана «наиболее специализированная» реализация. Тем не менее, разрешение перегрузки всегда происходит во время компиляции и никогда не будет скрытно скрываться.

```
void f(const A& a);
void f(const B& b);
C c;
f(c); // calls f(const B&)
B b;
A& r = b;
f(r); // calls f(const A&); the f(const B&) overload is not viable
```

Для указателей на члены, которые являются контравариантными относительно класса, аналогичное правило применяется в противоположном направлении: наименее производный производный класс является предпочтительным.

```
void f(int B::*p);
void f(int C::*p);
int A::*p = &A::m;
f(p); // calls f(int B::*)
```

Перегрузка на постоянство и волатильность

Передача аргумента указателя на параметр `T*`, если это возможно, лучше, чем передача его в параметр `const T*`.

```

struct Base {};
struct Derived : Base {};
void f(Base* pb);
void f(const Base* pb);
void f(const Derived* pd);
void f(bool b);

Base b;
f(&b); // f(Base*) is better than f(const Base*)
Derived d;
f(&d); // f(const Derived*) is better than f(Base*) though;
        // constness is only a "tie-breaker" rule

```

Аналогично, передача аргумента параметру `T&`, если это возможно, лучше, чем передача его в параметр `const T&`, даже если оба имеют точный рейтинг соответствия.

```

void f(int& r);
void f(const int& r);
int x;
f(x); // both overloads match exactly, but f(int&) is still better
const int y = 42;
f(y); // only f(const int&) is viable

```

Это правило применяется также к `const`-квалифицированным функциям-членам, где важно разрешить изменчивый доступ к неконстантным объектам и неизменный доступ к объектам `const`.

```

class IntVector {
public:
    // ...
    int* data() { return m_data; }
    const int* data() const { return m_data; }
private:
    // ...
    int* m_data;
};

IntVector v1;
int* data1 = v1.data(); // Vector::data() is better than Vector::data() const;
                        // data1 can be used to modify the vector's data

const IntVector v2;
const int* data2 = v2.data(); // only Vector::data() const is viable;
                             // data2 can't be used to modify the vector's data

```

Точно так же летучая перегрузка будет менее предпочтительной, чем энергонезависимая перегрузка.

```

class AtomicInt {
public:
    // ...
    int load();
    int load() volatile;
private:
    // ...
};

AtomicInt a1;

```

```
a1.load(); // non-volatile overload preferred; no side effect
volatile AtomicInt a2;
a2.load(); // only volatile overload is viable; side effect
static_cast<volatile AtomicInt&>(a1).load(); // force volatile semantics for a1
```

Прочитайте Разрешение перегрузки онлайн: <https://riptutorial.com/ru/cplusplus/topic/2021/разрешение-перегрузки>

глава 94: Реализация шаблона проектирования в C ++

Вступление

На этой странице вы можете найти примеры того, как шаблоны проектирования реализованы на C ++. Подробные сведения об этих шаблонах можно найти [в документации по шаблонам проектирования](#).

замечания

Шаблон проектирования является общим многоразовым решением общей проблемы в данном контексте при разработке программного обеспечения.

Examples

Схема наблюдателя

Целью Observer Pattern является определение зависимости один-на-один между объектами, чтобы при изменении состояния одного объекта все его иждивенцы были уведомлены и обновлены автоматически.

Субъект и наблюдатели определяют отношения «один ко многим». Наблюдатели зависят от предмета, так что, когда состояние субъекта изменяется, наблюдатели получают уведомление. В зависимости от уведомления наблюдатели могут также обновляться новыми значениями.

Вот пример из книги «Шаблоны проектирования» Гамма.

```
#include <iostream>
#include <vector>

class Subject;

class Observer
{
public:
    virtual ~Observer() = default;
    virtual void Update(Subject&) = 0;
};

class Subject
{
public:
    virtual ~Subject() = default;
```



```

void Attach(Observer& o) { observers.push_back(&o); }
void Detach(Observer& o)
{
    observers.erase(std::remove(observers.begin(), observers.end(), &o));
}
void Notify()
{
    for (auto* o : observers) {
        o->Update(*this);
    }
}
private:
    std::vector<Observer*> observers;
};

class ClockTimer : public Subject
{
public:

    void SetTime(int hour, int minute, int second)
    {
        this->hour = hour;
        this->minute = minute;
        this->second = second;

        Notify();
    }

    int GetHour() const { return hour; }
    int GetMinute() const { return minute; }
    int GetSecond() const { return second; }

private:
    int hour;
    int minute;
    int second;
};

class DigitalClock: public Observer
{
public:
    explicit DigitalClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
    ~DigitalClock() { subject.Detach(*this); }
    void Update(Subject& theChangedSubject) override
    {
        if (&theChangedSubject == &subject) {
            Draw();
        }
    }

    void Draw()
    {
        int hour = subject.GetHour();
        int minute = subject.GetMinute();
        int second = subject.GetSecond();

        std::cout << "Digital time is " << hour << ":"
                  << minute << ":"
                  << second << std::endl;
    }
}

```

```

private:
    ClockTimer& subject;
};

class AnalogClock: public Observer
{
public:
    explicit AnalogClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
    ~AnalogClock() { subject.Detach(*this); }
    void Update(Subject& theChangedSubject) override
    {
        if (&theChangedSubject == &subject) {
            Draw();
        }
    }
    void Draw()
    {
        int hour = subject.GetHour();
        int minute = subject.GetMinute();
        int second = subject.GetSecond();

        std::cout << "Analog time is " << hour << ":"
                  << minute << ":"
                  << second << std::endl;
    }
private:
    ClockTimer& subject;
};

int main()
{
    ClockTimer timer;

    DigitalClock digitalClock(timer);
    AnalogClock analogClock(timer);

    timer.SetTime(14, 41, 36);
}

```

Выход:

```

Digital time is 14:41:36
Analog time is 14:41:36

```

Вот краткое описание шаблона:

1. Объекты (объект `DigitalClock` или `AnalogClock`) используют интерфейсы `Subject` (`Attach()` или `Detach()`) либо для подписания (регистрации) в качестве наблюдателей, либо для `subject.Attach(*this);` от подписки (удаления) от наблюдателей (`subject.Attach(*this); subject.Detach(*this);`);
2. У каждого субъекта может быть много наблюдателей (`vector<Observer*> observers;`).
3. Все наблюдатели должны реализовать интерфейс `Observer`. Этот интерфейс имеет только один метод `Update()` , который вызывается при изменении состояния `Subject` (`Update(Subject &)`)

4. В дополнение к методам `Attach()` и `Detach()` конкретный объект реализует метод `Notify()` который используется для обновления всех текущих наблюдателей всякий раз, когда изменяется состояние. Но в этом случае все они выполняются в родительском классе `Subject` (`Subject::Attach (Observer&) , void Subject::Detach(Observer&) И void Subject::Notify() .`
5. Объект `Concrete` также может иметь методы настройки и получения состояния.
6. Конкретными наблюдателями могут быть любые классы, реализующие интерфейс `Observer`. Каждый наблюдатель подписывается (регистрируется) конкретным субъектом для получения обновлений (`subject.Attach(*this);`).
7. Два объекта шаблона наблюдателя **слабо связаны** друг с другом, они могут взаимодействовать, но не знают друг друга.

Вариация:

Сигнал и слоты

Сигналы и слоты - это языковая конструкция, введенная в Qt, что упрощает реализацию шаблона `Observer`, избегая при этом шаблона кода. Концепция состоит в том, что элементы управления (также известные как виджеты) могут отправлять сигналы, содержащие информацию о событиях, которые могут быть получены другими элементами управления, с использованием специальных функций, известных как слоты. Слот в Qt должен быть членом класса, объявленным как таковой. Система сигналов / слотов хорошо подходит для использования графических пользовательских интерфейсов. Аналогично, система сигнала / слота может использоваться для асинхронного ввода / вывода (включая сокеты, каналы, последовательные устройства и т. Д.) Уведомления о событиях или для связывания событий таймаута с соответствующими экземплярами объектов и методами или функциями. Нет необходимости записывать код регистрации / регистрации / вызова, поскольку компилятор метаданных Qt (МОС) автоматически генерирует необходимую инфраструктуру.

Язык C # также поддерживает аналогичную конструкцию, хотя с другой терминологией и синтаксисом: события играют роль сигналов, а делегаты - это слоты. Кроме того, делегат может быть локальной переменной, подобно указателю на функцию, а слот в Qt должен быть членом класса, объявленным как таковой.

Шаблон адаптера

Преобразуйте интерфейс класса в другой интерфейс. Адаптер (или `Wrapper`) позволяет работать вместе, что в противном случае не может быть связано с несовместимыми интерфейсами. Мотивация шаблона адаптера заключается в том, что мы можем повторно использовать существующее программное обеспечение, если мы можем изменить интерфейс.

1. Шаблон адаптера зависит от состава объекта.
2. Клиент вызывает операцию на объекте адаптера.
3. Адаптер вызывает Adaptee для выполнения операции.
4. В STL стек адаптируется из вектора: когда стек выполняет push (), базовый вектор делает vector :: push_back ().

Пример:

```
#include <iostream>

// Desired interface (Target)
class Rectangle
{
public:
    virtual void draw() = 0;
};

// Legacy component (Adaptee)
class LegacyRectangle
{
public:
    LegacyRectangle(int x1, int y1, int x2, int y2) {
        x1_ = x1;
        y1_ = y1;
        x2_ = x2;
        y2_ = y2;
        std::cout << "LegacyRectangle(x1,y1,x2,y2)\n";
    }
    void oldDraw() {
        std::cout << "LegacyRectangle:  oldDraw(). \n";
    }
private:
    int x1_;
    int y1_;
    int x2_;
    int y2_;
};

// Adapter wrapper
class RectangleAdapter: public Rectangle, private LegacyRectangle
{
public:
    RectangleAdapter(int x, int y, int w, int h):
        LegacyRectangle(x, y, x + w, y + h) {
        std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
    }

    void draw() {
        std::cout << "RectangleAdapter:  draw().\n";
        oldDraw();
    }
};

int main()
{

```

```
int x = 20, y = 50, w = 300, h = 200;
Rectangle *r = new RectangleAdapter(x,y,w,h);
r->draw();
}
```

```
//Output:
//LegacyRectangle(x1,y1,x2,y2)
//RectangleAdapter(x,y,x+w,x+h)
```

Резюме кода:

1. Клиент думает, что он разговаривает с `Rectangle`
2. Цель - это класс `Rectangle`. Это то, на что клиент вызывает метод.

```
Rectangle *r = new RectangleAdapter(x,y,w,h);
r->draw();
```

3. Обратите внимание, что класс адаптера использует множественное наследование.

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
    ...
}
```

4. Адаптер `RectangleAdapter` позволяет `LegacyRectangle` на запрос (`draw()` на `Rectangle`), наследуя классы BOTH.
5. Класс `LegacyRectangle` не имеет одинаковых методов (`draw()`) как `Rectangle`, но `Adapter(RectangleAdapter)` может принимать вызовы метода `Rectangle` и обходить и вызывать метод в `LegacyRectangle`, `oldDraw()`.

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
public:
    RectangleAdapter(int x, int y, int w, int h):
        LegacyRectangle(x, y, x + w, y + h) {
        std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
    }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
};
```

Шаблон проектирования **адаптера** переводит интерфейс для одного класса в совместимый, но другой интерфейс. Таким образом, это похоже на шаблон **прокси-сервера**, поскольку это однокомпонентная оболочка. Но интерфейс для класса адаптера и исходного класса может быть другим.

Как мы видели в приведенном выше примере, этот шаблон **адаптера** полезен для раскрытия другого интерфейса для существующего API, позволяющего ему работать с

другим кодом. Кроме того, используя шаблон адаптера, мы можем использовать гетерогенные интерфейсы и преобразовывать их для обеспечения согласованного API.

Мост имеет структуру, похожую на объектный адаптер, но Bridge имеет другое намерение: он предназначен для **отделения** интерфейса от его реализации, чтобы они могли легко и независимо варьироваться. **Адаптер** предназначен для **изменения интерфейса существующего** объекта.

Заводской шаблон

Factory pattern отделяет создание объекта и позволяет создавать по имени с использованием общего интерфейса:

```
class Animal{
public:
    virtual std::shared_ptr<Animal> clone() const = 0;
    virtual std::string  getname() const = 0;
};

class Bear: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Bear>(*this);
    }
    virtual std::string getname() const override
    {
        return "bear";
    }
};

class Cat: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Cat>(*this);
    }
    virtual std::string  getname() const override
    {
        return "cat";
    }
};

class AnimalFactory{
public:
    static std::shared_ptr<Animal> getAnimal( const std::string&   name )
    {
        if ( name == "bear" )
            return std::make_shared<Bear>();
        if ( name == "cat" )
            return std::shared_ptr<Cat>();

        return nullptr;
    }
}
```

```
};
```

Шаблон Builder с Fluent API

Шаблон Builder отделяет создание объекта от самого объекта. Основная идея заключается в том, что **объект не должен нести ответственность за свое собственное создание**. Правильная и достоверная сборка сложного объекта может быть сложной задачей сама по себе, поэтому эта задача может быть делегирована другому классу.

Вдохновленный разработчиком [электронной почты в C #](#), я решил создать версию на C ++. Объект электронной почты не обязательно является *очень сложным объектом*, но он может демонстрировать шаблон.

```
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

// Forward declaring the builder
class EmailBuilder;

class Email
{
public:
    friend class EmailBuilder; // the builder can access Email's privates

    static EmailBuilder make();

    string to_string() const {
        stringstream stream;
        stream << "from: " << m_from
            << "\nto: " << m_to
            << "\nsubject: " << m_subject
            << "\nbody: " << m_body;
        return stream.str();
    }

private:
    Email() = default; // restrict construction to builder

    string m_from;
    string m_to;
    string m_subject;
    string m_body;
};

class EmailBuilder
{
public:
    EmailBuilder& from(const string &from) {
        m_email.m_from = from;
        return *this;
    }
};
```

```

EmailBuilder& to(const string &to) {
    m_email.m_to = to;
    return *this;
}

EmailBuilder& subject(const string &subject) {
    m_email.m_subject = subject;
    return *this;
}

EmailBuilder& body(const string &body) {
    m_email.m_body = body;
    return *this;
}

operator Email&&() {
    return std::move(m_email); // notice the move
}

private:
    Email m_email;
};

EmailBuilder Email::make()
{
    return EmailBuilder();
}

// Bonus example!
std::ostream& operator <<(std::ostream& stream, const Email& email)
{
    stream << email.to_string();
    return stream;
}

int main()
{
    Email mail = Email::make().from("me@mail.com")
                               .to("you@mail.com")
                               .subject("C++ builders")
                               .body("I like this API, don't you?");

    cout << mail << endl;
}

```

Для более старых версий C++ можно просто игнорировать операцию `std::move` и удалить `&&` из оператора преобразования (хотя это создаст временную копию).

Строитель завершает свою работу, когда он отправляет построенное электронное письмо `operator Email&&()`. В этом примере строитель является временным объектом и возвращает сообщение перед его уничтожением. Вы также можете использовать явную операцию, например `Email EmailBuilder::build() {...}` вместо оператора преобразования.

Пропустите строитель вокруг

Отличная особенность, которую предлагает шаблон Builder, - это возможность **использовать несколько актеров для создания объекта вместе**. Это делается путем передачи строителя другим актерам, которые будут давать дополнительную информацию встроенному объекту. Это особенно удобно, когда вы создаете какой-то запрос, добавляете фильтры и другие спецификации.

```
void add_addresses(EmailBuilder& builder)
{
    builder.from("me@mail.com")
        .to("you@mail.com");
}

void compose_mail(EmailBuilder& builder)
{
    builder.subject("I know the subject")
        .body("And the body. Someone else knows the addresses.");
}

int main()
{
    EmailBuilder builder;
    add_addresses(builder);
    compose_mail(builder);

    Email mail = builder;
    cout << mail << endl;
}
```

Вариант исполнения: Mutable object

Вы можете изменить дизайн этого шаблона в соответствии с вашими потребностями. Я дам один вариант.

В данном примере объект электронной почты является неизменным, т. е. Его свойства не могут быть изменены, поскольку к ним нет доступа. Это была желаемая особенность. Если вам нужно изменить объект после его создания, вы должны предоставить ему некоторые сеттеры. Поскольку эти сеттеры будут дублироваться в строителе, вы можете рассмотреть возможность сделать все это в одном классе (больше не нужен строительный класс). Тем не менее, я бы подумал о необходимости сделать встроенный объект изменчивым в первую очередь.

Прочитайте Реализация шаблона проектирования в C++ онлайн:

<https://riptutorial.com/ru/cplusplus/topic/4335/реализация-шаблона-проектирования-в-cplusplus>

глава 95: Регулярные выражения

Вступление

Регулярные выражения (иногда называемые **регулярными выражениями** или **регулярными выражениями**) представляют собой текстовый синтаксис, который представляет шаблоны, которые могут быть сопоставлены в используемых строках.

Регулярные выражения, введенные в **c++ 11**, могут опционально поддерживать возвращаемый массив совпадающих строк или другой текстовый синтаксис, определяющий, как заменить сопоставленные шаблоны в строках, на которых работает.

Синтаксис

- `regex_match` // Возвращает, была ли вся последовательность символов сопоставлена регулярным выражением, при желании, захват в объект соответствия
- `regex_search` // Возвращает, соответствует ли последовательность символов символу регулярным выражением, необязательно, захватывая объект соответствия
- `regex_replace` // Возвращает последовательность символов ввода, модифицированную регулярным выражением, с помощью строки замещающего формата
- `regex_token_iterator` // Инициализируется символьной последовательностью, определенной итераторами, списком индексов захвата для итерации и регулярным выражением. Dereferencing возвращает текущее индексированное соответствие регулярного выражения. Приращение шагов к следующему индексу захвата или, если он находится в последнем индексе, сбрасывает индекс и задерживает следующее вхождение регулярного выражения в последовательности символов
- `regex_iterator` // Инициализируется символьной последовательностью, определенной итераторами и регулярным выражением. Выделение возвращает часть последовательности символов, совпадающей со всем регулярным выражением. Приращение находит следующее вхождение регулярного выражения в последовательности символов

параметры

Подпись	Описание
<pre>bool regex_match(BidirectionalIterator first, BidirectionalIterator last, smatch& sm, const regex& re, regex_constraints::match_flag_type flags)</pre>	BidirectionalIterator - это любой итератор символов, который обеспечивает приращения и уменьшения операторов <code>smatch</code> может быть <code>cmatch</code> или любым другим другим вариантом <code>match_results</code> который принимает тип

Подпись	Описание
	<code>BidirectionalIterator</code> аргумент <code>smatch</code> может быть пропущен, если результаты регулярного выражения не нужны. Возвращает, соответствует ли <code>re</code> полный символ последовательность, определенная <code>first</code> и <code>last</code>
<pre>bool regex_match(const string& str, smatch& sm, const regex re&, regex_constraints::match_flag_type flags)</pre>	<code>string</code> может быть либо <code>const char*</code> либо <code>string</code> L-Value, функции, принимающие <code>string</code> R-значения, явно удалены. <code>smatch</code> может быть <code>cmatch</code> или любым другим другим вариантом <code>match_results</code> который принимает тип <code>str</code> аргумент <code>smatch</code> может быть опущен, если результаты регулярного выражения не нужны. Возвращает, соответствует ли <code>re</code> всей последовательности символов, определяемой <code>str</code>

Examples

Основные примеры `regex_match` и `regex_search`

```
const auto input = "Some people, when confronted with a problem, think \"I know, I'll use
regular expressions.\"";
smatch sm;

cout << input << endl;

// If input ends in a quotation that contains a word that begins with "reg" and another word
beginning with "ex" then capture the preceeding portion of input
if (regex_match(input, sm, regex("(.*\\\".*\\breg.*\\bex.*\\\"\\s*$)")) {
    const auto capture = sm[1].str();

    cout << '\t' << capture << endl; // Outputs: "\tSome people, when confronted with a
problem, think\n"

    // Search our capture for "a problem" or "# problems"
    if(regex_search(capture, sm, regex("(a|d+)\s+problems?")) {
        const auto count = sm[1] == "a"s ? 1 : stoi(sm[1]);

        cout << '\t' << count << (count > 1 ? " problems\n" : " problem\n"); // Outputs: "\t1
problem\n"
        cout << "Now they have " << count + 1 << " problems.\n"; // Ouputs: "Now they have 2
problems\n"
    }
}
```

[Живой пример](#)

Пример regex_replace

Этот код принимает различные стили скобок и преобразует их в **One True Brace Style** :

```
const auto input = "if (KnR)\n\tfoo();\n\nif (spaces) {\n    foo();\n}\n\nif (allman)\n{\n\tfoo();\n}\n\nif (horstmann)\n{\tfoo();\n}\n\nif (pico)\n{\tfoo(); }\n\nif (whitesmiths)\n\t{\n\t\tfoo();\n\t}\n"s;
```

```
cout << input << regex_replace(input, regex("(.+?)\\s*\\{?\\s*(.+?;)\\s*\\}\\s*", "$1{\n\t$2\n}\n") << endl;
```

Живой пример

Пример regex_token_iterator

`std::regex_token_iterator` предоставляет потрясающий инструмент для **извлечения элементов файла Comma Separated Value** . Помимо преимуществ итерации, этот итератор также способен захватывать скрытые запятые, в которых работают другие методы:

```
const auto input = "please split,this,csv, ,line,\\,\\n"s;
const regex re{ "((?:[^\\"\\\\,]|\\\\\\\\.\\.)+)(?:,|$)" };
const vector<string> m_vecFields{ sregex_token_iterator(cbegin(input), cend(input), re, 1),
sregex_token_iterator() };

cout << input << endl;

copy(cbegin(m_vecFields), cend(m_vecFields), ostream_iterator<string>(cout, "\\n"));
```

Живой пример

Заметная gotcha с итераторами регулярных выражений состоит в том, что аргумент `regex` должен быть L-значением. **Значение R не будет работать**.

Пример regex_iterator

Когда обработка захватов должна выполняться итеративно, `regex_iterator` является хорошим выбором. Выделение `regex_iterator` возвращает `match_result`. Это отлично подходит для условных захватов или захватов, которые имеют взаимозависимость. Предположим, что мы хотим подделать некоторый код на C++. Дано:

```
enum TOKENS {
    NUMBER,
    ADDITION,
    SUBTRACTION,
    MULTIPLICATION,
    DIVISION,
    EQUALITY,
    OPEN_PARENTHESIS,
    CLOSE_PARENTHESIS
};
```

Мы можем tokenize эту строку: `const auto input = "42/2 + -8\t=\n(2 + 2) * 2 * 2 -3"s` `C` `regex_iterator` следующим образом:

```
vector<TOKENS> tokens;
const regex re{ "\\s*(\\(\\?)\\s*(-?\\s*\\d+)\\s*(\\)\\?)\\s*(?:\\(\\+\\)|\\(-\\)|\\(\\*\\)|\\(/\\)|\\(=\\))" };

for_each(sregex_iterator(cbegin(input), cend(input), re), sregex_iterator(), [&](const auto& i) {
    if(i[1].length() > 0) {
        tokens.push_back(OPEN_PARENTHESIS);
    }

    tokens.push_back(i[2].str().front() == '-' ? NEGATIVE_NUMBER : NON_NEGATIVE_NUMBER);

    if(i[3].length() > 0) {
        tokens.push_back(CLOSE_PARENTHESIS);
    }

    auto it = next(cbegin(i), 4);

    for(int result = ADDITION; it != cend(i); ++result, ++it) {
        if (it->length() > 0U) {
            tokens.push_back(static_cast<TOKENS>(result));
            break;
        }
    }
});

match_results<string::const_reverse_iterator> sm;

if(regex_search(crbegin(input), cend(input), sm, regex{ tokens.back() == SUBTRACTION ?
"^\s*\d+\s*-\\s*(-?)" : "^\s*\d+\s*(-?)" })) {
    tokens.push_back(sm[1].length() == 0 ? NON_NEGATIVE_NUMBER : NEGATIVE_NUMBER);
}
```

Живой пример

Заметная gotcha с итераторами регулярных выражений состоит в том, что аргумент `regex` должен быть L-значением, R-значение не будет работать: [Visual Studio regex_iterator Bug?](#)

Разделение строки

```
std::vector<std::string> split(const std::string &str, std::string regex)
{
    std::regex r{ regex };
    std::sregex_token_iterator start{ str.begin(), str.end(), r, -1 }, end;
    return std::vector<std::string>(start, end);
}
```

```
split("Some string\t with whitespace ", "\\s+"); // "Some", "string", "with", "whitespace"
```

Кванторы

Предположим, что нам предоставляется `const string input` в качестве номера телефона для

проверки. Мы могли бы начать с ввода числового ввода с **нулевым или более** `regex_match(input, regex("\\d*")) : regex_match(input, regex("\\d*"))` или **один или несколько квантификаторов** : `regex_match(input, regex("\\d+"))` Но оба из них действительно бывают короткими, если `input` содержит недопустимую числовую строку, например: «123». Давайте использовать **n или более квантификаторов**, чтобы гарантировать, что мы получаем как минимум 7 цифр:

```
regex_match(input, regex("\\d{7,}"))
```

Это гарантирует, что мы получим по крайней мере номер телефона, но `input` также может содержать цифровую строку, которая слишком длинна: «123456789012». Так что давайте перейдем с **квантором между n и m**, так что `input` будет не менее 7 цифр, но не более 11:

```
regex_match(input, regex("\\d{7,11}"));
```

Это приближает нас, но незаконные числовые строки, которые находятся в диапазоне [7, 11], все еще принимаются, например: «123456789». Давайте сделаем код страны необязательным с **ленивым квантиром** :

```
regex_match(input, regex("\\d?\\d{7,10}"))
```

Важно отметить, что **ленивый квантификатор** соответствует *как можно большему количеству символов* , поэтому единственный способ сопоставления этого символа - если у вас уже есть 10 символов, которые были сопоставлены `\\d{7,10}` . (Чтобы совместить первый символ с жадностью, нам пришлось бы делать: `\\d{0,1}` .) **Ленивый квантификатор** может быть добавлен к любому другому квантификатору.

Теперь, как бы мы сделали код области необязательным и принимали только код страны, если код области присутствовал?

```
regex_match(input, regex("(?:\\d{3,4})?\\d{7}"))
```

В этом финальном регулярном выражении `\\d{7}` *требуется* 7 цифр. Этим 7 цифрам необязательно предшествуют либо 3, либо 4 цифры.

Обратите внимание, что мы не добавляли **ленивый квантификатор** : `\\d{3,4}?\\d{7}` , `\\d{3,4}?` соответствовали бы 3 или 4 символам, предпочитая 3. Вместо этого мы собираем не-захват группового совпадения не более одного раза, предпочитая не совпадать. Вызвать несоответствие, если `input` не включал код зоны, например: «1234567».

В заключение темы квантора я хотел бы упомянуть о другом квантовом квантовании, который вы можете использовать, **притяжательный квантификатор** . *Либо ленивый квантор* или **притяжательное квантор** может быть добавлен к любому квантора. Единственная функция **приходящего квантификатора** - помочь движку регулярных

выражений, сообщая ему, жадно воспринимать эти символы и никогда не отказываться от них, даже если это приводит к сбою регулярного выражения. Это, например, не имеет особого смысла: `regex_match(input, regex("\\d{3,4}+\\d{7}"))` Поскольку `input` типа "1234567890" не будет соответствовать как `\\d{3,4}+` всегда будет соответствовать 4 символам, даже если совпадение 3 позволило бы регулярному выражению преуспеть.

Притяжательный квантификатор лучше всего использовать, когда квантованный токен ограничивает количество подходящих символов. Например:

```
regex_match(input, regex("(?:.*\\d{3,4}){3}"))
```

Может использоваться для соответствия, если `input` содержит любое из следующего:

```
123 456 7890
123-456-7890
(123)456-7890
(123) 456 - 7890
```

Но когда это регулярное выражение действительно сияет, когда `input` содержит недопустимый вход:

```
12345 - 67890
```

Без **притяжательного квантификатора** двигатель регулярных выражений должен возвращаться и проверять каждую комбинацию `.*` И 3 или 4 символа, чтобы увидеть, может ли он найти подходящую комбинацию. С помощью **притяжательного квантификатора** регулярное выражение начинается там, где остался 2-й **обладающий квантификатор**, символ «0» и механизм регулярных выражений пытаются настроить `.*` Чтобы позволить `\\d{3,4}` соответствовать; когда он не может просто повторять регулярное выражение, отслеживание обратной связи не выполняется, чтобы увидеть, если раньше `.*` Настройка могла позволить совпадение.

Якоря

C++ обеспечивает только 4 привязки:

- `^` который утверждает начало строки
- `$` который утверждает конец строки
- `\\b` который утверждает символ `\\w` или начало или конец строки
- `\\B` который утверждает символ `\\w`

Скажем, например, мы хотим захватить число с его знаком:

```
auto input = "+1--12*123/+1234"s;
smatch sm;

if(regex_search(input, sm, regex{ "(?:^|\\b\\W) ([+-]?\\d+)" })) {
```

```
do {  
    cout << sm[1] << endl;  
    input = sm.suffix().str();  
} while(regex_search(input, sm, regex{ "(?:^\\W|\\b\\W) ([+-]?\\d+)" }));  
}
```

Живой пример

Важно отметить, что якорь не потребляет никаких символов.

Прочитайте Регулярные выражения онлайн: <https://riptutorial.com/ru/cplusplus/topic/1681/регулярные-выражения>

глава 96: Рекомендации

Examples

Определение ссылки

Ссылки ведут себя аналогично, но не совсем как константные указатели. Ссылка определяется путем суффикса амперсанда & имени типа.

```
int i = 10;
int &refi = i;
```

Здесь `refi` является ссылкой, связанной с `i`.

Ссылки абстрагируют семантику указателей, действуя как псевдоним основного объекта:

```
refi = 20; // i = 20;
```

Вы также можете определить несколько ссылок в одном определении:

```
int i = 10, j = 20;
int &refi = i, &refj = j;

// Common pitfall :
// int& refi = i, k = j;
// refi will be of type int&.
// though, k will be of type int, not int&!
```

Ссылки **должны** быть правильно инициализированы во время определения и впоследствии не могут быть изменены. Следующий фрагмент кода вызывает ошибку компиляции:

```
int &i; // error: declaration of reference variable 'i' requires an initializer
```

Вы также не можете напрямую привязать ссылку к `nullptr`, в отличие от указателей:

```
int *const ptri = nullptr;
int &refi = nullptr; // error: non-const lvalue reference to type 'int' cannot bind to a
temporary of type 'nullptr_t'
```

Ссылки на C++ - это псевдоним существующих переменных

Ссылка в C++ - это просто `Alias` или другое имя переменной. Так же, как и большинство из нас может быть передано с использованием нашего имени паспорта и псевдонима.

Ссылки не существуют буквально, и они не занимают никакой памяти. Если мы напечатаем адрес ссылочной переменной, он напечатает тот же адрес, что и адрес переменной, на

которую ссылается.

```
int main() {
    int i = 10;
    int &j = i;

    cout<<&i<<endl;
    cout<<&b<<endl;
    return 0;
}
```

В приведенном выше примере оба `cout` будут печатать один и тот же адрес. Ситуация будет такой же, если мы возьмем переменную в качестве ссылки в функции

```
void func (int &fParam ) {
    cout<<"Address inside function => "<<&fParam<<endl;
}

int main() {
    int i = 10;
    cout<<"Address inside Main => "<<&i<<endl;

    func(i);

    return 0;
}
```

В этом примере оба `cout` будут печатать тот же адрес.

Поскольку мы знаем, что C++ References - это просто псевдоним, и для создания псевдонима нам нужно иметь то, к чему может относиться псевдоним.

Это точная причина, по которой такой оператор выдает ошибку компилятора

```
int &i;
```

Потому что псевдоним не имеет никакого отношения ни к чему.

Прочитайте Рекомендации онлайн: <https://riptutorial.com/ru/cplusplus/topic/1548/рекомендации>

глава 97: Рекурсивный Мьютекс

Examples

станд :: recursive_mutex

Рекурсивный мьютекс позволяет тому же потоку рекурсивно блокировать ресурс - до неопределенного предела.

Для этого очень мало реальных оправданий. Некоторым сложным реализациям, возможно, потребуется вызвать перегруженную копию функции без освобождения блокировки.

```
std::atomic_int temp{0};
std::recursive_mutex _mutex;

//launch_deferred launches asynchronous tasks on the same thread id

auto future1 = std::async(
    std::launch::deferred,
    [&]()
    {
        std::cout << std::this_thread::get_id() << std::endl;

        std::this_thread::sleep_for(std::chrono::seconds(3));
        std::unique_lock<std::recursive_mutex> lock( _mutex);
        temp=0;

    });

auto future2 = std::async(
    std::launch::deferred,
    [&]()
    {
        std::cout << std::this_thread::get_id() << std::endl;
        while ( true )
        {
            std::this_thread::sleep_for(std::chrono::milliseconds(1));
            std::unique_lock<std::recursive_mutex> lock( _mutex,
std::try_to_lock);

            if ( temp < INT_MAX )
                temp++;

            cout << temp << endl;

        }
    });
future1.get();
future2.get();
```

Прочитайте Рекурсивный Мьютекс онлайн: <https://riptutorial.com/ru/cplusplus/topic/9929/рекурсивный-мьютекс>

глава 98: Рекурсия в C ++

Examples

Используя рекурсию хвоста и рекурсию в стиле Фибоначи, чтобы решить последовательность Фибоначи

Простым и очевидным способом использования рекурсии для получения N-го члена последовательности Фибоначи является

```
int get_term_fib(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return get_term_fib(n - 1) + get_term_fib(n - 2);
}
```

Однако этот алгоритм не масштабируется для более высоких терминов: для большего и большего числа n количество вызовов функций, которые вам нужно сделать, растет экспоненциально. Это можно заменить простой хвостовой рекурсией.

```
int get_term_fib(int n, int prev = 0, int curr = 1)
{
    if (n == 0)
        return prev;
    if (n == 1)
        return curr;
    return get_term_fib(n - 1, curr, prev + curr);
}
```

Каждый вызов функции теперь сразу вычисляет следующий член в последовательности Фибоначи, поэтому число вызовов функций линейно масштабируется с n .

Рекурсия с воспоминаниями

Рекурсивные функции могут стать довольно дорогими. Если они являются чистыми функциями (функции, которые всегда возвращают одно и то же значение при вызове с теми же аргументами и не зависят ни от внешнего состояния, ни изменяют его), они могут быть сделаны значительно быстрее за счет памяти, сохраняя уже рассчитанные значения.

Ниже приведена реализация последовательности Фибоначчи с memoization:

```
#include <map>

int fibonacci(int n)
```

```

{
    static std::map<int, int> values;
    if (n==0 || n==1)
        return n;
    std::map<int,int>::iterator iter = values.find(n);
    if (iter == values.end())
    {
        return values[n] = fibonacci(n-1) + fibonacci(n-2);
    }
    else
    {
        return iter->second;
    }
}

```

Обратите внимание, что, несмотря на использование простой формулы рекурсии, при первом вызове эта функция равна $O(n)$. При последующих вызовах с одинаковым значением, конечно, $O(1)$.

Обратите внимание, однако, что эта реализация не является повторной. Кроме того, он не позволяет избавиться от сохраненных значений. Альтернативной реализацией было бы разрешить передачу карты в качестве дополнительного аргумента:

```

#include <map>

int fibonacci(int n, std::map<int, int> values)
{
    if (n==0 || n==1)
        return n;
    std::map<int,int>::iterator iter = values.find(n);
    if (iter == values.end())
    {
        return values[n] = fibonacci(n-1) + fibonacci(n-2);
    }
    else
    {
        return iter->second;
    }
}

```

Для этой версии вызывающий абонент должен поддерживать карту с сохраненными значениями. Это имеет то преимущество, что функция теперь является реентерабельной, и что вызывающий может удалить значения, которые больше не нужны, сохраняя память. Недостаток заключается в том, что он разрушает инкапсуляцию; вызывающий может изменить вывод, заполнив карту неверными значениями.

Прочитайте Рекурсия в C ++ онлайн: <https://riptutorial.com/ru/cplusplus/topic/5693/рекурсия-в-c-plusplus>

глава 99: Сгибание выражений

замечания

Fold Expressions поддерживаются для следующих операторов:

+	-	*	/	%	\	&		<<	>>		
+	знак равно	знак равно	знак равно	знак равно	\знак равно	знак равно		<<	>>	знак равно	
=							=	=	=		
==	знак равно	<	>	<=	>=	&&		,	.*	-> *	

При складывании над пустой последовательностью выражение сложения является плохо сформированным, за исключением следующих трех операторов:

оператор	Значение, когда пакет параметров пуст
&&	правда
	ложный
,	недействительными ()

Examples

Унарные складки

Унарные складки используются для *сложения пакетов параметров* над конкретным оператором. Существует 2 вида унарных складок:

- Unary **Left Fold** (... op pack) который расширяется следующим образом:

```
((Pack1 op Pack2) op ...) op PackN
```

- Unary **Right Fold** (pack op ...) который расширяется следующим образом:

```
Pack1 op (... (Pack(N-1) op PackN))
```

Вот пример

```
template<typename... Ts>
int sum(Ts... args)
{
    return (... + args); //Unary left fold
    //return (args + ...); //Unary right fold

    // The two are equivalent if the operator is associative.
    // For +, ((1+2)+3) (left fold) == (1+(2+3)) (right fold)
    // For -, ((1-2)-3) (left fold) != (1-(2-3)) (right fold)
}

int result = sum(1, 2, 3); // 6
```

Бинарные складки

Двоичные складки в основном [унарные складки](#) , с дополнительным аргументом.

Существует два вида двоичных сгибов:

- **Binary Left Fold** - (value op ... op pack) - расширяется следующим образом:

```
((Value op Pack1) op Pack2) op ... op PackN
```

- **Binary Right Fold** (pack op ... op value) - расширяется следующим образом:

```
Pack1 op (... op (Pack(N-1) op (PackN op Value)))
```

Вот пример:

```
template<typename... Ts>
int removeFrom(int num, Ts... args)
{
    return (num - ... - args); //Binary left fold
    // Note that a binary right fold cannot be used
    // due to the lack of associativity of operator-
}

int result = removeFrom(1000, 5, 10, 15); //'result' is 1000 - 5 - 10 - 15 = 970
```

Складывание запятой

Это общая операция, требующая выполнения определенной функции по каждому элементу в пакете параметров. С C ++ 11 лучше, что мы можем сделать, это:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

Но с выражением складки выражение выше упрощает:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    (void(os << args), ...);
}
```

Не требуется критический шаблон.

Прочитайте Сгибание выражений онлайн: <https://riptutorial.com/ru/cplusplus/topic/2676/сгибание-выражений>

глава 100: семафор

Вступление

Семафоры недоступны в C++ на данный момент, но могут быть легко реализованы с помощью мьютекса и переменной условия.

Этот пример был взят из:

[C++ 0x не имеет семафоров? Как синхронизировать потоки?](#)

Examples

Семафор C++ 11

```
#include <mutex>
#include <condition_variable>

class Semaphore {
public:
    Semaphore (int count_ = 0)
    : count(count_)
    {
    }

    inline void notify( int tid ) {
        std::unique_lock<std::mutex> lock(mtx);
        count++;
        cout << "thread " << tid << " notify" << endl;
        //notify the waiting thread
        cv.notify_one();
    }

    inline void wait( int tid ) {
        std::unique_lock<std::mutex> lock(mtx);
        while(count == 0) {
            cout << "thread " << tid << " wait" << endl;
            //wait on the mutex until notify is called
            cv.wait(lock);
            cout << "thread " << tid << " run" << endl;
        }
        count--;
    }

private:
    std::mutex mtx;
    std::condition_variable cv;
    int count;
};
```

Семафор в действии

Следующая функция добавляет четыре потока. Три потока конкурируют за семафор,

который установлен на счет одного. Более медленный поток вызывает `notify_one()`, позволяя продолжить один из ожидающих потоков.

В результате `s1` сразу начинает вращаться, в результате чего `count` использования `count` остается ниже 1. Другие потоки ожидают очереди переменных условия, пока не будет вызвано уведомление `()`.

```
int main()
{
    Semaphore sem(1);

    thread s1([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::seconds(5));
            sem.wait( 1 );
        }
    });
    thread s2([&]() {
        while(true){
            sem.wait( 2 );
        }
    });
    thread s3([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::milliseconds(600));
            sem.wait( 3 );
        }
    });
    thread s4([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::seconds(5));
            sem.notify( 4 );
        }
    });

    s1.join();
    s2.join();
    s3.join();
    s4.join();

    ...
}
```

Прочитайте семафор онлайн: <https://riptutorial.com/ru/cplusplus/topic/9785/семафор>

глава 101: Системы сборки

Вступление

C ++, как и C, имеет долгую и разнообразную историю, связанную с процессами компиляции и процессами сборки. Сегодня C ++ имеет различные популярные системы сборки, которые используются для компиляции программ, иногда для нескольких платформ в рамках одной системы сборки. Здесь будут рассмотрены и проанализированы несколько систем сборки.

замечания

В настоящее время нет универсальной или доминирующей системы сборки для C ++, которая является популярной и кросс-платформенной. Тем не менее, существует несколько крупных систем построения, которые привязаны к основным платформам / проектам, наиболее заметным из которых является GNU Make с операционной системой GNU / Linux и NMAKE с системой Visual C ++ / Visual Studio.

Кроме того, некоторые интегрированные среды разработки (IDE) также включают специализированные системы сборки, которые будут использоваться специально с родной средой IDE. Некоторые генераторы системы сборки могут генерировать эти собственные форматы IDE / форматы проекта, такие как CMake для Eclipse и Microsoft Visual Studio 2012.

Examples

Создание среды сборки с помощью CMake

CMake создает среду сборки практически для любого компилятора или IDE из одного определения проекта. В следующих примерах показано, как добавить файл CMake в [кросс-платформенный код «Hello World» C ++](#) .

Файлы CMake всегда называются «CMakeLists.txt» и должны уже существовать в корневом каталоге каждого проекта (и, возможно, в подкаталогах тоже). Основной файл CMakeLists.txt выглядит так:

```
cmake_minimum_required(VERSION 2.4)

project (HelloWorld)

add_executable (HelloWorld main.cpp)
```

Смотрите на [живую в Колиру](#) .

Этот файл сообщает CMake имя проекта, какую версию файла ожидать и инструкции для создания исполняемого файла под названием «HelloWorld», для которого требуется `main.cpp`.

Создайте среду сборки для вашего установленного компилятора / IDE из командной строки:

```
> cmake .
```

Создайте приложение с помощью:

```
> cmake --build .
```

Это создает среду сборки по умолчанию для системы, в зависимости от ОС и установленных инструментов. Держите исходный код в чистоте от любых артефактов сборки с использованием «внекорпоративных» сборников:

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

CMake также может абстрагировать основные команды платформы оболочки из предыдущего примера:

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

CMake включает в себя [генераторы](#) для ряда общих инструментов сборки и IDE. Чтобы создать `make` - `nmake` для `nmake Visual Studio` :

```
> cmake -G "NMake Makefiles" ..
> nmake
```

Компиляция с GNU make

Вступление

GNU Make (стиль `make`) - это программа, предназначенная для автоматизации выполнения команд оболочки. GNU Make - это одна конкретная программа, которая подпадает под семейство Make. Они остаются популярными среди Unix-подобных и POSIX-подобных операционных систем, в том числе производных от ядра Linux, Mac OS X и BSD.

GNU Make особенно примечателен тем, что он подключен к проекту GNU, который

подключен к популярной операционной системе GNU / Linux. GNU Make также имеет совместимые версии, работающие на разных вариантах Windows и Mac OS X. Это также очень стабильная версия с историческим значением, которое остается популярным. Именно по этим причинам GNU Make часто учат вместе с C и C ++.

Основные правила

Чтобы скомпилировать с make, создайте Makefile в каталоге проекта. Ваш Makefile может быть таким же простым, как:

Makefile

```
# Set some variables to use in our command
# First, we set the compiler to be g++
CXX=g++

# Then, we say that we want to compile with g++'s recommended warnings and some extra ones.
CXXFLAGS=-Wall -Wextra -pedantic

# This will be the output file
EXE=app

SRCS=main.cpp

# When you call `make` at the command line, this "target" is called.
# The $(EXE) at the right says that the `all` target depends on the `$(EXE)` target.
# $(EXE) expands to be the content of the EXE variable
# Note: Because this is the first target, it becomes the default target if `make` is called
without target
all: $(EXE)

# This is equivalent to saying
# app: $(SRCS)
# $(SRCS) can be separated, which means that this target would depend on each file.
# Note that this target has a "method body": the part indented by a tab (not four spaces).
# When we build this target, make will execute the command, which is:
# g++ -Wall -Wextra -pedantic -o app main.cpp
# I.E. Compile main.cpp with warnings, and output to the file ./app
$(EXE): $(SRCS)
    @$(CXX) $(CXXFLAGS) -o $@ $(SRCS)

# This target should reverse the `all` target. If you call
# make with an argument, like `make clean`, the corresponding target
# gets called.
clean:
    @rm -f $(EXE)
```

ПРИМЕЧАНИЕ. Убедитесь, что отступы имеют вкладку, а не четыре пробела.

В противном случае вы получите сообщение об ошибке Makefile:10: ***
missing separator. Stop.

Чтобы запустить это из командной строки, выполните следующие действия:

```
$ cd ~/Path/to/project
$ make
$ ls
app  main.cpp  Makefile

$ ./app
Hello World!

$ make clean
$ ls
main.cpp  Makefile
```

Инкрементальные сборки

Когда вы начинаете иметь больше файлов, make становится более полезным. Что делать, если вы отредактировали **файл a.cpp**, но не **b.cpp** ? Перекомпиляция **b.cpp** займет больше времени.

Со следующей структурой каталогов:

```
.
+-- src
|   +-- a.cpp
|   +-- a.hpp
|   +-- b.cpp
|   +-- b.hpp
+-- Makefile
```

Это будет хороший Makefile:

Makefile

```
CXX=g++
CXXFLAGS=-Wall -Wextra -pedantic
EXE=app

SRCS_GLOB=src/*.cpp
SRCS=$(wildcard $(SRCS_GLOB))
OBJS=$(SRCS:.cpp=.o)

all: $(EXE)

$(EXE): $(OBJS)
    @$(CXX) -o $@ $(OBJS)

depend: .depend

.depend: $(SRCS)
    @-rm -f ./depend
    @$(CXX) $(CXXFLAGS) -MM $^>>./depend

clean:
    -rm -f $(EXE)
    -rm $(OBJS)
```

```
-rm *~
-rm .depend

include .depend
```

Снова посмотрите вкладки. Этот новый Makefile гарантирует, что вы только перекомпилируете измененные файлы, минимизируя время компиляции.

Документация

Для получения дополнительной информации см. [Официальную документацию Free Software Foundation](#) , [документацию по стеку](#) и [подробный ответ dmckee на stackoverflow](#) .

Строительство с помощью SCons

Вы можете построить [кросс-платформенный код «Hello World» C ++](#) , используя инструмент разработки программного обеспечения [Scons](#) - A [Python](#)- language.

Сначала создайте файл `SConstruct` (обратите внимание, что SCons будет искать файл с этим точным именем по умолчанию). На данный момент файл должен находиться в каталоге прямо по вашему `hello.cpp` . Напишите в новом файле строку

```
Program('hello.cpp')
```

Теперь, с терминала, запустите `scons` . Вы должны увидеть что-то вроде

```
$ scon
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
g++ -o hello.o -c hello.cpp
g++ -o hello hello.o
scons: done building targets.
```

(хотя информация будет зависеть от вашей операционной системы и установленного компилятора).

Классы `Environment` и `Glob` помогут вам настроить конфигурацию. Например, файл `SConstruct`

```
env=Environment(CPPPATH='/usr/include/boost/',
    CPPDEFINES=[],
    LIBS=[],
    SCONS_CXX_STANDARD="c++11"
)

env.Program('hello', Glob('src/*.cpp'))
```

создает исполняемый `hello` , используя все файлы `cpp` в `src` . Его `CPPPATH` - это

`/usr/include/boost` и он указывает стандарт C ++ 11.

Ниндзя

Вступление

Система сборки ниндзя описывается на веб-сайте проекта как «[небольшая система сборки с акцентом на скорость](#)». Ninja предназначен для создания файлов, созданных генераторами системных файлов, и использует низкоуровневый подход для создания систем, в отличие от более высокоуровневых системных менеджеров, таких как CMake или Meson.

Ninja в основном написан на C ++ и Python и был создан как альтернатива системе сборки SCons для проекта Chromium.

NMAKE (Утилита технического обслуживания Microsoft)

Вступление

NMAKE - это утилита командной строки, разработанная Microsoft для использования в основном в сочетании с инструментами командной строки Microsoft Visual Studio и / или Visual C ++.

NMAKE - это система сборки, которая подпадает под семейство систем сборки, но имеет определенные отличительные особенности, которые отличаются от Unix-подобных программ Make, таких как поддержка синтаксиса пути к файлу Windows (который сам по себе отличается от путей файлов в стиле Unix).

Autotools (GNU)

Вступление

Autotools - это группа программ, которые создают GNU Build System для данного программного пакета. Это набор инструментов, которые работают вместе для создания различных ресурсов сборки, таких как Makefile (для использования с GNU Make). Таким образом, Autotools можно считать генератором системы де-факто.

Некоторые известные программы Autotools включают в себя:

- Autoconf
- Automake (не путать с `make`)

В общем, Autotools предназначен для создания Unix-совместимого скрипта и Makefile, чтобы позволить следующей команде создавать (а также устанавливать) большинство пакетов (в простом случае):

```
./configure && make && make install
```

Таким образом, Autotools также имеет отношение к определенным менеджерам пакетов, особенно к тем, которые подключены к операционным системам, которые соответствуют стандарту POSIX.

Прочитайте Системы сборки онлайн: <https://riptutorial.com/ru/cplusplus/topic/8200/системы-сборки>

глава 102: Сортировка

замечания

Семейство функций `std::sort` находится в библиотеке `algorithm`.

Examples

Сортировка контейнеров последовательностей с заданным заказом

Если значения в контейнере имеют некоторые уже перегруженные операторы, `std::sort` может использоваться со специализированными функторами для сортировки в порядке возрастания или убывания:

C ++ 11

```
#include <vector>
#include <algorithm>
#include <functional>

std::vector<int> v = {5,1,2,4,3};

//sort in ascending order (1,2,3,4,5)
std::sort(v.begin(), v.end(), std::less<int>());

// Or just:
std::sort(v.begin(), v.end());

//sort in descending order (5,4,3,2,1)
std::sort(v.begin(), v.end(), std::greater<int>());

//Or just:
std::sort(v.rbegin(), v.rend());
```

C ++ 14

В C ++ 14 нам не нужно указывать аргумент шаблона для объектов функции сравнения, а вместо этого вывести объект на основе того, что он передает:

```
std::sort(v.begin(), v.end(), std::less<>()); // ascending order
std::sort(v.begin(), v.end(), std::greater<>()); // descending order
```

Сортировка контейнеров последовательностей путем перегрузки меньшего оператора

Если функция заказа не передана, `std::sort` будет упорядочивать элементы, вызывая `operator<` на пары элементов, который должен возвращать тип, контекстно конвертируемый в `bool` (или просто `bool`). Основные типы (целые числа, поплавки, указатели и т. Д.) Уже

построены в операторах сравнения.

Мы можем перегрузить этот оператор, чтобы заставить `sort` по умолчанию работать с определенными пользователем типами.

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    // Use variable to provide total order operator less
    // `this` always represents the left-hand side of the compare.
    bool operator<(const Base &b) const {
        return this->variable < b.variable;
    }

    int variable;
};

int main() {
    std::vector<Base> vector;
    std::deque<Base> deque;
    std::list<Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using operator<(const Base &b) function
    std::sort(vector.begin(), vector.end());
    std::sort(deque.begin(), deque.end());
    // List must be sorted differently due to its design
    list.sort();

    return 0;
}
```

Сортировка контейнеров последовательностей с использованием

функции сравнения

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};

bool compare(const Base &a, const Base &b) {
    return a.variable < b.variable;
}

int main() {
    std::vector<Base> vector;
    std::deque<Base> deque;
    std::list<Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using comparing function
    std::sort(vector.begin(), vector.end(), compare);
    std::sort(deque.begin(), deque.end(), compare);
    list.sort(compare);

    return 0;
}
```

Сортировка контейнеров последовательностей с использованием лямбда-выражений (C ++ 11)

C ++ 11

```
// Include sequence containers
```

```

#include <vector>
#include <deque>
#include <list>
#include <array>
#include <forward_list>

// Include sorting algorithm
#include <algorithm>

class Base {
public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};

int main() {
    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // We're using C++11, so let's use initializer lists to insert items.
    std::vector<Base> vector = {a, b};
    std::deque<Base> deque = {a, b};
    std::list<Base> list = {a, b};
    std::array<Base, 2> array = {a, b};
    std::forward_list<Base> flist = {a, b};

    // We can sort data using an inline lambda expression
    std::sort(std::begin(vector), std::end(vector),
        [](const Base &a, const Base &b) { return a.variable < b.variable;});

    // We can also pass a lambda object as the comparator
    // and reuse the lambda multiple times
    auto compare = [](const Base &a, const Base &b) {
        return a.variable < b.variable;};
    std::sort(std::begin(deque), std::end(deque), compare);
    std::sort(std::begin(array), std::end(array), compare);
    list.sort(compare);
    flist.sort(compare);

    return 0;
}

```

Сортировка и последовательность контейнеров

`std::sort`, найденный в стандартном `algorithm` заголовка библиотеки, является стандартным библиотечным алгоритмом для сортировки диапазона значений, определяемых парой итераторов. `std::sort` принимает в качестве последнего параметра функтор, используемый для сравнения двух значений; так определяется порядок. Обратите внимание, что `std::sort` **нестабилен**.

Функция сравнения *должна* налагать **строгие, слабые порядки** на элементы. Достаточно

простого сравнения (или большего).

Контейнер с итераторами с произвольным доступом можно отсортировать с помощью алгоритма `std::sort` :

C ++ 11

```
#include <vector>
#include <algorithm>

std::vector<int> MyVector = {3, 1, 2}

//Default comparison of <
std::sort(MyVector.begin(), MyVector.end());
```

`std::sort` требует, чтобы его итераторы были итераторами произвольного доступа.

Контейнеры последовательности `std::list` и `std::forward_list` (требующие C ++ 11) не предоставляют итераторы произвольного доступа, поэтому их нельзя использовать с `std::sort` . Однако у них есть функции-члены `sort` которые реализуют алгоритм сортировки, который работает с их собственными типами итераторов.

C ++ 11

```
#include <list>
#include <algorithm>

std::list<int> MyList = {3, 1, 2}

//Default comparison of <
//Whole list only.
MyList.sort();
```

Их члены `sort` функции всегда сортировать весь список, так что они не могут сортировать поддиапазон элементов. Однако, поскольку `list` и `forward_list` имеют быстрые операции сращивания, вы можете извлечь элементы, которые будут отсортированы из списка, отсортировать их, а затем наполнить их там, где они были достаточно эффективны:

```
void sort_sublist(std::list<int>& mylist, std::list<int>::const_iterator start,
std::list<int>::const_iterator end) {
    //extract and sort half-open sub range denoted by start and end iterator
    std::list<int> tmp;
    tmp.splice(tmp.begin(), list, start, end);
    tmp.sort();
    //re-insert range at the point we extracted it from
    list.splice(end, tmp);
}
```

сортировка по std :: map (по возрастанию и убыванию)

В этом примере сортируются элементы в порядке **возрастания ключа** с помощью карты. Вы можете использовать любой тип, включая класс, вместо `std::string` , в приведенном

ниже примере.

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::map<double, std::string> sorted_map;
    // Sort the names of the planets according to their size
    sorted_map.insert(std::make_pair(0.3829, "Mercury"));
    sorted_map.insert(std::make_pair(0.9499, "Venus"));
    sorted_map.insert(std::make_pair(1, "Earth"));
    sorted_map.insert(std::make_pair(0.532, "Mars"));
    sorted_map.insert(std::make_pair(10.97, "Jupiter"));
    sorted_map.insert(std::make_pair(9.14, "Saturn"));
    sorted_map.insert(std::make_pair(3.981, "Uranus"));
    sorted_map.insert(std::make_pair(3.865, "Neptune"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (" << entry.first << " of Earth's radius)" << '\n';
    }
}
```

Выход:

```
Mercury (0.3829 of Earth's radius)
Mars (0.532 of Earth's radius)
Venus (0.9499 of Earth's radius)
Earth (1 of Earth's radius)
Neptune (3.865 of Earth's radius)
Uranus (3.981 of Earth's radius)
Saturn (9.14 of Earth's radius)
Jupiter (10.97 of Earth's radius)
```

Если записи с равными ключами возможны, используйте `multimap` вместо `map` (как в следующем примере).

Чтобы отсортировать элементы по **убыванию**, объявите карту с правильным функтором сравнения (`std::greater<>`):

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::multimap<int, std::string, std::greater<int>> sorted_map;
    // Sort the names of animals in descending order of the number of legs
    sorted_map.insert(std::make_pair(6, "bug"));
    sorted_map.insert(std::make_pair(4, "cat"));
    sorted_map.insert(std::make_pair(100, "centipede"));
    sorted_map.insert(std::make_pair(2, "chicken"));
    sorted_map.insert(std::make_pair(0, "fish"));
    sorted_map.insert(std::make_pair(4, "horse"));
}
```

```
sorted_map.insert(std::make_pair(8, "spider"));

for (auto const& entry: sorted_map)
{
    std::cout << entry.second << " (has " << entry.first << " legs)" << '\n';
}
}
```

Выход

```
centipede (has 100 legs)
spider (has 8 legs)
bug (has 6 legs)
cat (has 4 legs)
horse (has 4 legs)
chicken (has 2 legs)
fish (has 0 legs)
```

Сортировка встроенных массивов

Алгоритм `sort` сортирует последовательность, определенную двумя итераторами. Этого достаточно, чтобы отсортировать встроенный массив (также известный как c-style).

C ++ 11

```
int arr1[] = {36, 24, 42, 60, 59};

// sort numbers in ascending order
sort(std::begin(arr1), std::end(arr1));

// sort numbers in descending order
sort(std::begin(arr1), std::end(arr1), std::greater<int>());
```

До C ++ 11 конец массива должен был быть «вычислен» с использованием размера массива:

C ++ 11

```
// Use a hard-coded number for array size
sort(arr1, arr1 + 5);

// Alternatively, use an expression
const size_t arr1_size = sizeof(arr1) / sizeof(*arr1);
sort(arr1, arr1 + arr1_size);
```

Прочитайте Сортировка онлайн: <https://riptutorial.com/ru/cplusplus/topic/1675/сортировка>

глава 103: Союзы

замечания

Союзы - очень полезные инструменты, но приходят с несколькими важными оговорками:

- Неопределенное поведение по стандарту C ++ для доступа к элементу объединения, который не был последним измененным членом. Хотя многие компиляторы C ++ разрешают этот доступ определенными способами, это расширения и не могут быть гарантированы в разных компиляторах.

`std::variant` (поскольку C ++ 17) подобен объединению, только он сообщает вам, что он в настоящее время содержит (часть его видимого состояния - это тип значения, которое он имеет в данный момент: он обеспечивает доступ к значениям, доступным только к этому типу).

- Реализации не обязательно выравнивают членов разных размеров по одному и тому же адресу.

Examples

Основные возможности соединения

Союзы - это специализированная структура, внутри которой все члены занимают перекрывающуюся память.

```
union U {
    int a;
    short b;
    float c;
};
U u;

//Address of a and b will be equal
(void*)&u.a == (void*)&u.b;
(void*)&u.a == (void*)&u.c;

//Assigning to any union member changes the shared memory of all members
u.c = 4.f;
u.a = 5;
u.c != 4.f;
```

Типичное использование

Союзы полезны для минимизации использования памяти для исключительных данных, например, при реализации смешанных типов данных.

```

struct AnyType {
    enum {
        IS_INT,
        IS_FLOAT
    } type;

    union Data {
        int as_int;
        float as_float;
    } value;

    AnyType(int i) : type(IS_INT) { value.as_int = i; }
    AnyType(float f) : type(IS_FLOAT) { value.as_float = f; }

    int get_int() const {
        if(type == IS_INT)
            return value.as_int;
        else
            return (int)value.as_float;
    }

    float get_float() const {
        if(type == IS_FLOAT)
            return value.as_float;
        else
            return (float)value.as_int;
    }
};

```

Неопределенное поведение

```

union U {
    int a;
    short b;
    float c;
};
U u;

u.a = 10;
if (u.b == 10) {
    // this is undefined behavior since 'a' was the last member to be
    // written to. A lot of compilers will allow this and might issue a
    // warning, but the result will be "as expected"; this is a compiler
    // extension and cannot be guaranteed across compilers (i.e. this is
    // not compliant/portable code).
}

```

Прочитайте Союзы онлайн: <https://riptutorial.com/ru/cplusplus/topic/2678/союзы>

глава 104: Специальные функции участников

Examples

Виртуальные и защищенные деструкторы

Класс, предназначенный для наследования, называется базовым классом. Следует соблюдать особые функции члена такого класса.

Класс, предназначенный для полиморфного использования во время выполнения (посредством указателя на базовый класс), должен объявить деструктор `virtual`. Это позволяет соответствующим образом уничтожить производные части объекта, даже если объект уничтожен с помощью указателя на базовый класс.

```
class Base {
public:
    virtual ~Base() = default;

private:
    // data members etc.
};

class Derived : public Base { // models Is-A relationship
public:
    // some methods

private:
    // more data members
};

// virtual destructor in Base ensures that derived destructors
// are also called when the object is destroyed
std::unique_ptr<Base> base = std::make_unique<Derived>();
base = nullptr; // safe, doesn't leak Derived's members
```

Если класс не должен быть полиморфным, но все же необходимо разрешить его наследование, используйте не виртуальный `protected` деструктор.

```
class NonPolymorphicBase {
public:
    // some methods

protected:
    ~NonPolymorphicBase() = default; // note: non-virtual

private:
    // etc.
};
```

Такой класс никогда не может быть уничтожен с помощью указателя, избегая молчащих утечек из-за нарезки.

Этот метод особенно относится к классам, предназначенным для `private` базовых классов. Такой класс может быть использован для инкапсуляции некоторых общих деталей реализации, а также предоставления `virtual` методов в качестве точек настройки. Этот класс не должен использоваться полиморфно, а `protected` деструктор помогает документировать это требование непосредственно в коде.

Наконец, некоторые классы могут потребовать, чтобы они *никогда не* использовались в качестве базового класса. В этом случае класс может быть отмечен как `final`. В этом случае нормальный не виртуальный публичный деструктор.

```
class FinalClass final { //    marked final here
public:
    ~FinalClass() = default;

private:
    //    etc.
};
```

Неявное перемещение и копирование

Имейте в виду, что объявление деструктора запрещает компилятору создавать неявные конструкторы перемещения и переводить операторы присваивания. Если вы объявите деструктор, не забудьте также добавить соответствующие определения для операций перемещения.

Кроме того, объявление операций перемещения будет препятствовать генерации операций копирования, поэтому их также следует добавить (если для объектов этого класса требуется семантика копирования).

```
class Movable {
public:
    virtual ~Movable() noexcept = default;

    //    compiler won't generate these unless we tell it to
    //    because we declared a destructor
    Movable(Movable&&) noexcept = default;
    Movable& operator=(Movable&&) noexcept = default;

    //    declaring move operations will suppress generation
    //    of copy operations unless we explicitly re-enable them
    Movable(const Movable&) = default;
    Movable& operator=(const Movable&) = default;
};
```

Копирование и свопинг

Если вы пишете класс, который управляет ресурсами, вам необходимо реализовать все

специальные функции-члены (см. [Правило 3/5 / Zero](#)). Наиболее прямым подходом к написанию конструктора копирования и оператора присваивания будет:

```
person(const person &other)
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
{
    std::strcpy(name, other.name);
}

person& operator=(person const& rhs) {
    if (this != &other) {
        delete [] name;
        name = new char[std::strlen(other.name) + 1];
        std::strcpy(name, other.name);
        age = other.age;
    }

    return *this;
}
```

Но этот подход имеет некоторые проблемы. Это не дает надежной гарантии исключения - если `new[]` бросает, мы уже очистили ресурсы, принадлежащие `this` и не можем восстановить. Мы дублируем много логики построения копии при копировании. И мы должны помнить проверку самонаведения, которая обычно просто добавляет накладные расходы на операцию копирования, но по-прежнему имеет решающее значение.

Чтобы удовлетворить сильную гарантию исключения и избежать дублирования кода (в два раза с последующим оператором присваивания оператора), мы можем использовать идиому копирования и свопинга:

```
class person {
    char* name;
    int age;
public:
    /* all the other functions ... */

    friend void swap(person& lhs, person& rhs) {
        using std::swap; // enable ADL

        swap(lhs.name, rhs.name);
        swap(lhs.age, rhs.age);
    }

    person& operator=(person rhs) {
        swap(*this, rhs);
        return *this;
    }
};
```

Почему это работает? Подумайте, что произойдет, когда у нас есть

```
person p1 = ...;
person p2 = ...;
```

```
p1 = p2;
```

Во-первых, мы копируем конструкцию `rhs` из `p2` (которую мы не должны дублировать здесь). Если эта операция срабатывает, мы ничего не делаем в `operator=` и `p1` остается нетронутым. Затем мы заменяем элементы между `*this` и `rhs`, а затем `rhs` выходит за рамки. Когда `operator=`, это неявно очищает исходные ресурсы `this` (через деструктор, который мы не должны дублировать). Самозадание работает тоже - оно менее эффективно с копированием и заменой (связано с дополнительным распределением и освобождением), но если это маловероятный сценарий, мы не замедляем типичный вариант использования для его учета.

C ++ 11

Вышеприведенная формулировка работает как-уже для назначения перемещения.

```
p1 = std::move(p2);
```

Здесь мы перемещаем-строим `rhs` из `p2`, а все остальное справедливо. Если класс движется, но не копируется, нет необходимости удалять назначение копии, так как этот оператор присваивания будет просто плохо сформирован из-за конструктора удаленной копии.

Конструктор по умолчанию

Конструктор по умолчанию - это тип конструктора, который не требует никаких параметров при вызове. Он назван в честь того типа, который он строит, и является его функцией-членом (как и все конструкторы).

```
class C{
    int i;
public:
    // the default constructor definition
    C()
    : i(0){ // member initializer list -- initialize i to 0
        // constructor function body -- can do more complex things here
    }
};

C c1; // calls default constructor of C to create object c1
C c2 = C(); // calls default constructor explicitly
C c3(); // ERROR: this intuitive version is not possible due to "most vexing parse"
C c4{}; // but in C++11 {} CAN be used in a similar way

C c5[2]; // calls default constructor for both array elements
C* c6 = new C[2]; // calls default constructor for both array elements
```

Другим способом удовлетворения требований «без параметров» является предоставление разработчикам значений по умолчанию для всех параметров:

```
class D{
    int i;
    int j;
public:
    // also a default constructor (can be called with no parameters)
    D( int i = 0, int j = 42 )
    : i(i), j(j){
    }
};

D d; // calls constructor of D with the provided default values for the parameters
```

В некоторых случаях (т.е. разработчик не предоставляет никаких конструкторов и других условий дисквалификации), компилятор неявно предоставляет пустой конструктор по умолчанию:

```
class C{
    std::string s; // note: members need to be default constructible themselves
};

C c1; // will succeed -- C has an implicitly defined default constructor
```

Наличие другого типа конструктора является одним из условий дисквалификации, упомянутых ранее:

```
class C{
    int i;
public:
    C( int i ) : i(i){}
};

C c1; // Compile ERROR: C has no (implicitly defined) default constructor
```

C ++ 11

Чтобы предотвратить неявное создание конструктора по умолчанию, общепринятой техникой является объявить ее как `private` (без определения). Цель состоит в том, чтобы вызвать ошибку компиляции, когда кто-то пытается использовать конструктор (это либо приводит к ошибке *доступа к закрытой* ошибке, либо ошибке компоновщика, в зависимости от компилятора).

Чтобы убедиться, что конструктор по умолчанию (функционально подобный неявному) определен, разработчик может написать пустой ящик явно.

C ++ 11

В C ++ 11 разработчик также может использовать ключевое слово `delete` чтобы компилятор не предоставлял конструктор по умолчанию.

```
class C{
    int i;
```

```
public:
    // default constructor is explicitly deleted
    C() = delete;
};

C c1; // Compile ERROR: C has its default constructor deleted
```

Кроме того, разработчик также может быть явно о том, что компилятор должен предоставить конструктор по умолчанию.

```
class C{
    int i;
public:
    // does have automatically generated default constructor (same as implicit one)
    C() = default;

    C( int i ) : i(i){}
};

C c1; // default constructed
C c2( 1 ); // constructed with the int taking constructor
```

C ++ 14

Вы можете определить, имеет ли тип конструктор по умолчанию (или является примитивным типом), используя `std::is_default_constructible` из `<type_traits>` :

```
class C1{ };
class C2{ public: C2(){} };
class C3{ public: C3(int){} };

using std::cout; using std::boolalpha; using std::endl;
using std::is_default_constructible;
cout << boolalpha << is_default_constructible<int>() << endl; // prints true
cout << boolalpha << is_default_constructible<C1>() << endl; // prints true
cout << boolalpha << is_default_constructible<C2>() << endl; // prints true
cout << boolalpha << is_default_constructible<C3>() << endl; // prints false
```

C ++ 11

В C ++ 11 все еще можно использовать версию `std::is_default_constructible` для `std::is_default_constructible` :

```
cout << boolalpha << is_default_constructible<C1>::value << endl; // prints true
```

Destructor

Деструктор - это функция без аргументов, которая вызывается, когда пользовательский объект будет уничтожен. Она названа в честь типа он Уничтожает с ~ префиксом.

```
class C{
    int* is;
    string s;
```



```

public:
    C()
    : is( new int[10] ){
    }

    ~C(){ // destructor definition
        delete[] is;
    }
};

class C_child : public C{
    string s_ch;
public:
    C_child(){}
    ~C_child(){} // child destructor
};

void f(){
    C c1; // calls default constructor
    C c2[2]; // calls default constructor for both elements
    C* c3 = new C[2]; // calls default constructor for both array elements

    C_child c_ch; // when destructed calls destructor of s_ch and of C base (and in turn s)

    delete[] c3; // calls destructors on c3[0] and c3[1]
} // automatic variables are destroyed here -- i.e. c1, c2 and c_ch

```

В большинстве случаев (т. Е. Пользователь не предоставляет деструктора и других условий дисквалификации), компилятор предоставляет деструктор по умолчанию неявно:

```

class C{
    int i;
    string s;
};

void f(){
    C* c1 = new C;
    delete c1; // C has a destructor
}

```

```

class C{
    int m;
private:
    ~C(){} // not public destructor!
};

class C_container{
    C c;
};

void f(){
    C_container* c_cont = new C_container;
    delete c_cont; // Compile ERROR: C has no accessible destructor
}

```

В C++ 11 разработчик может переопределить это поведение, запретив компилятору предоставлять деструктор по умолчанию.

```
class C{
    int m;
public:
    ~C() = delete; // does NOT have implicit destructor
};

void f{
    C c1;
} // Compile ERROR: C has no destructor
```

Кроме того, разработчик также может быть явно о том, что компилятор должен предоставить деструктор по умолчанию.

```
class C{
    int m;
public:
    ~C() = default; // saying explicitly it does have implicit/empty destructor
};

void f(){
    C c1;
} // C has a destructor -- c1 properly destroyed
```

C++ 11

Вы можете определить, имеет ли тип деструктор (или является примитивным типом), используя `std::is_destructible` из `<type_traits>`:

```
class C1{ };
class C2{ public: ~C2() = delete };
class C3 : public C2{ };

using std::cout; using std::boolalpha; using std::endl;
using std::is_destructible;
cout << boolalpha << is_destructible<int>() << endl; // prints true
cout << boolalpha << is_destructible<C1>() << endl; // prints true
cout << boolalpha << is_destructible<C2>() << endl; // prints false
cout << boolalpha << is_destructible<C3>() << endl; // prints false
```

Прочитайте Специальные функции участников онлайн:

<https://riptutorial.com/ru/cplusplus/topic/1476/специальные-функции-участников>

глава 105: Спецификаторы класса хранения

Вступление

Спецификаторы класса хранения - это **ключевые слова**, которые могут использоваться в объявлениях. Они не влияют на тип объявления, но обычно изменяют способ хранения объекта.

замечания

Существует шесть спецификаций класса хранения, хотя и не все в одной и той же версии языка: `auto` (до C ++ 11), `register` (до C ++ 17), `static`, `thread_local` (начиная с C ++ 11), `extern` и `mutable`.

Согласно стандарту,

Не более одного *спецификатора класса хранения* должно появляться в заданном *spec-spec-seq*, за исключением того, что `thread_local` может отображаться со `static` или `extern`.

Объявление может содержать спецификатор класса хранения. В этом случае язык определяет поведение по умолчанию. Например, по умолчанию переменная, объявленная в области блока, неявно имеет автоматическую продолжительность хранения.

Examples

изменчивый

Спецификатор, который может быть применен к объявлению нестатического, не ссылочного элемента данных класса. Изменчивый член класса не `const` даже если объект `const`.

```
class C {
    int x;
    mutable int times_accessed;
public:
    C(): x(0), times_accessed(0) {
    }
    int get_x() const {
        ++times_accessed; // ok: const member function can modify mutable data member
        return x;
    }
    void set_x(int x) {
```

```

        ++times_accessed;
        this->x = x;
    }
};

```

C ++ 11

Второе значение для `mutable` было добавлено в C ++ 11. Когда он следует за списком параметров лямбда, он подавляет неявный `const` в операторе вызова функции лямбда. Следовательно, изменяемая лямбда может изменять значения объектов, захваченных копией. См. [Изменяемые лямбды](#) для более подробной информации.

```

std::vector<int> my_iota(int start, int count) {
    std::vector<int> result(count);
    std::generate(result.begin(), result.end(),
        [start]() mutable { return start++; });
    return result;
}

```

Обратите внимание, что `mutable` *не* является спецификатором класса хранения, когда используется таким образом, чтобы сформировать изменяемую лямбду.

регистр

C ++ 17

Спецификатор класса хранения, который подсказывает компилятору, что переменная будет сильно использоваться. Слово «регистр» связано с тем, что компилятор может захотеть сохранить такую переменную в регистре CPU, чтобы к ней можно было получить доступ за меньшее количество тактов. Он устарел, начиная с C ++ 11.

```

register int i = 0;
while (i < 100) {
    f(i);
    int g = i*i;
    i += h(i, g);
}

```

Локальные переменные и параметры функции могут быть объявлены `register`. В отличие от C, C ++ не устанавливает никаких ограничений на то, что можно сделать с помощью переменной `register`. Например, можно принять адрес переменной `register`, но это может помешать компилятору фактически сохранить такую переменную в регистре.

C ++ 17

`register` ключевых слов не используется и сохраняется. Программа, использующая `register` ключевых слов, плохо сформирована.

статический

Спецификатор класса `static` хранилища имеет три разных значения.

1. Дает внутреннюю связь с переменной или функцией, объявленной в области пространства имен.

```
// internal function; can't be linked to
static double semiperimeter(double a, double b, double c) {
    return (a + b + c)/2.0;
}
// exported to client
double area(double a, double b, double c) {
    const double s = semiperimeter(a, b, c);
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

2. Объявляет переменную для статической продолжительности хранения (если она не является `thread_local`). Переменные пространства-пространства неявно статичны. Статическая локальная переменная инициализируется только один раз, первый контроль времени проходит через ее определение и не уничтожается при каждом выходе из нее.

```
void f() {
    static int count = 0;
    std::cout << "f has been called " << ++count << " times so far\n";
}
```

3. При применении к объявлению члена класса объявляет этот элемент **статическим членом**.

```
struct S {
    static S* create() {
        return new S;
    }
};
int main() {
    S* s = S::create();
}
```

Обратите внимание, что в случае статического члена данных класса оба одновременно применяются как 2, так и 3: `static` ключевое слово делает его членом в статическом элементе данных и превращает его в переменную со статической продолжительностью хранения.

авто

C ++ 03

Объявляет переменную для автоматического хранения. Это избыточно, поскольку автоматическая длительность хранения уже является значением по умолчанию в области блока, а авто-спецификатор не разрешен в области пространства имен.

```
void f() {
    auto int x; // equivalent to: int x;
    auto y;      // illegal in C++; legal in C89
}
auto int z;      // illegal: namespace-scope variable cannot be automatic
```

В C++ 11 `auto` изменяется смысл и больше не является спецификатором класса хранения, а вместо этого используется для [вывода типа](#).

ВНЕШНИЙ

Спецификатор класса `extern` хранилища может изменять объявление одним из трех следующих способов, в зависимости от контекста:

1. Его можно использовать для объявления переменной без ее определения. Как правило, это используется в файле заголовка для переменной, которая будет определена в отдельном файле реализации.

```
// global scope
int x;                // definition; x will be default-initialized
extern int y;          // declaration; y is defined elsewhere, most likely another TU
extern int z = 42;     // definition; "extern" has no effect here (compiler may warn)
```

2. Это дает внешнюю связь с переменной в пространстве имен, даже если `const` или `constexpr` иначе вызвали бы ее внутреннюю связь.

```
// global scope
const int w = 42;      // internal linkage in C++; external linkage in C
static const int x = 42; // internal linkage in both C++ and C
extern const int y = 42; // external linkage in both C++ and C
namespace {
    extern const int z = 42; // however, this has internal linkage since
                           // it's in an unnamed namespace
}
```

3. Он переопределяет переменную в области блока, если она была ранее объявлена с помощью связи. В противном случае он объявляет новую переменную с привязкой, которая является членом ближайшего охватывающего пространства имен.

```
// global scope
namespace {
    int x = 1;
    struct C {
        int x = 2;
        void f() {
            extern int x; // redeclares namespace-scope x
            std::cout << x << '\n'; // therefore, this prints 1, not 2
        }
    };
}
void g() {
    extern int y; // y has external linkage; refers to global y defined elsewhere
```

```
}
```

Функция также может быть объявлена `extern`, но это не имеет никакого эффекта. Он обычно используется как подсказка для читателя о том, что заявленная здесь функция определена в другой единицы перевода. Например:

```
void f();           // typically a forward declaration; f defined later in this TU
extern void g();    // typically not a forward declaration; g defined in another TU
```

В приведенном выше коде, если `f` был изменен на `extern` и `g` на `non extern`, это никак не повлияет на правильность или семантику программы, но скорее всего путает читателя кода.

Прочитайте Спецификаторы класса хранения онлайн:

<https://riptutorial.com/ru/cplusplus/topic/9225/спецификаторы-класса-хранения>

глава 106: Сравнение сборок с классическими примерами C ++, разрешенными с помощью C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17

Examples

Прохождение через контейнер

В C ++ цикл через контейнер последовательности `c` может быть выполнен с использованием индексов следующим образом:

```
for(size_t i = 0; i < c.size(); ++i) c[i] = 0;
```

В то время как простые, такие записи подвержены общим семантическим ошибкам, таким как неправильный оператор сравнения или неправильная переменная индексации:

```
for(size_t i = 0; i <= c.size(); ++j) c[i] = 0;
                ^~~~~~^
```

Цикл также может быть достигнут для всех контейнеров с использованием итераторов с аналогичными недостатками:

```
for(iterator it = c.begin(); it != c.end(); ++it) (*it) = 0;
```

В C ++ 11 введены диапазоны для циклов и ключевое слово `auto`, что позволяет коду стать:

```
for(auto& x : c) x = 0;
```

Здесь единственными параметрами являются контейнер `c`, а переменная `x` - текущее значение. Это предотвращает преждевременные ошибки семантики.

Согласно стандарту C ++ 11, базовая реализация эквивалентна:

```
for(auto begin = c.begin(), end = c.end(); begin != end; ++begin)
{
    // ...
}
```

В такой реализации выражение `auto begin = c.begin(), end = c.end();` силы `begin` и `end`

одинаковым типом, а `end` никогда не увеличивается и не разыгрывается. Таким образом, цикл, основанный на диапазоне, работает только для контейнеров, определяемых паре `iterator / iterator`. Стандарт C ++ 17 смягчает это ограничение, изменяя реализацию:

```
auto begin = c.begin();
auto end = c.end();
for(; begin != end; ++begin)
{
    // ...
}
```

Здесь `begin` и `end` допускаются к разным типам, если их можно сравнить с неравенством. Это позволяет прокручивать больше контейнеров, например, контейнер, определенный парой итератор / дозорный.

Прочитайте Сравнение сборок с классическими примерами C ++, разрешенными с помощью C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17 онлайн: <https://riptutorial.com/ru/cplusplus/topic/7134/сравнение-сборок-с-классическими-примерами-c-plusplus--разрешенными-с-помощью-c-plusplus-vs-c-plusplus-11-vs-c-plusplus-14-vs-c-plusplus-17>

глава 107: Средства и методы отладки и отладки C ++

Вступление

Отладка от разработчиков C ++ отнимает много времени. Эта тема призвана помочь в решении этой задачи и вдохновить техник. Не ожидайте обширного списка проблем и решений, установленных инструментами или руководством по указанным инструментам.

замечания

Эта тема еще не завершена, примеры из следующих методов / инструментов были бы полезны:

- Упомяните больше инструментов статического анализа
- Инструменты двоичных инструментов (например, UBSan, TSan, MSan, ESan ...)
- Закалка (CFI ...)
- Fuzzing

Examples

Моя программа на C ++ заканчивается segfault - valgrind

Давайте попробуем выполнить базовую неудачную программу:

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p3 << std::endl;
    }
}

int main() {
    fail();
}
```

Постройте его (добавьте -g для включения информации об отладке):

```
g++ -g -o main main.cpp
```

Бежать:

```
$ ./main
Segmentation fault (core dumped)
$
```

Давайте отлаживаем его с valgrind:

```
$ valgrind ./main
==8515== Memcheck, a memory error detector
==8515== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==8515== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==8515== Command: ./main
==8515==
==8515== Conditional jump or move depends on uninitialised value(s)
==8515==   at 0x400813: fail() (main.cpp:7)
==8515==   by 0x40083F: main (main.cpp:13)
==8515==
==8515== Invalid read of size 4
==8515==   at 0x400819: fail() (main.cpp:8)
==8515==   by 0x40083F: main (main.cpp:13)
==8515== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==8515==
==8515==
==8515== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==8515== Access not within mapped region at address 0x0
==8515==   at 0x400819: fail() (main.cpp:8)
==8515==   by 0x40083F: main (main.cpp:13)
==8515== If you believe this happened as a result of a stack
==8515== overflow in your program's main thread (unlikely but
==8515== possible), you can try to increase the size of the
==8515== main thread stack using the --main-stacksize= flag.
==8515== The main thread stack size used in this run was 8388608.
==8515==
==8515== HEAP SUMMARY:
==8515==   in use at exit: 72,704 bytes in 1 blocks
==8515==   total heap usage: 1 allocs, 0 frees, 72,704 bytes allocated
==8515==
==8515== LEAK SUMMARY:
==8515==   definitely lost: 0 bytes in 0 blocks
==8515==   indirectly lost: 0 bytes in 0 blocks
==8515==   possibly lost: 0 bytes in 0 blocks
==8515==   still reachable: 72,704 bytes in 1 blocks
==8515==   suppressed: 0 bytes in 0 blocks
==8515== Rerun with --leak-check=full to see details of leaked memory
==8515==
==8515== For counts of detected and suppressed errors, rerun with: -v
==8515== Use --track-origins=yes to see where uninitialised values come from
==8515== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
$
```

Сначала мы сосредоточимся на этом блоке:

```
==8515== Invalid read of size 4
==8515==   at 0x400819: fail() (main.cpp:8)
==8515==   by 0x40083F: main (main.cpp:13)
==8515== Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

В первой строке указывается, что segfault вызвано чтением 4 байтов. Вторая и третья

строки - стек вызовов. Это означает, что недопустимое чтение выполняется в функции `fail()` , строка 8 `main.cpp`, которая вызывается основной строкой 13 `main.cpp`.

Рассматривая строку 8 `main.cpp`, мы видим

```
std::cout << *p3 << std::endl;
```

Но сначала проверяем указатель, так что что случилось? Позволяет проверить другой блок:

```
==8515== Conditional jump or move depends on uninitialised value(s)
==8515==    at 0x400813: fail() (main.cpp:7)
==8515==    by 0x40083F: main (main.cpp:13)
```

Это говорит нам, что в строке 7 есть униализированная переменная, и мы ее читаем:

```
if (p3) {
```

Что указывает нам на прямую, где мы проверяем `p3` вместо `p2`. Но как же возможно, что `p3` неинициализируется? Мы инициализируем его:

```
int *p3 = p1;
```

Valgrind советует нам повторить с `--track-origins=yes` , давайте сделаем это:

```
valgrind --track-origins=yes ./main
```

Аргумент для `valgrind` сразу после `valgrind`. Если мы поместим его после нашей программы, он будет передан нашей программе.

Вывод почти такой же, есть только одно отличие:

```
==8517== Conditional jump or move depends on uninitialised value(s)
==8517==    at 0x400813: fail() (main.cpp:7)
==8517==    by 0x40083F: main (main.cpp:13)
==8517== Uninitialised value was created by a stack allocation
==8517==    at 0x4007F6: fail() (main.cpp:3)
```

Что говорит нам, что неинициализированное значение, которое мы использовали в строке 7, было создано в строке 3:

```
int *p1;
```

который ведет нас к нашему неинициализированному указателю.

Анализ Segfault с GDB

Для этого примера используется тот же код, что и выше.

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p2 << std::endl;
    }
}

int main() {
    fail();
}
```

Сначала скомпилируем его

```
g++ -g -o main main.cpp
```

Позволяет запустить его с помощью gdb

```
gdb ./main
```

Теперь мы будем в оболочке gdb. Введите run.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/opencog/code-snippets/stackoverflow/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400850 in fail () at debugging_with_gdb.cc:11
11          std::cout << *p2 << std::endl;
```

Мы видим, что ошибка сегментации происходит в строке 11. Таким образом, единственной переменной, используемой в этой строке, является указатель p2. Давайте рассмотрим его печатную печать содержимого.

```
(gdb) print p2
$1 = (int *) 0x0
```

Теперь мы видим, что p2 был инициализирован равным 0x0, что означает NULL. В этой строке мы знаем, что мы пытаемся разыменовать указатель NULL. Поэтому мы и исправим это.

Чистый код

Отладка начинается с понимания кода, который вы пытаетесь отлаживать.

Плохой код:

```
int main() {
    int value;
    std::vector<int> vectorToSort;
    vectorToSort.push_back(42); vectorToSort.push_back(13);
    for (int i = 52; i; i = i - 1)
    {
        vectorToSort.push_back(i * 2);
    }
    /// Optimized for sorting small vectors
    if (vectorToSort.size() == 1);
    else
    {
        if (vectorToSort.size() <= 2)
            std::sort(vectorToSort.begin(), std::end(vectorToSort));
    }
    for (value : vectorToSort) std::cout << value << ' ';
    return 0; }
```

Лучший код:

```
std::vector<int> createSemiRandomData() {
    std::vector<int> data;
    data.push_back(42);
    data.push_back(13);
    for (int i = 52; i; --i)
        vectorToSort.push_back(i * 2);
    return data;
}

/// Optimized for sorting small vectors
void sortVector(std::vector &v) {
    if (vectorToSort.size() == 1)
        return;
    if (vectorToSort.size() > 2)
        return;

    std::sort(vectorToSort.begin(), vectorToSort.end());
}

void printVector(const std::vector<int> &v) {
    for (auto i : v)
        std::cout << i << ' ';
}

int main() {
    auto vectorToSort = createSemiRandomData();
    sortVector(std::ref(vectorToSort));
    printVector(vectorToSort);

    return 0;
}
```

Независимо от стилей кодирования, которые вы предпочитаете и используете, наличие последовательного стиля кодирования (и форматирования) поможет вам понять код.

Рассматривая приведенный выше код, можно выделить пару улучшений для улучшения

удобочитаемости и отладки:

Использование отдельных функций для отдельных действий

Использование отдельных функций позволяет пропустить некоторые функции в отладчике, если вас не интересуют детали. В этом конкретном случае вам может быть не интересно создавать или распечатывать данные и только хотеть входить в сортировку.

Еще одно преимущество заключается в том, что вам нужно читать меньше кода (и запоминать его), выполняя код. Теперь вам нужно только прочитать 3 строки кода в `main()`, чтобы понять это, а не целую функцию.

Третье преимущество заключается в том, что у вас просто меньше кода для просмотра, что помогает обученному глазу обнаружить эту ошибку за считанные секунды.

Использование последовательного форматирования / построения

Использование последовательного форматирования и построения приведет к удалению помех из кода, что упростит фокусировку на коде вместо текста. Многие дискуссии обсуждались в стиле «правильного» форматирования. Независимо от этого стиля, наличие единого кода в коде улучшит знакомство и упростит фокусировку на коде.

Поскольку код форматирования занимает много времени, рекомендуется использовать для этого специальный инструмент. Большинство IDE имеют хотя бы некоторую поддержку для этого и могут сделать форматирование более последовательным, чем люди.

Вы можете заметить, что стиль не ограничивается пробелами и символами новой строки, так как мы больше не смешиваем функции `free-style` и `member`, чтобы получить начало / конец контейнера. (`v.begin()` VS `std::end(v)`).

Обратите внимание на важные части вашего кода.

Независимо от стиля, который вы определяете для выбора, приведенный выше код содержит пару маркеров, которые могут дать вам подсказку о том, что может быть важно:

- Комментарий, который `optimized`, указывает на некоторые причудливые методы
- Некоторые ранние возвращения в `sortVector()` показывают, что мы делаем что-то особенное
- `std::ref()` указывает, что что-то происходит с `sortVector()`

Заключение

Чистый код поможет вам понять код и сократит время, необходимое для его отладки. Во втором примере анализатор кода может даже обнаружить ошибку на первый взгляд, в то время как ошибка может быть скрыта в деталях в первом. (PS: ошибка находится в сравнении с 2).

Статический анализ

Статический анализ - это метод, при котором проверяется код шаблонов, связанных с известными ошибками. Использование этого метода занимает меньше времени, чем анализ кода, однако его проверки ограничиваются только запрограммированными в инструменте.

Проверки могут включать неправильную полуточку за оператором `if (var); ( if (var); )` до расширенных алгоритмов графа, которые определяют, не инициализирована ли переменная.

Предупреждения компилятора

Включение статического анализа легко, самая упрощенная версия уже встроена в ваш компилятор:

- `clang++ -Wall -Weverything -Werror ...`
- `g++ -Wall -Weverything -Werror ...`
- `cl.exe /W4 /WX ...`

Если вы включите эти параметры, вы заметите, что каждый компилятор обнаружит ошибки, которые другие не делают, и что вы получите ошибки в тех методах, которые могут быть действительными или действительными в определенном контексте. `while (staticAtomicBool);` может быть приемлемым, даже если `while (localBool);` нет.

Поэтому, в отличие от обзора кода, вы боретесь с инструментом, который понимает ваш код, сообщает вам много полезных ошибок и иногда не соглашается с вами. В этом последнем случае вам может потребоваться локальное предупреждение.

Поскольку приведенные выше опции включают все предупреждения, они могут включать предупреждения, которые вам не нужны. (Почему ваш код должен быть совместим с C ++ 98?) Если это так, вы можете просто отключить это конкретное предупреждение:

- `clang++ -Wall -Weverything -Werror -Wno-error-to-accept ...`
- `g++ -Wall -Weverything -Werror -Wno-error-to-accept ...`
- `cl.exe /W4 /WX /wd<no of warning>...`

Когда предупреждения компилятора помогают вам во время разработки, они немного замедляют компиляцию. Вот почему вы не всегда можете включить их по умолчанию. Либо вы запускаете их по умолчанию, либо включаете некоторую непрерывную интеграцию с

более дорогостоящими проверками (или всеми из них).

Внешние инструменты

Если вы решите провести некоторую непрерывную интеграцию, использование других инструментов не является таким растяжкой. У инструмента, такого как [clang-tidy](#), есть [список проверок](#), который охватывает широкий круг проблем, некоторые примеры:

- Актуальные ошибки
 - Предотвращение нарезки
 - Утверждения с побочными эффектами
- Проверка читаемости
 - Вводящий в заблуждение отступ
 - Идентификация идентификатора
- Проверка модернизации
 - Использовать `make_unique()`
 - Использовать `nullptr`
- Проверка производительности
 - Найти ненужные копии
 - Поиск неэффективных вызовов алгоритмов

Список может быть не таким большим, поскольку у Клана уже много предупреждений компилятора, однако он приблизит вас на один шаг к высококачественной кодовой базе.

Другие инструменты

Существуют и другие инструменты с аналогичной целью:

- [статический анализатор визуальной студии](#) как внешний инструмент
- [clazy](#) , плагин компилятора Clang для проверки кода Qt

Заключение

Для C++ существует множество инструментов статического анализа, которые встроены в компилятор в качестве внешних инструментов. Попытка их не занимает много времени для простых настроек, и они найдут ошибки, которые могут упустить при просмотре кода.

Безопасный стек (повреждение стека)

Коррупция стека вызывает раздражающие ошибки. Когда стек поврежден, отладчик часто не может дать вам хорошую трассировку стека, где вы находитесь, и как вы туда попали.

Здесь вступает в действие безопасный стек. Вместо использования одного стека для

ваших потоков он будет использовать два: безопасный стек и опасный стек. Безопасный стек работает так же, как и раньше, за исключением того, что некоторые части перемещаются в опасный стек.

Какие части стека перемещаются?

Каждая часть, которая имеет потенциал для разложения стека, будет выведена из безопасного стека. Как только переменная в стеке передается по ссылке или одна берет адрес этой переменной, компилятор решит выделить это во втором стеке, а не в безопасном.

В результате любая операция, которую вы выполняете с этими указателями, любые изменения, внесенные вами в память (на основе этих указателей / ссылок), могут влиять только на память во втором стеке. Поскольку никогда не получается указатель, который находится рядом с безопасным стеком, стек не может повредить стек, и отладчик все равно может прочитать все функции в стеке, чтобы получить хорошую трассировку.

Для чего он фактически используется?

Безопасный стек не был изобретен, чтобы дать вам лучший опыт отладки, однако это хороший побочный эффект для неприятных ошибок. Это первоначальная цель - это часть проекта [Integrity \(CPI\) Code-Pointer Integrity \(CPI\)](#), в котором они пытаются предотвратить переопределение обратных адресов для предотвращения ввода кода. Другими словами, они пытаются предотвратить выполнение кода хакеров.

По этой причине эта функция активирована на хrome и, как [сообщается](#), имеет накладные расходы <1% ЦП.

Как включить его?

Прямо сейчас этот параметр доступен только в [компиляторе clang](#), где можно передать `-fsanitize=safe-stack` в компилятор. Было [предложено](#) реализовать ту же функцию в GCC.

Заключение

Повреждения стека могут стать более легкими для отладки при включении безопасного стека. Из-за низкой производительности, вы можете даже активировать по умолчанию в своей конфигурации сборки.

Прочитайте [Средства и методы отладки и отладки C ++ онлайн](#):

<https://riptutorial.com/ru/cplusplus/topic/9814/средства-и-методы-отладки-и-отладки-c-plusplus>

глава 108: станд :: forward_list

Вступление

`std::forward_list` - это контейнер, который поддерживает быструю установку и удаление элементов из любого места в контейнере. Быстрый случайный доступ не поддерживается. Он реализован как односвязный список и по существу не имеет каких-либо накладных расходов по сравнению с его реализацией в С. По сравнению с `std::list` этот контейнер обеспечивает более экономичное пространство, когда двунаправленная итерация не нужна.

замечания

Добавление, удаление и перемещение элементов внутри списка или нескольких списков не приводит к недействительности итераторов, которые в настоящее время ссылаются на другие элементы в списке. Однако итератор или ссылка, ссылающаяся на элемент, недействительна, когда соответствующий элемент удаляется (через `erase_after`) из списка. `std::forward_list` соответствует требованиям Container (за исключением функции члена `size` и сложность оператора `==` всегда линейна), `AllocatorAwareContainer` и `SequenceContainer`.

Examples

пример

```
#include <forward_list>
#include <string>
#include <iostream>

template<typename T>
std::ostream& operator<<(std::ostream& s, const std::forward_list<T>& v) {
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
}

int main()
{
    // c++11 initializer list syntax:
    std::forward_list<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};
    std::cout << "words1: " << words1 << '\n';

    // words2 == words1
```

```

std::forward_list<std::string> words2(words1.begin(), words1.end());
std::cout << "words2: " << words2 << '\n';

// words3 == words1
std::forward_list<std::string> words3(words1);
std::cout << "words3: " << words3 << '\n';

// words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
std::forward_list<std::string> words4(5, "Mo");
std::cout << "words4: " << words4 << '\n';
}

```

Выход:

```

words1: [the, frogurt, is, also, cursed]
words2: [the, frogurt, is, also, cursed]
words3: [the, frogurt, is, also, cursed]
words4: [Mo, Mo, Mo, Mo, Mo]

```

методы

Имя метода	Определение
operator=	присваивает значения контейнеру
assign	присваивает значения контейнеру
get_allocator	возвращает связанный распределитель
-----	-----
Доступ к элементу	
front	доступ к первому элементу
-----	-----
итераторы	
before_begin	возвращает итератор элементу перед началом
cbegin	возвращает константный итератор элементу перед началом
begin	возвращает итератор в начало
cbegin	возвращает константный итератор в начало
end	возвращает итератор до конца
cend	возвращает итератор до конца

Имя метода	Определение
Вместимость	
empty	проверяет, пуст ли контейнер
max_size	возвращает максимально возможное количество элементов
Модификаторы	
clear	очищает содержимое
insert_after	вставляет элементы после элемента
emplace_after	создает элементы на месте после элемента
erase_after	стирает элемент после элемента
push_front	вставляет элемент в начало
emplace_front	сначала создает элемент на месте
pop_front	удаляет первый элемент
resize	изменяет количество сохраненных элементов
swap	своп содержимого
операции	
merge	объединяет два отсортированных списка
splice_after	перемещает элементы из другого forward_list
remove	удаляет элементы, удовлетворяющие определенным критериям
remove_if	удаляет элементы, удовлетворяющие определенным критериям
reverse	меняет порядок элементов
unique	удаляет последовательные дублирующие элементы
sort	сортирует элементы

Прочитайте стандарт :: forward_list онлайн: <https://riptutorial.com/ru/cplusplus/topic/9703/станд----forward-list>

глава 109: станд :: integer_sequence

Вступление

Шаблон класса `std::integer_sequence<Type, Values...>` представляет последовательность значений типа `Type` где `Type` является одним из встроенных целочисленных типов. Эти последовательности используются при реализации шаблонов классов или функций, которые извлекают выгоду из позиционного доступа. Стандартная библиотека также содержит «фабричные» типы, которые создают восходящие последовательности целочисленных значений только из числа элементов.

Examples

Поверните `std::tuple` в функциональные параметры

`std::tuple<T...>` может использоваться для передачи нескольких значений. Например, его можно использовать для хранения последовательности параметров в какой-то форме очереди. При обработке такого кортежа его элементы должны быть превращены в аргументы вызова функции:

```
#include <array>
#include <iostream>
#include <string>
#include <tuple>
#include <utility>

// -----
// Example functions to be called:
void f(int i, std::string const& s) {
    std::cout << "f(" << i << ", " << s << ")\n";
}
void f(int i, double d, std::string const& s) {
    std::cout << "f(" << i << ", " << d << ", " << s << ")\n";
}
void f(char c, int i, double d, std::string const& s) {
    std::cout << "f(" << c << ", " << i << ", " << d << ", " << s << ")\n";
}
void f(int i, int j, int k) {
    std::cout << "f(" << i << ", " << j << ", " << k << ")\n";
}

// -----
// The actual function expanding the tuple:
template <typename Tuple, std::size_t... I>
void process(Tuple const& tuple, std::index_sequence<I...>) {
    f(std::get<I>(tuple)...);
}

// The interface to call. Sadly, it needs to dispatch to another function
// to deduce the sequence of indices created from std::make_index_sequence<N>
```

```

template <typename Tuple>
void process(Tuple const& tuple) {
    process(tuple, std::make_index_sequence<std::tuple_size<Tuple>::value>());
}

// -----
int main() {
    process(std::make_tuple(1, 3.14, std::string("foo")));
    process(std::make_tuple('a', 2, 2.71, std::string("bar")));
    process(std::make_pair(3, std::string("pair")));
    process(std::array<int, 3>{ 1, 2, 3 });
}

```

Пока класс поддерживает `std::get<I>(object)` и `std::tuple_size<T>::value` он может быть расширен с помощью указанной выше функции `process()`. Сама функция полностью не зависит от количества аргументов.

Создайте пакет параметров, состоящий из целых чисел

Сама `std::integer_sequence` - это сохранение последовательности целых чисел, которая может быть преобразована в пакет параметров. Его основной ценностью является возможность создания шаблонов класса «фабрика», создающих эти последовательности:

```

#include <iostream>
#include <initializer_list>
#include <utility>

template <typename T, T... I>
void print_sequence(std::integer_sequence<T, I...>) {
    std::initializer_list<bool>{ bool(std::cout << I << ' ')... };
    std::cout << '\n';
}

template <int Offset, typename T, T... I>
void print_offset_sequence(std::integer_sequence<T, I...>) {
    print_sequence(std::integer_sequence<T, T(I + Offset)...>());
}

int main() {
    // explicitly specify sequences:
    print_sequence(std::integer_sequence<int, 1, 2, 3>());
    print_sequence(std::integer_sequence<char, 'f', 'o', 'o'>());

    // generate sequences:
    print_sequence(std::make_index_sequence<10>());
    print_sequence(std::make_integer_sequence<short, 10>());
    print_offset_sequence<'A'>(std::make_integer_sequence<char, 26>());
}

```

В `print_sequence()` функции `print_sequence()` используется `std::initializer_list<bool>` при расширении целочисленной последовательности, чтобы гарантировать порядок оценки и не создавать неиспользуемую переменную `[array]`.

Поверните последовательность индексов в копии элемента

Развертывание пакета параметров индексов в выражении запятой со значением создает копию значения для каждого из индексов. К сожалению, `gcc` и `clang` считают, что индекс не имеет никакого эффекта и предупреждает об этом (`gcc` можно заставить замолчать, указав индекс на `void`):

```
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>
#include <string>
#include <utility>

template <typename T, std::size_t... I>
std::array<T, sizeof...(I)> make_array(T const& value, std::index_sequence<I...>) {
    return std::array<T, sizeof...(I)>{ (I, value)... };
}

template <int N, typename T>
std::array<T, N> make_array(T const& value) {
    return make_array(value, std::make_index_sequence<N>());
}

int main() {
    auto array = make_array<20>(std::string("value"));
    std::copy(array.begin(), array.end(),
              std::ostream_iterator<std::string>(std::cout, " "));
    std::cout << "\n";
}
```

Прочитайте стандарт `integer_sequence` онлайн: <https://riptutorial.com/ru/cplusplus/topic/8315/станд----integer-sequence>

глава 110: станд :: iomanip

Examples

станд :: setw

```
int val = 10;
// val will be printed to the extreme left end of the output console:
std::cout << val << std::endl;
// val will be printed in an output field of length 10 starting from right end of the field:
std::cout << std::setw(10) << val << std::endl;
```

Эти результаты:

```
10
      10
1234567890
```

(где последняя строка там, чтобы помочь увидеть смещения персонажа).

Иногда нам нужно установить ширину поля вывода, как правило, когда нам нужно получить результат в некотором структурированном и правильном макете. Это можно сделать, используя `std::setw` из `std :: iomanip` .

Синтаксис для `std::setw` :

```
std::setw(int n)
```

где `n` - длина выходного поля, которое должно быть установлено

станд :: setprecision

При использовании в выражении `out << setprecision(n)` или `in >> setprecision(n)` , задает параметр точности потока или в точном `n`. Параметр этой функции является целым числом, что является новым значением для точности.

Пример:

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <limits>
int main()
{
    const long double pi = std::acos(-1.L);
    std::cout << "default precision (6): " << pi << '\n'
              << "std::precision(10):    " << std::setprecision(10) << pi << '\n'
```

```

        << "max precision:          "
        << std::setprecision(std::numeric_limits<long double>::digits10 + 1)
        << pi << '\n';
    }
//Output
//default precision (6): 3.14159
//std::precision(10):    3.141592654
//max precision:        3.141592653589793239

```

станд :: setfill

При использовании в выражении `out << setfill(c)` задает заполняющий символ потока до `c`.

Примечание. Текущий символ заполнения может быть получен с помощью `std::ostream::fill`.

Пример:

```

#include <iostream>
#include <iomanip>
int main()
{
    std::cout << "default fill: " << std::setw(10) << 42 << '\n'
              << "setfill('*'): " << std::setfill('*')
              << std::setw(10) << 42 << '\n';
}
//output::
//default fill:      42
//setfill('*'): *****42

```

СТД :: setiosflags

При использовании в выражении `out << setiosflags(mask)` или `in >> setiosflags(mask)`, устанавливает все флаги формата потока или в соответствии с маской.

Список всех `std::ios_base::fmtflags`:

- `dec` - десятичная база для целочисленного ввода-вывода
- `oct` - использование восьмеричной базы для целых операций ввода-вывода
- `hex` - использовать шестнадцатеричную базу для целых операций ввода-вывода
- `basefield` - `dec|oct|hex|0` полезно для операций маскировки
- `left` - левая настройка (добавьте символы заполнения вправо)
- `right` - правая (добавляет символы заполнения влево)
- `internal` - внутренняя настройка (добавляет символы заполнения во внутреннюю обозначенную точку)
- `adjustfield` - `left|right|internal`. Полезно для маскировки
- `scientific` - генерировать типы с плавающей точкой, используя научную нотацию, или шестнадцатеричную нотацию, если они объединены с фиксированными
- `fixed`

- генерируют типы с плавающей точкой с использованием фиксированной нотации или шестнадцатеричной нотации, если они объединены с научными

- `floatfield` - `scientific|fixed|(scientific|fixed)|0` . Полезно для маскировки
- `boolalpha` - вставить и извлечь `bool` тип в алфавитно-цифровом формате
- `showbase` - генерировать префикс, указывающий числовую базу для целочисленного вывода, для этого требуется индикатор валюты в валютном вводе-выводе
- `showpoint` - `showpoint` генерировать символ десятичной точки для вывода чисел с плавающей запятой
- `showpos` - генерировать символ `+` для неотрицательного числового вывода
- `skipws` - пропускать ведущие пробелы перед определенными операциями ввода
- `unitbuf` выход после каждой операции вывода
- `uppercase` - заменить некоторые строчные буквы с их прописными эквивалентами в определенных выходных операциях вывода

Пример манипуляторов:

```
#include <iostream>
#include <string>
#include <iomanip>
int main()
{
    int l_iTemp = 47;
    std::cout<< std::resetiosflags(std::ios_base::basefield);
    std::cout<<std::setiosflags( std::ios_base::oct)<<l_iTemp<<std::endl;
    //output: 57
    std::cout<< std::resetiosflags(std::ios_base::basefield);
    std::cout<<std::setiosflags( std::ios_base::hex)<<l_iTemp<<std::endl;
    //output: 2f
    std::cout<<std::setiosflags( std::ios_base::uppercase)<<l_iTemp<<std::endl;
    //output 2F
    std::cout<<std::setfill('0')<<std::setw(12);
    std::cout<<std::resetiosflags(std::ios_base::uppercase);
    std::cout<<std::setiosflags( std::ios_base::right)<<l_iTemp<<std::endl;
    //output: 00000000002f

    std::cout<<std::resetiosflags(std::ios_base::basefield|std::ios_base::adjustfield);
    std::cout<<std::setfill('.')<<std::setw(10);
    std::cout<<std::setiosflags( std::ios_base::left)<<l_iTemp<<std::endl;
    //output: 47.....

    std::cout<<std::resetiosflags(std::ios_base::adjustfield)<<std::setfill('#');
    std::cout<<std::setiosflags(std::ios_base::internal|std::ios_base::showpos);
    std::cout<<std::setw(10)<<l_iTemp<<std::endl;
    //output +#####47

    double l_dTemp = -1.2;
    double pi = 3.14159265359;
    std::cout<<pi<<"    "<<l_dTemp<<std::endl;
    //output +3.14159    -1.2
    std::cout<<std::setiosflags(std::ios_base::showpoint)<<l_dTemp<<std::endl;
    //output -1.20000
    std::cout<<setiosflags(std::ios_base::scientific)<<pi<<std::endl;
    //output: +3.141593e+00
    std::cout<<std::resetiosflags(std::ios_base::floatfield);
    std::cout<<setiosflags(std::ios_base::fixed)<<pi<<std::endl;
```

```
//output: +3.141593
bool b = true;
std::cout<<std::setiosflags(std::ios_base::unitbuf|std::ios_base::boolalpha)<<b;
//output: true
return 0;
}
```

Прочитайте стандарт :: iomanip онлайн: [https://riptutorial.com/ru/cplusplus/topic/6936/станд---
iomanip](https://riptutorial.com/ru/cplusplus/topic/6936/станд---iomanip)

глава 111: станд :: вариант

замечания

Вариант является заменой для использования в сыром `union`. Он безопасен по типу и знает, какой он тип, и он тщательно конструирует и уничтожает объекты внутри него, когда это необходимо.

Он почти никогда не бывает пустым: только в угловых случаях, когда замена его содержимого бросается, и он не может безопасно отступить, в конечном итоге он находится в пустом состоянии.

Он ведет себя как `std::tuple` и несколько похож на `std::optional`.

Использование `std::get` и `std::get_if` - обычно плохая идея. Правильный ответ, как правило, `std::visit`, который позволяет вам иметь дело с любой возможностью прямо там. `if constexpr` можно использовать в ходе `visit` если вам нужно разветвить ваше поведение, вместо того, чтобы выполнять последовательность проверок времени выполнения, которые дублируют то, что `visit` будет делать более эффективно.

Examples

Основное использование `std::variant`

Это создает вариант (тегированный союз), который может хранить либо `int` либо `string`.

```
std::variant< int, std::string > var;
```

Мы можем сохранить в нем один из них:

```
var = "hello"s;
```

И мы можем получить доступ к содержимому через `std::visit`:

```
// Prints "hello\n":
visit( [](auto&& e) {
    std::cout << e << '\n';
}, var );
```

путем прохождения в полиморфном лямбда или подобном функциональном объекте.

Если мы уверены, что знаем, что это такое, мы можем это получить:

```
auto str = std::get<std::string>(var);
```

но это будет бросать, если мы ошибаемся. `get_if` :

```
auto* str = std::get_if<std::string>(&var);
```

возвращает `nullptr` если вы ошибаетесь.

Варианты не гарантируют динамического распределения памяти (кроме того, что выделяется их содержащимися типами). Там хранится только один из типов в варианте, и в редких случаях (с исключениями при назначении и без безопасного пути для возврата) вариант может стать пустым.

Варианты позволяют хранить несколько типов значений в одной переменной безопасно и эффективно. Они в основном умные, безопасные по типу `union`.

Создание указателей псевдо-методов

Это расширенный пример.

Вы можете использовать вариант для стирания легкого веса.

```
template<class F>
struct pseudo_method {
    F f;
    // enable C++17 class type deduction:
    pseudo_method( F&& fin ):f(std::move(fin)) {}

    // Koenig lookup operator->*, as this is a pseudo-method it is appropriate:
    template<class Variant> // maybe add SFINAE test that LHS is actually a variant.
    friend decltype(auto) operator->*( Variant&& var, pseudo_method const& method ) {
        // var->*method returns a lambda that perfect forwards a function call,
        // behaving like a method pointer basically:
        return [&](auto&&...args)->decltype(auto) {
            // use visit to get the type of the variant:
            return std::visit(
                [&](auto&& self)->decltype(auto) {
                    // decltype(x)(x) is perfect forwarding in a lambda:
                    return method.f( decltype(self)(self), decltype(args)(args)... );
                },
                std::forward<Var>(var)
            );
        };
    }
};
```

это создает тип, который перегружает `operator->*` с помощью `Variant` с левой стороны.

```
// C++17 class type deduction to find template argument of `print` here.
// a pseudo-method lambda should take `self` as its first argument, then
// the rest of the arguments afterwards, and invoke the action:
pseudo_method print = [](auto&& self, auto&&...args)->decltype(auto) {
    return decltype(self)(self).print( decltype(args)(args)... );
};
```

Теперь, если у нас есть 2 типа с методом `print` :

```
struct A {
    void print( std::ostream& os ) const {
        os << "A";
    }
};
struct B {
    void print( std::ostream& os ) const {
        os << "B";
    }
};
```

обратите внимание, что они не связаны между собой. Мы можем:

```
std::variant<A,B> var = A{};

(var->*print) (std::cout);
```

и он отправит вызов непосредственно на `A::print(std::cout)` для нас. Если мы вместо этого инициализировали `var` с `B{}` , он отправил бы `B::print(std::cout)` .

Если мы создали новый тип `C`:

```
struct C {};
```

затем:

```
std::variant<A,B,C> var = A{};
(var->*print) (std::cout);
```

не будет компилироваться, потому что не существует `C.print(std::cout)` .

Расширение вышеизложенного позволит обнаружить и использовать функцию бесплатной `print` `S`, возможно, с использованием `if constexpr` в псевдо-методе `print` .

[живой пример](#) в настоящее время использует `boost::variant` вместо `std::variant` .

Построение `std::variant`

Это не распространяется на распределители.

```
struct A {};
```

```
struct B { B()=default; B(B const&)=default; B(int){}; };
struct C { C()=delete; C(int) {}; C(C const&)=default; };
struct D { D( std::initializer_list<int> ) {}; D(D const&)=default; D()=default; };

std::variant<A,B> var_ab0; // contains a A()
std::variant<A,B> var_ab1 = 7; // contains a B(7)
std::variant<A,B> var_ab2 = var_ab1; // contains a B(7)
std::variant<A,B,C> var_abc0{ std::in_place_type<C>, 7 }; // contains a C(7)
```

```
std::variant<C> var_c0; // illegal, no default ctor for C
std::variant<A,D> var_ad0( std::in_place_type<D>, {1,3,3,4} ); // contains D{1,3,3,4}
std::variant<A,D> var_ad1( std::in_place_index<0> ); // contains A{}
std::variant<A,D> var_ad2( std::in_place_index<1>, {1,3,3,4} ); // contains D{1,3,3,4}
```

Прочитайте стандарт :: вариант онлайн: <https://riptutorial.com/ru/cplusplus/topic/5239/станд-вариант>

глава 112: станд :: вектор

Вступление

Вектор представляет собой динамический массив с автоматически обрабатываемым хранилищем. Элементы в векторе могут быть доступны так же эффективно, как и элементы в массиве с преимуществом, заключающиеся в том, что векторы могут динамически изменять размер.

В терминах хранения векторные данные (обычно) помещаются в динамически распределенную память, что требует некоторых незначительных накладных расходов; наоборот, C-arrays и `std::array` используют автоматическое хранилище относительно объявленного местоположения и, следовательно, не имеют накладных расходов.

замечания

Использование `std::vector` требует включения заголовка `<vector>` используя `#include <vector>`.

Элементы в `std::vector` хранятся смежно в свободном хранилище. Следует отметить, что когда векторы вложены как в `std::vector<std::vector<int> >`, элементы каждого вектора смежны, но каждый вектор выделяет свой собственный базовый буфер в свободном хранилище.

Examples

Инициализация `std :: vector`

`std::vector` может быть [инициализирован](#) несколькими способами, объявляя его:

C ++ 11

```
std::vector<int> v{ 1, 2, 3 }; // v becomes {1, 2, 3}

// Different from std::vector<int> v(3, 6)
std::vector<int> v{ 3, 6 }; // v becomes {3, 6}
```

```
// Different from std::vector<int> v{3, 6} in C++11
std::vector<int> v(3, 6); // v becomes {6, 6, 6}

std::vector<int> v(4); // v becomes {0, 0, 0, 0}
```

Вектор может быть инициализирован из другого контейнера несколькими способами:

Скопировать построение (только с другого вектора), которое копирует данные из `v2` :

```
std::vector<int> v(v2);  
std::vector<int> v = v2;
```

C++ 11

Переместить конструкцию (только из другого вектора), которая перемещает данные из `v2` :

```
std::vector<int> v(std::move(v2));  
std::vector<int> v = std::move(v2);
```

Итератор (диапазон) `copy-construction`, который копирует элементы в `v` :

```
// from another vector  
std::vector<int> v(v2.begin(), v2.begin() + 3); // v becomes {v2[0], v2[1], v2[2]}  
  
// from an array  
int z[] = { 1, 2, 3, 4 };  
std::vector<int> v(z, z + 3); // v becomes {1, 2, 3}  
  
// from a list  
std::list<int> list1{ 1, 2, 3 };  
std::vector<int> v(list1.begin(), list1.end()); // v becomes {1, 2, 3}
```

C++ 11

Перемещение итератора с использованием `std::make_move_iterator` , который перемещает элементы в `v` :

```
// from another vector  
std::vector<int> v(std::make_move_iterator(v2.begin()),  
                 std::make_move_iterator(v2.end()));  
  
// from a list  
std::list<int> list1{ 1, 2, 3 };  
std::vector<int> v(std::make_move_iterator(list1.begin()),  
                 std::make_move_iterator(list1.end()));
```

С помощью функции-члена `assign()` , `std::vector` может быть повторно инициализирован после его построения:

```
v.assign(4, 100); // v becomes {100, 100, 100, 100}  
  
v.assign(v2.begin(), v2.begin() + 3); // v becomes {v2[0], v2[1], v2[2]}  
  
int z[] = { 1, 2, 3, 4 };  
v.assign(z + 1, z + 4); // v becomes {2, 3, 4}
```

Вставка элементов

Добавление элемента в конце вектора (путем копирования / перемещения):

```
struct Point {
    double x, y;
    Point(double x, double y) : x(x), y(y) {}
};
std::vector<Point> v;
Point p(10.0, 2.0);
v.push_back(p); // p is copied into the vector.
```

C ++ 11

Добавление элемента в конец вектора путем создания элемента на месте:

```
std::vector<Point> v;
v.emplace_back(10.0, 2.0); // The arguments are passed to the constructor of the
                           // given type (here Point). The object is constructed
                           // in the vector, avoiding a copy.
```

Обратите внимание, что `std::vector` *не* имеет функции-члена `push_front()` из-за причин производительности. Добавление элемента в начале приводит к перемещению всех существующих элементов в векторе. Если вы хотите часто вставлять элементы в начале вашего контейнера, вы можете вместо этого использовать `std::list` или `std::deque`.

Вставка элемента в любую позицию вектора:

```
std::vector<int> v{ 1, 2, 3 };
v.insert(v.begin(), 9); // v now contains {9, 1, 2, 3}
```

C ++ 11

Вставка элемента в любую позицию вектора путем построения элемента на месте:

```
std::vector<int> v{ 1, 2, 3 };
v.emplace(v.begin()+1, 9); // v now contains {1, 9, 2, 3}
```

Вставка другого вектора в любую позицию вектора:

```
std::vector<int> v(4); // contains: 0, 0, 0, 0
std::vector<int> v2(2, 10); // contains: 10, 10
v.insert(v.begin()+2, v2.begin(), v2.end()); // contains: 0, 0, 10, 10, 0, 0
```

Вставка массива в любую позицию вектора:

```
std::vector<int> v(4); // contains: 0, 0, 0, 0
int a [] = {1, 2, 3}; // contains: 1, 2, 3
v.insert(v.begin()+1, a, a+sizeof(a)/sizeof(a[0])); // contains: 0, 1, 2, 3, 0, 0, 0
```

Используйте `use reserve()` перед тем, как вставить несколько элементов, если

результатирующий векторный размер известен заранее, чтобы избежать множественных перераспределений (см. [Размер и емкость вектора](#)):

```
std::vector<int> v;
v.reserve(100);
for(int i = 0; i < 100; ++i)
    v.emplace_back(i);
```

Обязательно не делайте ошибку при вызове `resize()` в этом случае, или вы случайно создадите вектор с 200 элементами, в которых только последняя ста будет иметь значение, которое вы намеревались.

Итерация над `std::vector`

Вы можете перебирать `std::vector` несколькими способами. Для каждого из следующих разделов `v` определяется следующим образом:

```
std::vector<int> v;
```

Итерация в прямом направлении

C++ 11

```
// Range based for
for(const auto& value: v) {
    std::cout << value << "\n";
}

// Using a for loop with iterator
for(auto it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}

// Using for_each algorithm, using a function or functor:
void fun(int const& value) {
    std::cout << value << "\n";
}

std::for_each(std::begin(v), std::end(v), fun);

// Using for_each algorithm. Using a lambda:
std::for_each(std::begin(v), std::end(v), [](int const& value) {
    std::cout << value << "\n";
});
```

C++ 11

```
// Using a for loop with iterator
for(std::vector<int>::iterator it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[i] << "\n";
}
```

Итерация в обратном направлении

C ++ 14

```
// There is no standard way to use range based for for this.
// See below for alternatives.

// Using for_each algorithm
// Note: Using a lambda for clarity. But a function or functor will work
std::for_each(std::rbegin(v), std::rend(v), [](auto const& value) {
    std::cout << value << "\n";
});

// Using a for loop with iterator
for(auto rit = std::rbegin(v); rit != std::rend(v); ++rit) {
    std::cout << *rit << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[v.size() - 1 - i] << "\n";
}
```

Хотя нет встроенного способа использовать диапазон, основанный на обратном итерации; относительно просто исправить это. Диапазон, основанный на использовании `begin()` и `end()` для получения итераторов, и, таким образом, имитация этого объекта-обертки может обеспечить требуемые результаты.

C ++ 14

```
template<class C>
struct ReverseRange {
    C c; // could be a reference or a copy, if the original was a temporary
    ReverseRange(C&& cin): c(std::forward<C>(cin)) {}
    ReverseRange(ReverseRange&&)=default;
    ReverseRange& operator=(ReverseRange&&)=delete;
    auto begin() const {return std::rbegin(c);}
    auto end() const {return std::rend(c);}
};

// C is meant to be deduced, and perfect forwarded into
template<class C>
ReverseRange<C> make_ReverseRange(C&& c) {return {std::forward<C>(c)};}

int main() {
    std::vector<int> v { 1,2,3,4};
    for(auto const& value: make_ReverseRange(v)) {
        std::cout << value << "\n";
    }
}
```

Принудительные элементы const

Начиная с C ++ 11 `cbegin()` и `cend()` позволяют получить *константный итератор* для вектора, даже если вектор не `const`. Постоянный итератор позволяет вам читать, но не изменять содержимое вектора, что полезно для обеспечения корректности `const`:

C ++ 11

```
// forward iteration
for (auto pos = v.cbegin(); pos != v.cend(); ++pos) {
    // type of pos is vector<T>::const_iterator
    // *pos = 5; // Compile error - can't write via const iterator
}

// reverse iteration
for (auto pos = v.crbegin(); pos != v.crend(); ++pos) {
    // type of pos is vector<T>::const_iterator
    // *pos = 5; // Compile error - can't write via const iterator
}

// expects Functor::operand() (T&)
for_each(v.begin(), v.end(), Functor());

// expects Functor::operand() (const T&)
for_each(v.cbegin(), v.cend(), Functor())
```

C ++ 17

[as_const](#) расширяет это до итерации диапазона:

```
for (auto const& e : std::as_const(v)) {
    std::cout << e << '\n';
}
```

Это легко реализовать в более ранних версиях C ++:

C ++ 14

```
template <class T>
constexpr std::add_const_t<T>& as_const(T& t) noexcept {
    return t;
}
```

Замечание об эффективности

Поскольку класс `std::vector` - это в основном класс, который управляет динамически распределенным смежным массивом, тот же принцип, описанный [здесь](#), относится к векторам C ++. Доступ к содержимому вектора по индексу намного эффективнее при соблюдении принципа строкового порядка. Разумеется, каждый доступ к вектору также

помещает его содержимое управления в кеш, но, как обсуждалось много раз (особенно [здесь](#) и [здесь](#)), разница в производительности для итерации по `std::vector` по сравнению с необработанным массивом является незначительным. Таким образом, тот же принцип эффективности для необработанных массивов в C также применяется для `std::vector` C++.

Доступ к элементам

Существует два основных способа доступа к элементам в `std::vector`

- доступ по индексу
- [итераторы](#)

Доступ по индексу:

Это можно сделать либо с помощью оператора индекса `[]`, либо с помощью функции-члена `at()`.

Оба возвращают ссылку на элемент в соответствующей позиции в `std::vector` (если это не `vector<bool>`), так что он может быть прочитан, а также изменен (если вектор не `const`).

`[]` и `at()` отличаются тем, что `[]` не гарантированно выполняет проверку границ, а `at()`. Доступ к элементам, где `index < 0` или `index >= size` является [неопределенным поведением](#) для `[]`, а `at()` выбрасывает исключение `std::out_of_range`.

Примечание. В приведенных ниже примерах для ясности используется инициализация типа C++ 11, но операторы могут использоваться со всеми версиями (если не указано C++ 11).

C++ 11

```
std::vector<int> v{ 1, 2, 3 };
```

```
// using []
int a = v[1];    // a is 2
v[1] = 4;        // v now contains { 1, 4, 3 }

// using at()
int b = v.at(2); // b is 3
v.at(2) = 5;     // v now contains { 1, 4, 5 }
int c = v.at(3); // throws std::out_of_range exception
```

Поскольку метод `at()` выполняет проверку границ и может генерировать исключения, он медленнее, чем `[]`. Это делает `[]` предпочтительным кодом, где семантика операции гарантирует, что индекс находится в границах. В любом случае доступ к элементам векторов осуществляется в постоянное время. Это означает, что доступ к первому

элементу вектора имеет одинаковую стоимость (по времени) доступа к второму элементу, третьему элементу и так далее.

Например, рассмотрим этот цикл

```
for (std::size_t i = 0; i < v.size(); ++i) {  
    v[i] = 1;  
}
```

Здесь мы знаем, что индексная переменная `i` всегда находится в границах, поэтому было бы потерей циклов ЦП, чтобы проверить, что `i` находится в границах для каждого вызова `operator[]`.

Функции `front()` и `back()` позволяют легко обращаться к первому и последнему элементу вектора соответственно. Эти позиции часто используются, и специальные аксессоры могут быть более читабельными, чем их альтернативы, используя `[]`:

```
std::vector<int> v{ 4, 5, 6 }; // In pre-C++11 this is more verbose  
  
int a = v.front(); // a is 4, v.front() is equivalent to v[0]  
v.front() = 3;     // v now contains {3, 5, 6}  
int b = v.back();  // b is 6, v.back() is equivalent to v[v.size() - 1]  
v.back() = 7;     // v now contains {3, 5, 7}
```

Примечание . Неопределенное поведение вызывает функцию `front()` или `back()` для пустого вектора. Вы должны проверить, что контейнер не опустошить, используя `empty()` функции - члена (который проверяет, если контейнер пуст) перед вызовом `front()` или `back()`. Простой пример использования 'empty ()' для проверки пустого вектора:

```
int main ()  
{  
    std::vector<int> v;  
    int sum (0);  
  
    for (int i=1;i<=10;i++) v.push_back(i); //create and initialize the vector  
  
    while (!v.empty()) //loop through until the vector tests to be empty  
    {  
        sum += v.back(); //keep a running total  
        v.pop_back(); //pop out the element which removes it from the vector  
    }  
  
    std::cout << "total: " << sum << '\n'; //output the total to the user  
  
    return 0;  
}
```

В приведенном выше примере создается вектор с последовательностью чисел от 1 до 10. Затем он выталкивает элементы вектора до тех пор, пока вектор не станет пустым (используя «`empty ()`»), чтобы предотвратить неопределенное поведение. Затем вычисляется сумма чисел в векторе и отображается пользователю.

C ++ 11

Метод `data()` возвращает указатель на необработанную память, используемую `std::vector` для внутреннего хранения своих элементов. Это чаще всего используется при передаче векторных данных в устаревший код, который ожидает массив C-стиля.

```
std::vector<int> v{ 1, 2, 3, 4 }; // v contains {1, 2, 3, 4}
int* p = v.data(); // p points to 1
*p = 4;           // v now contains {4, 2, 3, 4}
++p;             // p points to 2
*p = 3;          // v now contains {4, 3, 3, 4}
p[1] = 2;        // v now contains {4, 3, 2, 4}
*(p + 2) = 1;    // v now contains {4, 3, 2, 1}
```

C ++ 11

Перед C ++ 11 метод `data()` можно моделировать, вызывая `front()` и принимая адрес возвращаемого значения:

```
std::vector<int> v(4);
int* ptr = &(v.front()); // or &v[0]
```

Это работает, потому что векторы всегда гарантированно сохраняют свои элементы в смежных ячейках памяти, предполагая, что содержимое вектора не переопределяет унарный `operator&`. Если это произойдет, вам придется повторно реализовать `std::addressof` в pre-C ++ 11. Он также предполагает, что вектор не пуст.

итераторы:

Итераторы более подробно объясняются в примере «Итерация по `std::vector`» и статье «[Итераторы](#)». Короче говоря, они действуют аналогично указателям на элементы вектора:

C ++ 11

```
std::vector<int> v{ 4, 5, 6 };

auto it = v.begin();
int i = *it; // i is 4
++it;
i = *it;     // i is 5
*it = 6;     // v contains { 4, 6, 6 }
auto e = v.end(); // e points to the element after the end of v. It can be
                // used to check whether an iterator reached the end of the vector:
++it;
it == v.end(); // false, it points to the element at position 2 (with value 6)
++it;
it == v.end(); // true
```

Это соответствует стандарту, что итераторы `std::vector<T>` на самом деле являются `T*`, но большинство стандартных библиотек этого не делают. Не делая этого и улучшает

сообщения об ошибках, улавливает непереносимый код и может использоваться для управления итераторами с проверками отладки в сборках без релиза. Затем в релиз-сборках класс, обтекающий базовый указатель, оптимизирован.

Вы можете сохранить ссылку или указатель на элемент вектора для косвенного доступа. Эти ссылки или указатели на элементы в `vector` остаются стабильными, и доступ остается определенным, если вы не добавляете / не удаляете элементы в элементе или перед элементом в `vector`, или вы не можете изменить `vector` емкость. Это то же самое, что и правило для недействительности итераторов.

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data() + 1;      // p points to 2
v.insert(v.begin(), 0);    // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;          // p points to 1
v.reserve(10);             // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;          // p points to 1
v.erase(v.begin());        // p is now invalid, accessing *p is a undefined behavior.
```

Использование `std::vector` в качестве массива C

Существует несколько способов использования `std::vector` в качестве массива C (например, для совместимости с библиотеками C). Это возможно, потому что элементы в векторе хранятся смежно.

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data();
```

В отличие от решений, основанных на предыдущих стандартах C ++ (см. Ниже), функция-член `.data()` также может применяться к пустым векторам, поскольку в этом случае она не вызывает неопределенного поведения.

Перед C ++ 11 вы берете адрес первого элемента вектора для получения эквивалентного указателя, если вектор не пуст, эти оба метода взаимозаменяемы:

```
int* p = &v[0];           // combine subscript operator and 0 literal

int* p = &v.front();      // explicitly reference the first element
```

Примечание. Если вектор пуст, `v[0]` и `v.front()` не определены и не могут быть использованы.

При сохранении базового адреса данных вектора обратите внимание, что многие операции (такие как `push_back`, `resize` и т. Д.) Могут изменить местоположение памяти данных

вектора, тем самым **аннулируя предыдущие указатели данных** . Например:

```
std::vector<int> v;  
int* p = v.data();  
v.resize(42);           // internal memory location changed; value of p is now invalid
```

Итератор / указатель Invalidation

Итераторы и указатели, указывающие на `std::vector` могут стать недействительными, но только при выполнении определенных операций. Использование недействительных итераторов / указателей приведет к неопределенному поведению.

Операции, которые делают недействительными итераторы / указатели, включают:

- Любая операция вставки, которая изменяет `capacity` `vector` , аннулирует *все* итераторы / указатели:

```
vector<int> v(5); // Vector has a size of 5; capacity is unknown.  
int *p1 = &v[0];  
v.push_back(2);   // p1 may have been invalidated, since the capacity was unknown.  
  
v.reserve(20);    // Capacity is now at least 20.  
int *p2 = &v[0];  
v.push_back(4);   // p2 is *not* invalidated, since the size of `v` is now 7.  
v.insert(v.end(), 30, 9); // Inserts 30 elements at the end. The size exceeds the  
                          // requested capacity of 20, so `p2` is (probably) invalidated.  
int *p3 = &v[0];  
v.reserve(v.capacity() + 20); // Capacity exceeded, thus `p3` is invalid.
```

C ++ 11

```
auto old_cap = v.capacity();  
v.shrink_to_fit();  
if(old_cap != v.capacity())  
    // Iterators were invalidated.
```

- Любая операция вставки, которая не увеличивает емкость, все равно приведет к недействительности итераторов / указателей, указывающих на элементы в позиции вставки и мимо нее. Это включает в себя `end` итератор:

```
vector<int> v(5);  
v.reserve(20);           // Capacity is at least 20.  
int *p1 = &v[0];  
int *p2 = &v[3];  
v.insert(v.begin() + 2, 5, 0); // `p2` is invalidated, but since the capacity  
                               // did not change, `p1` remains valid.  
int *p3 = &v[v.size() - 1];  
v.push_back(10); // The capacity did not change, so `p3` and `p1` remain valid.
```

- Любая операция удаления приведет к недействительности итераторов / указателей, указывающих на удаленные элементы и на любые элементы, прошедшие удаленные

элементы. Это включает в себя `end` итератор:

```
vector<int> v(10);
int *p1 = &v[0];
int *p2 = &v[5];
v.erase(v.begin() + 3, v.end()); // `p2` is invalid, but `p1` remains valid.
```

- `operator=` (копирование, перемещение или иное) и `clear()` приведет к аннулированию всех итераторов / указателей, указывающих на вектор.

Удаление элементов

Удаление последнего элемента:

```
std::vector<int> v{ 1, 2, 3 };
v.pop_back(); // v becomes {1, 2}
```

Удаление всех элементов:

```
std::vector<int> v{ 1, 2, 3 };
v.clear(); // v becomes an empty vector
```

Удаление элемента по индексу:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 3); // v becomes {1, 2, 3, 5, 6}
```

Примечание. Для `vector` удаляющего элемент, который не является последним элементом, все элементы за удаленным элементом должны быть скопированы или перемещены, чтобы заполнить пробел, см. Примечание ниже и [std :: list](#).

Удаление всех элементов в диапазоне:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 1, v.begin() + 5); // v becomes {1, 6}
```

Примечание. Вышеуказанные методы не изменяют емкость вектора, а только размер. См. [Размер и емкость файла](#).

Метод `erase`, который удаляет ряд элементов, часто используется как часть идиомы [стирания-удаления](#). То есть, сначала `std::remove` перемещает некоторые элементы в

конец вектора, а затем `erase` их. Это относительно неэффективная операция для любых индексов, меньших последнего индекса вектора, потому что все элементы после стертых сегментов должны быть перемещены в новые позиции. Для критически важных приложений, требующих эффективного удаления произвольных элементов в контейнере, см. [Std :: list](#).

Удаление элементов по значению:

```
std::vector<int> v{ 1, 1, 2, 2, 3, 3 };
int value_to_remove = 2;
v.erase(std::remove(v.begin(), v.end(), value_to_remove), v.end()); // v becomes {1, 1, 3, 3}
```

Удаление элементов по условию:

```
// std::remove_if needs a function, that takes a vector element as argument and returns true,
// if the element shall be removed
bool _predicate(const int& element) {
    return (element > 3); // This will cause all elements to be deleted that are larger than 3
}
...
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(), _predicate), v.end()); // v becomes {1, 2, 3}
```

Удаление элементов с помощью лямбда, не создавая дополнительной функции предиката

C++ 11

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(),
    [](auto& element){return element > 3;} ), v.end()
);
```

Удаление элементов по условию из цикла:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
std::vector<int>::iterator it = v.begin();
while (it != v.end()) {
    if (condition)
        it = v.erase(it); // after erasing, 'it' will be set to the next element in v
}
```

```

else
    ++it;           // manually set 'it' to the next element in v
}

```

Хотя важно *не* увеличивать `it` в случае удаления, вам следует рассмотреть возможность использования другого метода при повторном стирании в цикле. Рассмотрим `remove_if` для более эффективного способа.

Удаление элементов по условию из обратной петли:

```

std::vector<int> v{ -1, 0, 1, 2, 3, 4, 5, 6 };
typedef std::vector<int>::reverse_iterator rev_itr;
rev_itr it = v.rbegin();

while (it != v.rend()) { // after the loop only '0' will be in v
    int value = *it;
    if (value) {
        ++it;
        // See explanation below for the following line.
        it = rev_itr(v.erase(it.base()));
    } else
        ++it;
}

```

Обратите внимание на некоторые моменты для предыдущего цикла:

- Учитывая обратный итератор `it`, указывая на какой-то элемент, метод `base` дает регулярный (не обратный) итератор, указывающий на тот же элемент.
- `vector::erase(iterator)` стирает элемент, на который указывает итератор, и возвращает итератор элементу, который следует за данным элементом.
- `reverse_iterator::reverse_iterator(iterator)` строит обратный итератор с итератора.

Помещенный в целом, линия `it = rev_itr(v.erase(it.base()))` говорит: взять реверсивный итератор `it`, есть `v`. Сотрите элемент, указав его регулярной итератором; принимает полученный итератор, построить обратный итератор от него, и назначить его на обратный итератор `it`.

Удаление всех элементов с помощью `v.clear()` не освобождает память (`capacity()` вектора остается неизменной). Чтобы освободить место, используйте:

```

std::vector<int>().swap(v);

```

`shrink_to_fit()` освобождает неиспользованную емкость вектора:

```
v.shrink_to_fit();
```

Функция `shrink_to_fit` не гарантирует освобождения пространства, но большинство современных реализаций.

Поиск элемента в `std::vector`

Функция `std::find`, определенная в заголовке `<algorithm>`, может использоваться для поиска элемента в `std::vector`.

`std::find` использует `operator==` для сравнения элементов для равенства. Он возвращает итератор в первый элемент в диапазоне, который сравнивается с значением.

Если этот элемент не найден, `std::find` возвращает `std::vector::end` (или `std::vector::cend` если вектор `const`).

C++ 11

```
static const int arr[] = {5, 4, 3, 2, 1};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find(v.begin(), v.end(), 4);
std::vector<int>::difference_type index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

std::vector<int>::iterator missing = std::find(v.begin(), v.end(), 10);
std::vector<int>::difference_type index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

C++ 11

```
std::vector<int> v { 5, 4, 3, 2, 1 };

auto it = std::find(v.begin(), v.end(), 4);
auto index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

auto missing = std::find(v.begin(), v.end(), 10);
auto index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

Если вам нужно выполнить много запросов в большом векторе, вы можете сначала рассмотреть сортировку вектора, прежде чем использовать алгоритм `binary_search`.

Чтобы найти первый элемент в векторе, который удовлетворяет условию, можно использовать `std::find_if`. В дополнение к двум параметрам, заданным для `std::find`, `std::find_if` принимает третий аргумент, который является объектом функции или указателем функции на функцию предиката. Предикат должен принять элемент из

контейнера в качестве аргумента и вернуть значение, конвертируемое в `bool` , без изменения контейнера:

C ++ 11

```
bool isEven(int val) {
    return (val % 2 == 0);
}

struct moreThan {
    moreThan(int limit) : _limit(limit) {}

    bool operator()(int val) {
        return val > _limit;
    }

    int _limit;
};

static const int arr[] = {1, 3, 7, 8};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find_if(v.begin(), v.end(), isEven);
// `it` points to 8, the first even element

std::vector<int>::iterator missing = std::find_if(v.begin(), v.end(), moreThan(10));
// `missing` is v.end(), as no element is greater than 10
```

C ++ 11

```
// find the first value that is even
std::vector<int> v = {1, 3, 7, 8};
auto it = std::find_if(v.begin(), v.end(), [](int val){return val % 2 == 0;});
// `it` points to 8, the first even element

auto missing = std::find_if(v.begin(), v.end(), [](int val){return val > 10;});
// `missing` is v.end(), as no element is greater than 10
```

Преобразование массива в `std::vector`

Массив можно легко преобразовать в `std::vector` , используя `std::begin` и `std::end` :

C ++ 11

```
int values[5] = { 1, 2, 3, 4, 5 }; // source array

std::vector<int> v(std::begin(values), std::end(values)); // copy array to new vector

for(auto &x: v)
    std::cout << x << " ";
std::cout << std::endl;
```

1 2 3 4 5

```
int main(int argc, char* argv[]) {
```



```
// convert main arguments into a vector of strings.
std::vector<std::string> args(argv, argv + argc);
}
```

С инициализатор_класса C ++ 11 может также использоваться для инициализации вектора сразу

```
initializer_list<int> arr = { 1,2,3,4,5 };
vector<int> vec1 {arr};

for (auto & i : vec1)
    cout << i << endl;
```

вектор : Исключение для многих, так много правил

Стандарт (раздел 23.3.7) указывает, что предоставляется специализация `vector<bool>`, которая оптимизирует пространство, упаковывая значения `bool`, так что каждый занимает только один бит. Поскольку биты не адресуются в C ++, это означает, что на `vector<bool>` не помещаются несколько требований к `vector`:

- Сохраненные данные не обязательно должны быть смежными, поэтому `vector<bool>` не может быть передан в API C, который ожидает массив `bool`.
- `at()`, `operator []`, а разыменование итераторов не возвращает ссылку на `bool`. Скорее они возвращают прокси-объект, который (несовершенно) имитирует ссылку на `bool`, перегружая его операторы присваивания. В качестве примера следующий код может быть недействительным для `std::vector<bool>`, поскольку разыменование итератора не возвращает ссылку:

C ++ 11

```
std::vector<bool> v = {true, false};
for (auto &b: v) { } // error
```

Аналогично, функции, ожидающие `bool&` аргумент, не могут использоваться с результатом `operator []` или `at()` примененного к `vector<bool>`, или с результатом разыменования его итератора:

```
void f(bool& b);
f(v[0]);           // error
f(*v.begin());     // error
```

Реализация `std::vector<bool>` зависит как от компилятора, так и от архитектуры.

Специализация реализуется путем упаковки `n` Booleans в младший адресный раздел памяти. Здесь `n` - это размер в битах самой младшей адресуемой памяти. В большинстве современных систем это 1 байт или 8 бит. Это означает, что один байт может хранить 8 булевых значений. Это улучшение по сравнению с традиционной реализацией, где 1 булево значение хранится в 1 байт памяти.

Примечание. В приведенном ниже примере показаны возможные побитовые значения отдельных байтов в традиционном и оптимизированном `vector<bool>`. Это не всегда верно для всех архитектур. Это, однако, хороший способ визуализации оптимизации. В приведенных ниже примерах байт представлен как `[x, x, x, x, x, x, x, x]`.

Традиционный `std::vector<char>` *Сохранение 8 Булевых значений:*

C ++ 11

```
std::vector<char> trad_vect = {true, false, false, false, true, false, true, true};
```

Поразрядное представление:

```
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0],  
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,1]
```

Специализированный `std::vector<bool>` *Сохранение 8 Булевы значения:*

C ++ 11

```
std::vector<bool> optimized_vect = {true, false, false, false, true, false, true, true};
```

Поразрядное представление:

```
[1,0,0,0,1,0,1,1]
```

Обратите внимание, что в традиционной версии `std::vector<bool>` 8 булевых значений занимают 8 байт памяти, тогда как в оптимизированной версии `std::vector<bool>` они используют только 1 байт объем памяти. Это значительное улучшение использования памяти. Если вам нужно передать `vector<bool>` в API стиля C, вам может потребоваться скопировать значения в массив или найти лучший способ использования API, если память и производительность подвержены риску.

Размер и емкость вектора

Размер вектора - это просто количество элементов в векторе:

1. Текущий **размер** вектора запрашивается функцией `size()`. Функция `empty()` возвращает `true` если размер равен 0:

```
vector<int> v = { 1, 2, 3 }; // size is 3  
const vector<int>::size_type size = v.size();  
cout << size << endl; // prints 3  
cout << boolalpha << v.empty() << endl; // prints false
```

2. По умолчанию сконструированный вектор начинается с размера 0:

```
vector<int> v; // size is 0
cout << v.size() << endl; // prints 0
```

3. Добавление N элементов к вектору увеличивает **размер** на N (например, с помощью функций `push_back()` , `insert()` или `resize()`).
4. Удаление N элементов из вектора уменьшает **размер** на N (например, с помощью `pop_back()` , `pop_back()` `erase()` или `clear()`).
5. Вектор имеет верхний предел для реализации по размеру, но вы, скорее всего, исчерпаете ОЗУ до его достижения:

```
vector<int> v;
const vector<int>::size_type max_size = v.max_size();
cout << max_size << endl; // prints some large number
v.resize( max_size ); // probably won't work
v.push_back( 1 ); // definitely won't work
```

Общая ошибка: **размер** не обязательно (или даже обычно) `int` :

```
// !!!bad!!!evil!!!
vector<int> v_bad( N, 1 ); // constructs large N size vector
for( int i = 0; i < v_bad.size(); ++i ) { // size is not supposed to be int!
    do_something( v_bad[i] );
}
```

Векторная емкость отличается от **размера** . В то время как **размер** - это просто количество элементов, которые вектор имеет в настоящее время, **емкость** для количества элементов, на которые он выделил / зарезервировал память. Это полезно, потому что слишком частые (пере) выделения слишком больших размеров могут быть дорогими.

1. Текущая **емкость** вектора запрашивается функцией функции `capacity()` . **Емкость** всегда больше или равна **размеру** :

```
vector<int> v = { 1, 2, 3 }; // size is 3, capacity is >= 3
const vector<int>::size_type capacity = v.capacity();
cout << capacity << endl; // prints number >= 3
```

2. Вы можете вручную зарезервировать емкость по `reserve( N )` функции (она меняет векторную емкость на N):

```
// !!!bad!!!evil!!!
vector<int> v_bad;
for( int i = 0; i < 10000; ++i ) {
    v_bad.push_back( i ); // possibly lot of reallocations
}

// good
vector<int> v_good;
v_good.reserve( 10000 ); // good! only one allocation
```

```
for( int i = 0; i < 10000; ++i ) {
    v_good.push_back( i ); // no allocations needed anymore
}
```

3. Вы можете запросить избыточную емкость, которая будет выпущена с помощью `shrink_to_fit()` (но реализация не должна вас подчиняться). Это полезно для сохранения использованной памяти:

```
vector<int> v = { 1, 2, 3, 4, 5 }; // size is 5, assume capacity is 6
v.shrink_to_fit(); // capacity is 5 (or possibly still 6)
cout << boolalpha << v.capacity() == v.size() << endl; // prints likely true (but
possibly false)
```

Вектор частично управляет емкостью автоматически, когда вы добавляете элементы, которые он может решить, чтобы расти. Исполнители любят использовать 2 или 1,5 для фактора роста (золотое соотношение будет идеальным значением, но нецелесообразным из-за рационального числа). С другой стороны, вектор обычно не сжимается автоматически. Например:

```
vector<int> v; // capacity is possibly (but not guaranteed) to be 0
v.push_back( 1 ); // capacity is some starter value, likely 1
v.clear(); // size is 0 but capacity is still same as before!

v = { 1, 2, 3, 4 }; // size is 4, and lets assume capacity is 4.
v.push_back( 5 ); // capacity grows - let's assume it grows to 6 (1.5 factor)
v.push_back( 6 ); // no change in capacity
v.push_back( 7 ); // capacity grows - let's assume it grows to 9 (1.5 factor)
// and so on
v.pop_back(); v.pop_back(); v.pop_back(); v.pop_back(); // capacity stays the same
```

Конкатенационные векторы

Один `std::vector` может быть добавлен к другому с помощью функции-члена `insert()` :

```
std::vector<int> a = {0, 1, 2, 3, 4};
std::vector<int> b = {5, 6, 7, 8, 9};

a.insert(a.end(), b.begin(), b.end());
```

Однако это решение выходит из строя, если вы пытаетесь добавить вектор к себе, потому что стандарт указывает, что итераторы, данные для `insert()` не должны быть того же диапазона, что и элементы объекта получателя.

C++ 11

Вместо использования функций-членов вектора можно использовать функции `std::begin()` и `std::end()` :

```
a.insert(std::end(a), std::begin(b), std::end(b));
```

Это более общее решение, например, поскольку `b` также может быть массивом. Однако это решение не позволяет вам добавлять вектор к себе.

Если порядок элементов в принимающем векторе не имеет значения, учитывая количество элементов в каждом векторе, можно избежать ненужных операций копирования:

```
if (b.size() < a.size())
    a.insert(a.end(), b.begin(), b.end());
else
    b.insert(b.end(), a.begin(), a.end());
```

Уменьшение пропускной способности вектора

`std::vector` автоматически увеличивает свою емкость при вставке по мере необходимости, но никогда не уменьшает ее емкость после удаления элемента.

```
// Initialize a vector with 100 elements
std::vector<int> v(100);

// The vector's capacity is always at least as large as its size
auto const old_capacity = v.capacity();
// old_capacity >= 100

// Remove half of the elements
v.erase(v.begin() + 50, v.end()); // Reduces the size from 100 to 50 (v.size() == 50),
// but not the capacity (v.capacity() == old_capacity)
```

Чтобы уменьшить его емкость, мы можем скопировать содержимое вектора в новый временный вектор. Новый вектор будет иметь минимальную емкость, необходимую для хранения всех элементов исходного вектора. Если уменьшение размера исходного вектора было значительным, то уменьшение емкости для нового вектора, вероятно, будет значительным. Затем мы можем поменять исходный вектор на временный, чтобы сохранить его минимальную емкость:

```
std::vector<int> (v).swap(v);
```

C++ 11

В C++ 11 мы можем использовать `shrink_to_fit()` члена `shrink_to_fit()` для аналогичного эффекта:

```
v.shrink_to_fit();
```

Примечание. Функция `shrink_to_fit()` является запросом и не гарантирует уменьшение емкости.

Использование сортированного вектора для быстрого поиска элементов

Заголовок `<algorithm>` предоставляет ряд полезных функций для работы со отсортированными векторами.

Важным предварительным условием для работы с отсортированными векторами является то, что сохраненные значения сравнимы с `<`.

Несортированный вектор можно отсортировать с помощью функции `std::sort()` :

```
std::vector<int> v;
// add some code here to fill v with some elements
std::sort(v.begin(), v.end());
```

Сортированные векторы позволяют эффективно искать элементы, используя функцию `std::lower_bound()`. В отличие от `std::find()`, он выполняет эффективный двоичный поиск на векторе. Недостатком является то, что он дает только действительные результаты для отсортированных диапазонов ввода:

```
// search the vector for the first element with value 42
std::vector<int>::iterator it = std::lower_bound(v.begin(), v.end(), 42);
if (it != v.end() && *it == 42) {
    // we found the element!
}
```

Примечание. Если запрошенное значение не является частью вектора, `std::lower_bound()` вернет итератор в первый элемент, который *больше* запрашиваемого значения. Такое поведение позволяет нам вставить новый элемент в нужное место в уже отсортированном векторе:

```
int const new_element = 33;
v.insert(std::lower_bound(v.begin(), v.end(), new_element), new_element);
```

Если вам нужно вставить сразу несколько элементов, возможно, более эффективно сначала вызвать `push_back()` для всех них, а затем вызвать `std::sort()` после того, как будут вставлены все элементы. В этом случае увеличенная стоимость сортировки может окупиться сниженной стоимостью вставки новых элементов в конце вектора, а не в середине.

Если ваш вектор содержит несколько элементов одного значения, `std::lower_bound()` попытается вернуть итератор первому элементу `std::lower_bound()` значения. Однако, если вам нужно вставить новый элемент *после* последнего элемента `std::upper_bound()` значения, вы должны использовать функцию `std::upper_bound()` поскольку это приведет к меньшему смещению элементов:

```
v.insert(std::upper_bound(v.begin(), v.end(), new_element), new_element);
```

Если вам нужны итераторы верхней и нижней границ, вы можете использовать функцию `std::equal_range()` для эффективного извлечения обоих из них одним вызовом:

```
std::pair<std::vector<int>::iterator,  
        std::vector<int>::iterator> rg = std::equal_range(v.begin(), v.end(), 42);  
std::vector<int>::iterator lower_bound = rg.first;  
std::vector<int>::iterator upper_bound = rg.second;
```

Чтобы проверить, существует ли элемент в отсортированном векторе (хотя это и не относится к векторам), вы можете использовать функцию `std::binary_search()` :

```
bool exists = std::binary_search(v.begin(), v.end(), value_to_find);
```

Функции, возвращающие большие векторы

C ++ 11

В C ++ 11 компиляторы должны неявно перемещаться из локальной возвращаемой переменной. Более того, большинство компиляторов во многих случаях могут выполнять **копирование** и исключать переход вообще. В результате этого возвращение больших объектов, которые могут быть перемещены дешево, больше не требует специальной обработки:

```
#include <vector>  
#include <iostream>  
  
// If the compiler is unable to perform named return value optimization (NRVO)  
// and elide the move altogether, it is required to move from v into the return value.  
std::vector<int> fillVector(int a, int b) {  
    std::vector<int> v;  
    v.reserve(b-a+1);  
    for (int i = a; i <= b; i++) {  
        v.push_back(i);  
    }  
    return v; // implicit move  
}  
  
int main() { // declare and fill vector  
    std::vector<int> vec = fillVector(1, 10);  
  
    // print vector  
    for (auto value : vec)  
        std::cout << value << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "  
  
    std::cout << std::endl;  
  
    return 0;  
}
```

C ++ 11

Перед C ++ 11, копирование elision было уже разрешено и реализовано большинством компиляторов. Однако из-за отсутствия семантики перемещения в устаревшем коде или коде, который должен быть скомпилирован со старыми версиями компилятора, которые не реализуют эту оптимизацию, вы можете найти векторы, передаваемые в качестве

выходных аргументов, чтобы предотвратить ненужную копию:

```
#include <vector>
#include <iostream>

// passing a std::vector by reference
void fillVectorFrom_By_Ref(int a, int b, std::vector<int> &v) {
    assert(v.empty());
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
}

int main() { // declare vector
    std::vector<int> vec;

    // fill vector
    fillVectorFrom_By_Ref(1, 10, vec);
    // print vector
    for (std::vector<int>::const_iterator it = vec.begin(); it != vec.end(); ++it)
        std::cout << *it << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "
    std::cout << std::endl;
    return 0;
}
```

Найти максимальный и минимальный элемент и соответствующий индекс в векторе

Чтобы найти самый большой или самый маленький элемент, хранящийся в векторе, вы можете использовать методы `std::max_element` и `std::min_element` соответственно. Эти методы определены в заголовке `<algorithm>`. Если несколько элементов эквивалентны наибольшему (наименьшему) элементу, методы возвращают итератор в первый такой элемент. Вернуть `v.end()` для пустых векторов.

```
std::vector<int> v = {5, 2, 8, 10, 9};
int maxElementIndex = std::max_element(v.begin(), v.end()) - v.begin();
int maxElement = *std::max_element(v.begin(), v.end());

int minElementIndex = std::min_element(v.begin(), v.end()) - v.begin();
int minElement = *std::min_element(v.begin(), v.end());

std::cout << "maxElementIndex:" << maxElementIndex << ", maxElement:" << maxElement << '\n';
std::cout << "minElementIndex:" << minElementIndex << ", minElement:" << minElement << '\n';
```

Выход:

```
maxElementIndex: 3, maxElement: 10
minElementIndex: 1, minElement: 2
```

C++ 11

Минимальный и максимальный элемент в векторе можно получить одновременно,

используя метод `std::minmax_element` , который также определен в заголовке `<algorithm>` :

```
std::vector<int> v = {5, 2, 8, 10, 9};
auto minmax = std::minmax_element(v.begin(), v.end());

std::cout << "minimum element: " << *minmax.first << '\n';
std::cout << "maximum element: " << *minmax.second << '\n';
```

Выход:

минимальный элемент: 2
максимальный элемент: 10

Матрицы с использованием векторов

Векторы могут использоваться в качестве 2D-матрицы, определяя их как вектор векторов.

Матрица с 3 строками и 4 столбцами с каждой ячейкой, инициализированной как 0, может быть определена как:

```
std::vector<std::vector<int> > matrix(3, std::vector<int>(4));
```

C ++ 11

Синтаксис для их инициализации с использованием списков инициализации или иначе аналогичен синтаксису нормального вектора.

```
std::vector<std::vector<int>> matrix = { {0,1,2,3},
                                         {4,5,6,7},
                                         {8,9,10,11}
};
```

Значения в таком векторе могут быть доступны аналогично двумерному массиву

```
int var = matrix[0][2];
```

Итерация по всей матрице аналогична и для нормального вектора, но с дополнительной размерностью.

```
for(int i = 0; i < 3; ++i)
{
    for(int j = 0; j < 4; ++j)
    {
        std::cout << matrix[i][j] << std::endl;
    }
}
```

C ++ 11

```
for(auto& row: matrix)
```

```
{
    for(auto& col : row)
    {
        std::cout << col << std::endl;
    }
}
```

Вектор векторов - удобный способ представления матрицы, но это не самый эффективный: отдельные векторы разбросаны по памяти, а структура данных не кэширована.

Кроме того, в соответствующей матрице длина каждой строки должна быть одинаковой (это не относится к вектору векторов). Дополнительная гибкость может быть источником ошибок.

Прочитайте стандарт :: вектор онлайн: <https://riptutorial.com/ru/cplusplus/topic/511/станд-вектор>

глава 113: станд :: Карта

замечания

- Чтобы использовать любой из `std::map` или `std::multimap` должен быть включен заголовочный файл `<map>` .
- `std::map` и `std::multimap` обе сохраняют свои элементы отсортированными в соответствии с возрастающим порядком ключей. В случае `std::multimap` сортировка не производится для значений одного и того же ключа.
- Основное различие между `std::map` и `std::multimap` заключается в том, что `std::map` не позволяет дублировать значения для того же ключа, где `std::multimap` .
- Карты реализованы как двоичные деревья поиска. Таким образом, `search()` , `insert()` , `erase()` занимает среднее время ($\log n$). Для работы с постоянным временем используйте `std::unordered_map` .
- Функции `size()` и `empty()` имеют временную сложность $\Theta(1)$, количество узлов кэшируется, чтобы избежать ходьбы по дереву каждый раз, когда вызывается эти функции.

Examples

Доступ к элементам

`std::map` принимает (key, value) пары в качестве входных данных.

Рассмотрим следующий пример инициализации `std::map` :

```
std::map < std::string, int > ranking { std::make_pair("stackoverflow", 2),  
                                       std::make_pair("docs-beta", 1) };
```

На `std::map` элементы можно вставить следующим образом:

```
ranking["stackoverflow"]=2;  
ranking["docs-beta"]=1;
```

В приведенном выше примере, если ключевой `stackoverflow` уже присутствует, его значение будет обновлено до 2. Если он еще не присутствует, будет создана новая запись.

На `std::map` элементы можно получить напрямую, указав ключ как индекс:

```
std::cout << ranking[ "stackoverflow" ] << std::endl;
```

Обратите внимание, что использование `operator[]` на карте фактически *введет новое значение* с запрошенным ключом в карту. Это означает, что вы не можете использовать его на `const std::map`, даже если ключ уже сохранен на карте. Чтобы предотвратить эту установку, проверьте, существует ли элемент (например, с помощью `find()`) или используйте его `at()` как описано ниже.

C++ 11

К элементам `std::map` можно обращаться с помощью `at()`:

```
std::cout << ranking.at("stackoverflow") << std::endl;
```

Обратите внимание, что `at()` будет `std::out_of_range` исключение `std::out_of_range` если контейнер не содержит запрошенный элемент.

В обоих контейнерах `std::map` и `std::multimap`, элементы могут быть доступны с помощью итераторов:

C++ 11

```
// Example using begin()
std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                       std::make_pair(1, "docs-beta"),
                                       std::make_pair(2, "stackexchange") };

auto it = mmp.begin();
std::cout << it->first << " : " << it->second << std::endl; // Output: "1 : docs-beta"
it++;
std::cout << it->first << " : " << it->second << std::endl; // Output: "2 : stackoverflow"
it++;
std::cout << it->first << " : " << it->second << std::endl; // Output: "2 : stackexchange"

// Example using rbegin()
std::map < int, std::string > mp { std::make_pair(2, "stackoverflow"),
                                std::make_pair(1, "docs-beta"),
                                std::make_pair(2, "stackexchange") };

auto it2 = mp.rbegin();
std::cout << it2->first << " : " << it2->second << std::endl; // Output: "2 : stackoverflow"
it2++;
std::cout << it2->first << " : " << it2->second << std::endl; // Output: "1 : docs-beta"
```

Инициализация `std::map` или `std::multimap`

`std::map` и `std::multimap` оба могут быть инициализированы путем предоставления пар ключ-значение, разделенных запятой. Ключи-значения могут быть предоставлены либо `{key, value}` либо могут быть явно созданы с помощью `std::make_pair(key, value)`. Поскольку `std::map` не позволяет дублировать ключи, а оператор запятой выполняет справа налево, пара справа будет перезаписана парой с одним и тем же ключом слева.

```
std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                       std::make_pair(1, "docs-beta"),
                                       std::make_pair(2, "stackexchange") };
```

```
// 1 docs-beta
// 2 stackoverflow
// 2 stackexchange

std::map < int, std::string > mp {  std::make_pair(2, "stackoverflow"),
                                std::make_pair(1, "docs-beta"),
                                std::make_pair(2, "stackexchange")  };

// 1 docs-beta
// 2 stackoverflow
```

Оба могут быть инициализированы итератором.

```
// From std::map or std::multimap iterator
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {6, 8}, {3, 4},
                               {6, 7} };
                               // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 8}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); //moved cursor on first {6, 5}
std::map< int, int > mp(it, mmp.end()); // {6, 5}, {8, 9}

//From std::pair array
std::pair< int, int > arr[10];
arr[0] = {1, 3};
arr[1] = {1, 5};
arr[2] = {2, 5};
arr[3] = {0, 1};
std::map< int, int > mp(arr,arr+4); //{0 , 1}, {1, 3}, {2, 5}

//From std::vector of std::pair
std::vector< std::pair<int, int> > v{ {1, 5}, {5, 1}, {3, 6}, {3, 2} };
std::multimap< int, int > mp(v.begin(), v.end());
                               // {1, 5}, {3, 6}, {3, 2}, {5, 1}
```

Удаление элементов

Удаление всех элементов:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
mmp.clear(); //empty multimap
```

Удаление элемента из где-то с помощью итератора:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                               // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); // moved cursor on first {6, 5}
mmp.erase(it); // {1, 2}, {3, 4}, {3, 4}, {6, 7}, {8, 9}
```

Удаление всех элементов в диапазоне:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                               // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
auto it2 = it;
```

```
it++; //moved first cursor on first {3, 4}
std::advance(it2,3); //moved second cursor on first {6, 5}
mmp.erase(it,it2); // {1, 2}, {6, 5}, {6, 7}, {8, 9}
```

Удаление всех элементов, имеющих заданное значение в качестве ключа:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
// {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
mmp.erase(6); // {1, 2}, {3, 4}, {3, 4}, {8, 9}
```

Удаление элементов, которые удовлетворяют предикату `pred` :

```
std::map<int,int> m;
auto it = m.begin();
while (it != m.end())
{
    if (pred(*it))
        it = m.erase(it);
    else
        ++it;
}
```

Вставка элементов

Элемент может быть вставлен в `std::map` только если его ключ еще не присутствует на карте. Дано, например:

```
std::map< std::string, size_t > fruits_count;
```

- Пара ключ-значение вставляется в `std::map` через функцию-член `insert()` . Для этого требуется `pair` в качестве аргумента:

```
fruits_count.insert({"grapes", 20});
fruits_count.insert(make_pair("orange", 30));
fruits_count.insert(pair<std::string, size_t>("banana", 40));
fruits_count.insert(map<std::string, size_t>::value_type("cherry", 50));
```

Функция `insert()` возвращает `pair` состоящую из итератора и значения `bool` :

- Если вставка прошла успешно, итератор указывает на вновь вставленный элемент, а значение `bool` равно `true` .
- Если уже был элемент с одним и тем же `key` , вставка не выполняется. Когда это происходит, итератор указывает на элемент, вызывающий конфликт, а значение `bool` равно `false` .

Для комбинирования операций вставки и поиска можно использовать следующий метод:

```
auto success = fruits_count.insert({"grapes", 20});
```

```
if (!success.second) {           // we already have 'grapes' in the map
    success.first->second += 20; // access the iterator to update the value
}
```

- Для удобства контейнер `std::map` предоставляет оператору индексирования доступ к элементам на карте и вставлять новые, если они не существуют:

```
fruits_count["apple"] = 10;
```

Хотя это проще, он не позволяет пользователю проверить, существует ли элемент уже. Если элемент отсутствует, `std::map::operator[]` неявно создает его, инициализируя его конструктором по умолчанию, прежде чем перезаписывать его с указанным значением.

- `insert()` может быть использована для добавления нескольких элементов одновременно с использованием скопированного списка пар. Эта версия `insert()` возвращает `void`:

```
fruits_count.insert({{"apricot", 1}, {"jackfruit", 1}, {"lime", 1}, {"mango", 7}});
```

- `insert()` также может использоваться для добавления элементов с помощью итераторов, обозначающих значения начала и конца `value_type`:

```
std::map< std::string, size_t > fruit_list{ {"lemon", 0}, {"olive", 0}, {"plum", 0}};
fruits_count.insert(fruit_list.begin(), fruit_list.end());
```

Пример:

```
std::map<std::string, size_t> fruits_count;
std::string fruit;
while(std::cin >> fruit){
    // insert an element with 'fruit' as key and '1' as value
    // (if the key is already stored in fruits_count, insert does nothing)
    auto ret = fruits_count.insert({fruit, 1});
    if(!ret.second){           // 'fruit' is already in the map
        ++ret.first->second;    // increment the counter
    }
}
```

Сложность времени для операции вставки - $O(\log n)$, поскольку `std::map` реализованы как деревья.

C++ 11

`pair` может быть построена явно с использованием `make_pair()` и `emplace()`:

```
std::map< std::string , int > runs;
runs.emplace("Babe Ruth", 714);
runs.insert(make_pair("Barry Bonds", 762));
```

Если мы знаем, где будет вставлен новый элемент, мы можем использовать `emplace_hint()` чтобы указать `hint` итератора. Если новый элемент можно вставить непосредственно перед `hint`, то вставка может быть выполнена в постоянное время. В противном случае он ведет себя так же, как `emplace()`:

```
std::map< std::string , int > runs;
auto it = runs.emplace("Barry Bonds", 762); // get iterator to the inserted element
// the next element will be before "Barry Bonds", so it is inserted before 'it'
runs.emplace_hint(it, "Babe Ruth", 714);
```

Итерация по `std::map` или `std::multimap`

`std::map` или `std::multimap` может быть пройден следующими способами:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };

//Range based loop - since C++11
for(const auto &x: mmp)
    std::cout<< x.first << ":"<< x.second << std::endl;

//Forward iterator for loop: it would loop through first element to last element
//it will be a std::map< int, int >::iterator
for (auto it = mmp.begin(); it != mmp.end(); ++it)
    std::cout<< it->first << ":"<< it->second << std::endl; //Do something with iterator

//Backward iterator for loop: it would loop through last element to first element
//it will be a std::map< int, int >::reverse_iterator
for (auto it = mmp.rbegin(); it != mmp.rend(); ++it)
    std::cout<< it->first << " " << it->second << std::endl; //Do something with iterator
```

При повторении по `std::map` или `std::multimap` использование `auto` предпочтительнее, чтобы избежать бесполезных неявных преобразований (подробнее см. [Этот ответ SO](#)).

Поиск в `std::map` или в `std::multimap`

Существует несколько способов поиска ключа в `std::map` или в `std::multimap`.

- Чтобы получить итератор первого вхождения ключа, можно использовать функцию `find()`. Он возвращает `end()` если ключ не существует.

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
auto it = mmp.find(6);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl; //prints: 6, 5
else
    std::cout << "Value does not exist!" << std::endl;

it = mmp.find(66);
if(it!=mmp.end())
    std::cout << it->first << ", " << it->second << std::endl;
else
    std::cout << "Value does not exist!" << std::endl; // This line would be executed.
```


- Другой способ определить, существует ли запись в `std::map` или в `std::multimap`, используется функция `count()`, которая подсчитывает, сколько значений связано с данным ключом. Поскольку `std::map` связывает только одно значение с каждым ключом, функция `count()` может возвращать только 0 (если ключ отсутствует) или 1 (если есть). Для `std::multimap` `count()` может возвращать значения, превышающие 1, поскольку может быть несколько значений, связанных с одним и тем же ключом.

```
std::map< int , int > mp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
if(mp.count(3) > 0) // 3 exists as a key in map
    std::cout << "The key exists!" << std::endl; // This line would be executed.
else
    std::cout << "The key does not exist!" << std::endl;
```

Если вам все равно, существует ли какой-то элемент, `find` будет строго лучше: он документирует ваше намерение, и для `multimaps` он может остановиться после обнаружения первого совпадающего элемента.

- В случае `std::multimap` может быть несколько элементов, имеющих один и тот же ключ. Чтобы получить этот диапазон, используется `equal_range()` которая возвращает `std::pair` имеющую нижнюю границу итератора (включительно) и верхнюю границу (исключая) соответственно. Если ключ не существует, оба итератора будут указывать на `end()`.

```
auto egr = mmp.equal_range(6);
auto st = egr.first, en = egr.second;
for(auto it = st; it != en; ++it){
    std::cout << it->first << ", " << it->second << std::endl;
}

// prints: 6, 5
//         6, 7
```

Проверка количества элементов

Контейнер `std::map` имеет функцию-член `empty()`, которая возвращает `true` или `false`, в зависимости от того, пуста или нет карта. Элемент функции `size()` возвращает число элементов, хранящихся в `std::map`:

```
std::map<std::string , int> rank {{"facebook.com", 1}, {"google.com", 2}, {"youtube.com", 3}};
if(!rank.empty()){
    std::cout << "Number of elements in the rank map: " << rank.size() << std::endl;
}
else{
    std::cout << "The rank map is empty" << std::endl;
}
```

Типы карт

Регулярная карта

Карта представляет собой ассоциативный контейнер, содержащий пары ключ-значение.

```
#include <string>
#include <map>
std::map<std::string, size_t> fruits_count;
```

В приведенном выше примере `std::string` - это тип *ключа*, а `size_t` - *значение*.

Ключ действует как индекс на карте. Каждый ключ должен быть уникальным и должен быть заказан.

- Если вам нужны multiple элементы с одним и тем же ключом, рассмотрите возможность использования `multimap` (объяснено ниже)
- Если ваш тип значения не указывает какой-либо заказ или вы хотите переопределить порядок по умолчанию, вы можете указать его:

```
#include <string>
#include <map>
#include <cstring>
struct StrLess {
    bool operator()(const std::string& a, const std::string& b) {
        return strncmp(a.c_str(), b.c_str(), 8)<0;
        //compare only up to 8 first characters
    }
}
std::map<std::string, size_t, StrLess> fruits_count2;
```

Если компаратор `StrLess` возвращает `false` для двух ключей, они считаются одинаковыми, даже если их фактическое содержимое отличается.

Multi-Map

`Multimap` позволяет хранить несколько пар ключ-значение с одним и тем же ключом на карте. В противном случае его интерфейс и создание очень похожи на обычную карту.

```
#include <string>
#include <map>
std::multimap<std::string, size_t> fruits_count;
std::multimap<std::string, size_t, StrLess> fruits_count2;
```

Хэш-карта (неупорядоченная карта)

В хэш-карте хранятся пары ключ-значение, аналогичные обычной карте. Однако он не упорядочивает элементы по отношению к ключу. Вместо этого *хэш*-значение для ключа используется для быстрого доступа к необходимым парам ключ-значение.

```
#include <string>
```

```
#include <unordered_map>
std::unordered_map<std::string, size_t> fruits_count;
```

Неупорядоченные карты обычно быстрее, но элементы не сохраняются в каком-либо предсказуемом порядке. Например, итерация по всем элементам в `unordered_map` дает элементы в кажущемся случайном порядке.

Создание `std::map` с определяемыми пользователем типами как ключ

Чтобы иметь возможность использовать класс в качестве ключа на карте, все, что требуется от ключа, состоит в том, что оно может быть `copiable` и `assignable`. Порядок в карте определяется третьим аргументом шаблона (и аргументом для конструктора, если он используется). По умолчанию используется `std::less<KeyType>`, по умолчанию используется оператор `<`, но нет необходимости использовать значения по умолчанию. Просто напишите оператор сравнения (желательно как функциональный объект):

```
struct CmpMyType
{
    bool operator()( MyType const& lhs, MyType const& rhs ) const
    {
        // ...
    }
};
```

В C++ предикат «сравнить» должен быть **строгим слабым порядком**. В частности, `compare(X, X)` должно возвращать `false` для любого `X` т.е. если `CmpMyType()(a, b)` возвращает `true`, то `CmpMyType()(b, a)` должен возвращать значение `false`, а если оба возвращают `false`, то элементы считаются равными (членами одного и того же класса эквивалентности).

Строгий слабый порядок

Это математический термин для определения отношения между двумя объектами. Его определение:

Два объекта `x` и `y` эквивалентны, если оба `f(x, y)` и `f(y, x)` ложны. Заметим, что объект всегда (по инварианту неререфлексивности) эквивалентен самому себе.

В терминах C++ это означает, что если у вас есть два объекта данного типа, вы должны вернуть следующие значения по сравнению с оператором `<`.

```
X    a;
X    b;
```

Condition:	Test:	Result
a is equivalent to b:	a < b	false
a is equivalent to b	b < a	false

a is less than b	a < b	true
a is less than b	b < a	false
b is less than a	a < b	false
b is less than a	b < a	true

То, как вы определяете эквивалент / меньше, полностью зависит от типа вашего объекта.

Прочитайте стандарт :: Карта онлайн: <https://riptutorial.com/ru/cplusplus/topic/681/станд----карта>

глава 114: станд :: любой

замечания

Класс `std::any` предоставляет контейнер с типом безопасности, в который мы можем поместить одиночные значения любого типа.

Examples

Основное использование

```
std::any an_object{ std::string("hello world") };
if (an_object.has_value()) {
    std::cout << std::any_cast<std::string>(an_object) << '\n';
}

try {
    std::any_cast<int>(an_object);
} catch (std::bad_any_cast&) {
    std::cout << "Wrong type\n";
}

std::any_cast<std::string&>(an_object) = "42";
std::cout << std::any_cast<std::string>(an_object) << '\n';
```

Выход

```
hello world
Wrong type
42
```

Прочитайте станд :: любой онлайн: <https://riptutorial.com/ru/cplusplus/topic/7894/станд----любой>

глава 115: станд :: массив

параметры

параметр	Определение
class T	Задаёт тип данных элементов массива
std::size_t N	Указывает количество элементов в массиве

замечания

Использование `std::array` требует включения заголовка `<array>` с помощью `#include <array>`.

Examples

Инициализация `std::array`

Инициализация `std::array<T, N>`, где `T` - скалярный тип, а `N` - число элементов типа `T`

Если `T` является скалярным типом, `std::array` может быть инициализирован следующими способами:

```
// 1) Using aggregate-initialization
std::array<int, 3> a{ 0, 1, 2 };
// or equivalently
std::array<int, 3> a = { 0, 1, 2 };

// 2) Using the copy constructor
std::array<int, 3> a{ 0, 1, 2 };
std::array<int, 3> a2(a);
// or equivalently
std::array<int, 3> a2 = a;

// 3) Using the move constructor
std::array<int, 3> a = std::array<int, 3>{ 0, 1, 2 };
```

Инициализация `std::array<T, N>`, где `T` - не скалярный тип, а `N` - число элементов типа `T`

Если `T` - не скалярный тип, `std::array` может быть инициализирован следующими способами:

```
struct A { int values[3]; }; // An aggregate type

// 1) Using aggregate initialization with brace elision
// It works only if T is an aggregate type!
std::array<A, 2> a{ 0, 1, 2, 3, 4, 5 };
```

```
// or equivalently
std::array<A, 2> a = { 0, 1, 2, 3, 4, 5 };

// 2) Using aggregate initialization with brace initialization of sub-elements
std::array<A, 2> a{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };
// or equivalently
std::array<A, 2> a = { A{ 0, 1, 2 }, A{ 3, 4, 5 } };

// 3)
std::array<A, 2> a{{ { 0, 1, 2 }, { 3, 4, 5 } }};
// or equivalently
std::array<A, 2> a = {{ { 0, 1, 2 }, { 3, 4, 5 } }};

// 4) Using the copy constructor
std::array<A, 2> a{ 1, 2, 3 };
std::array<A, 2> a2(a);
// or equivalently
std::array<A, 2> a2 = a;

// 5) Using the move constructor
std::array<A, 2> a = std::array<A, 2>{ 0, 1, 2, 3, 4, 5 };
```

Доступ к элементу

1. at (pos)

Возвращает ссылку на элемент в позиции `pos` с проверкой границ. Если `pos` не находится в пределах контейнера, `std::out_of_range` исключение типа `std::out_of_range`.

Сложность постоянна $O(1)$.

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr.at(0) = 2;
    arr.at(1) = 4;
    arr.at(2) = 6;

    // read values
    int a = arr.at(0); // a is now 2
    int b = arr.at(1); // b is now 4
    int c = arr.at(2); // c is now 6

    return 0;
}
```

2) operator[pos]

Возвращает ссылку на элемент в позиции `pos` без проверки границ. Если `pos` не находится в пределах контейнера, может произойти ошибка *нарушения сегментации* во время выполнения. Этот метод обеспечивает доступ к элементу, эквивалентный классическим

массивам и более эффективному, чем `at(pos)` .

Сложность постоянна $O(1)$.

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr[0] = 2;
    arr[1] = 4;
    arr[2] = 6;

    // read values
    int a = arr[0]; // a is now 2
    int b = arr[1]; // b is now 4
    int c = arr[2]; // c is now 6

    return 0;
}
```

3) `std::get<pos>`

Эта функция **не являющийся членом** возвращает ссылку на элемент во **время компиляции постоянной** позиции `pos` без проверки границ. Если `pos` не находится в пределах контейнера, может произойти ошибка *нарушения сегментации* во время выполнения.

Сложность постоянна $O(1)$.

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    std::get<0>(arr) = 2;
    std::get<1>(arr) = 4;
    std::get<2>(arr) = 6;

    // read values
    int a = std::get<0>(arr); // a is now 2
    int b = std::get<1>(arr); // b is now 4
    int c = std::get<2>(arr); // c is now 6

    return 0;
}
```

4) `front()`

Возвращает ссылку на первый элемент в контейнере. Вызов `front()` на пустой контейнер не определен.

Сложность постоянна $O(1)$.

Примечание. Для контейнера с выражение `c.front()` эквивалентно `*c.begin()` .

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.front(); // a is now 2

    return 0;
}
```

5) `back()`

Возвращает ссылку на последний элемент в контейнере. Вызов `back()` на пустой контейнер не определен.

Сложность постоянна $O(1)$.

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.back(); // a is now 6

    return 0;
}
```

6) `data()`

Возвращает указатель на базовый массив, служащий в качестве хранилища элементов. Указатель таков, что `range [data(); data() + size())` всегда является допустимым диапазоном, даже если контейнер пуст (`data()` не вызывает различий в этом случае).

Сложность постоянна $O(1)$.

```
#include <iostream>
#include <cstring>
#include <array>

int main ()
{
    const char* cstr = "Test string";
    std::array<char, 12> arr;

    std::memcpy(arr.data(), cstr, 12); // copy cstr to arr

    std::cout << arr.data(); // outputs: Test string
}
```

```
    return 0;
}
```

Проверка размера массива

Одним из основных преимуществ `std::array` по сравнению с массивом стиля `c` является то, что мы можем проверить размер массива с помощью функции члена `size()`

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    cout << arr.size() << endl;
}
```

Итерирование через массив

`std::array` являющийся контейнером STL, может использовать диапазон для цикла, аналогичный другим контейнерам, таким как `vector`

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    for (auto i : arr)
        cout << i << '\n';
}
```

Одновременное изменение всех элементов массива

Функция member `fill()` может использоваться в `std::array` для изменения значений сразу после инициализации

```
int main() {

    std::array<int, 3> arr = { 1, 2, 3 };
    // change all elements of the array to 100
    arr.fill(100);

}
```

Прочитайте стандарт :: массив онлайн: <https://riptutorial.com/ru/cplusplus/topic/2712/станд-массив>

глава 116: станд :: опционально

Examples

Вступление

Опционы (также известные как типы Maybe) используются для представления типа, содержимое которого может быть или не быть. Они реализованы на C ++ 17 в качестве `std::optional`. Например, объект типа `std::optional<int>` может содержать некоторое значение типа `int` или не может содержать никакого значения.

Опционы обычно используются либо для представления значения, которое может не существовать, либо как возвращаемого типа из функции, которая не может вернуть значимый результат.

Другие подходы к выбору

Есть много других подходов к решению проблемы, что `std::optional` решает, но ни один из них не является полным: использование указателя, использование часового, или использование `pair<bool, T>`.

Необязательный против указателя

В некоторых случаях мы можем предоставить указатель на существующий объект или `nullptr` для указания отказа. Но это ограничивается теми случаями, когда объекты уже существуют - `optional`, как тип значения, также может использоваться для возврата новых объектов, не прибегая к распределению памяти.

Необязательный против Sentinel

Общей идиомой является использование специального значения, указывающего на то, что значение не имеет смысла. Это может быть 0 или -1 для целых типов, или `nullptr` для указателей. Тем не менее, это уменьшает пространство допустимых значений (вы не можете отличить действительное значение 0 от бессмысленного 0), и многие типы не имеют естественного выбора для дозорного значения.

Необязательный vs `std::pair<bool, T>`

Другой распространенной идиомой является предоставление пары, где один из элементов - это `bool` указывающий, имеет ли значение значение.

Это зависит от типа значения, являющегося конструктивным по умолчанию в случае

ошибки, что невозможно для некоторых типов и возможно, но нежелательно для других. `optional<T>` , в случае ошибки, не нужно ничего строить.

Использование опций для представления отсутствия значения

До C ++ 17, имея указатели со значением `nullptr` обычно представляло отсутствие значения. Это хорошее решение для больших объектов, которые были динамически распределены и уже управляются указателями. Однако это решение не работает хорошо для небольших или примитивных типов, таких как `int` , которые редко когда-либо динамически распределяются или управляются указателями. `std::optional` обеспечивает жизнеспособное решение этой общей проблемы.

В этом примере определяется `struct Person` . Человек может иметь домашнее животное, но не обязательно. Таким образом, `pet` член `Person` объявлен с `std::optional` оберткой.

```
#include <iostream>
#include <optional>
#include <string>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::optional<Animal> pet;
};

int main() {
    Person person;
    person.name = "John";

    if (person.pet) {
        std::cout << person.name << "'s pet's name is " <<
            person.pet->name << std::endl;
    }
    else {
        std::cout << person.name << " is alone." << std::endl;
    }
}
```

Использование опций для представления отказа функции

До C ++ 17 функция обычно представляла отказ одним из нескольких способов:

- Был возвращен нулевой указатель.
 - например, вызов функции `Delegate *App::get_delegate()` в экземпляре `App` , у которого не было делегата, вернет `nullptr` .
 - Это хорошее решение для объектов, которые были динамически распределены или являются большими и управляются указателями, но не являются хорошим решением для небольших объектов, которые обычно распределяются по стеклу

и передаются путем копирования.

- Определенное значение типа возврата зарезервировано для указания отказа.
 - например, вызов функции `unsigned shortest_path_distance(Vertex a, Vertex b)` на две вершины, которые не связаны, может возвращать ноль, чтобы указать этот факт.
- Значение было сопряжено вместе с `bool` чтобы указать, что возвращаемое значение имеет смысл.
 - например, вызов функции `std::pair<int, bool> parse(const std::string &str)` со строковым аргументом, который не является целым числом, возвратит пару с неопределенным `int` и `bool` установленным в `false`.

В этом примере Джону даются два питомца: «Пушистый» и «Furball». Затем

`Person::pet_with_name()` функция `Person::pet_with_name()` для извлечения любимых

`Person::pet_with_name()` Джона. Поскольку у John нет домашнего животного с именем

`Whiskers`, функция не работает, и вместо этого возвращается `std::nullopt`.

```
#include <iostream>
#include <optional>
#include <string>
#include <vector>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::vector<Animal> pets;

    std::optional<Animal> pet_with_name(const std::string &name) {
        for (const Animal &pet : pets) {
            if (pet.name == name) {
                return pet;
            }
        }
        return std::nullopt;
    }
};

int main() {
    Person john;
    john.name = "John";

    Animal fluffy;
    fluffy.name = "Fluffy";
    john.pets.push_back(fluffy);

    Animal furball;
    furball.name = "Furball";
    john.pets.push_back(furball);

    std::optional<Animal> whiskers = john.pet_with_name("Whiskers");
    if (whiskers) {
        std::cout << "John has a pet named Whiskers." << std::endl;
    }
}
```

```

else {
    std::cout << "Whiskers must not belong to John." << std::endl;
}
}

```

необязательно как возвращаемое значение

```

std::optional<float> divide(float a, float b) {
    if (b!=0.f) return a/b;
    return {};
}

```

Здесь мы возвращаем либо дробь a/b , но если она не определена (была бы бесконечной), мы вместо этого возвращаем пустой необязательный.

Более сложный случай:

```

template<class Range, class Pred>
auto find_if( Range&& r, Pred&& p ) {
    using std::begin; using std::end;
    auto b = begin(r), e = end(r);
    auto r = std::find_if(b, e, p );
    using iterator = decltype(r);
    if (r==e)
        return std::optional<iterator>();
    return std::optional<iterator>(r);
}

template<class Range, class T>
auto find( Range&& r, T const& t ) {
    return find_if( std::forward<Range>(r), [&t](auto&& x){return x==t;} );
}

```

`find( some_range, 7 )` ищет контейнер или диапазон `some_range` для чего-то равного числу 7 .
`find_if` делает это с предикатом.

Он возвращает либо пустой необязательный, если он не был найден, либо необязательный, содержащий итератор для элемента, если он был.

Это позволяет:

```

if (find( vec, 7 )) {
    // code
}

```

или даже

```

if (auto oit = find( vec, 7 )) {
    vec.erase(*oit);
}

```

без необходимости возиться с итераторами начала и окончания.

value_or

```
void print_name( std::ostream& os, std::optional<std::string> const& name ) {  
    std::cout << "Name is: " << name.value_or("<name missing>") << '\n';  
}
```

`value_or` либо возвращает значение, хранящееся в необязательном, либо аргумент, если там ничего нет.

Это позволяет вам взять необязательный необязательный параметр и дать поведение по умолчанию, когда вам действительно нужно значение. Делая это таким образом, решение «поведение по умолчанию» может быть отброшено до точки, где это лучше всего сделать и немедленно необходимо, вместо того, чтобы генерировать значение по умолчанию в глубине некоторого движка.

Прочитайте стандарт :: опционально онлайн: <https://riptutorial.com/ru/cplusplus/topic/2423/станд-опционально>

глава 117: станд :: пара

Examples

Создание пары и доступ к элементам

Пара позволяет нам рассматривать два объекта как один объект. Пары можно легко построить с помощью функции шаблона `std::make_pair`.

Альтернативным способом является создание пары и назначение ее элементов (`first` и `second`) позже.

```
#include <iostream>
#include <utility>

int main()
{
    std::pair<int,int> p = std::make_pair(1,2); //Creating the pair
    std::cout << p.first << " " << p.second << std::endl; //Accessing the elements

    //We can also create a pair and assign the elements later
    std::pair<int,int> p1;
    p1.first = 3;
    p1.second = 4;
    std::cout << p1.first << " " << p1.second << std::endl;

    //We can also create a pair using a constructor
    std::pair<int,int> p2 = std::pair<int,int>(5, 6);
    std::cout << p2.first << " " << p2.second << std::endl;

    return 0;
}
```

Сравнить операторы

Параметры этих операторов: `lhs` и `rhs`

- `operator==` если оба элемента на парах `lhs` и `rhs` равны. Возвращаемое значение `true` если оба `lhs.first == rhs.first AND lhs.second == rhs.second` , в противном случае `false`

```
std::pair<int, int> p1 = std::make_pair(1, 2);
std::pair<int, int> p2 = std::make_pair(2, 2);

if (p1 == p2)
    std::cout << "equals";
else
    std::cout << "not equal"//statement will show this, because they are not identical
```


- `operator!=` тесты, если любые элементы на парах `lhs` и `rhs` не равны. Возвращаемое значение `true` если либо `lhs.first != rhs.first` OR `lhs.second != rhs.second`, в противном случае возвращает `false`.
- `operator<` тесты, если `lhs.first < rhs.first`, возвращает `true`. В противном случае, если `rhs.first < lhs.first` возвращает `false`. В противном случае, если `lhs.second < rhs.second` возвращает `true`, в противном случае возвращает `false`.
- `operator<=` возвращает `!(rhs < lhs)`
- `operator>` возвращает `rhs < lhs`
- `operator>=` returns `!(lhs < rhs)`

Другой пример с контейнерами пар. Он использует `operator<` потому что ему нужно сортировать контейнер.

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include <string>

int main()
{
    std::vector<std::pair<int, std::string>> v = { {2, "baz"},
                                                  {2, "bar"},
                                                  {1, "foo"} };

    std::sort(v.begin(), v.end());

    for(const auto& p: v) {
        std::cout << "(" << p.first << "," << p.second << " ";
        //output: (1,foo) (2,bar) (2,baz)
    }
}
```

Прочитайте стандарт :: пара онлайн: <https://riptutorial.com/ru/cplusplus/topic/4834/станд----пара>

глава 118: станд :: строка

Вступление

Строки - это объекты, которые представляют последовательности символов. Стандартный класс `string` обеспечивает простую, безопасную и универсальную альтернативу использованию явных массивов `char` при работе с текстом и другими последовательностями символов. `string` класс C++ является частью пространства имен `std` и стандартизован в 1998 году.

Синтаксис

- *// Пустое объявление строки*

```
std :: string s;
```

- *// Построение из const char * (c-string)*

```
std :: string s ("Hello");
```

```
std :: string s = "Hello";
```

- *// Построение с использованием конструктора копирования*

```
std :: string s1 ("Hello");
```

```
std :: string s2 (s1);
```

- *// Построение из подстроки*

```
std :: string s1 ("Hello");
```

```
std :: string s2 (s1, 0, 4); // Копировать 4 символа из позиции 0 из s1 в s2
```

- *// Построение из буфера символов*

```
std :: string s1 («Hello World»);
```

```
std :: string s2 (s1, 5); // Скопируем первые 5 символов s1 в s2
```

- *// Создаем конструктор заполнения (только char)*

```
std :: string s (5, 'a'); // s содержит ааааа
```

- *// Конструкция с использованием конструктора диапазонов и итератора*

```
std :: string s1 («Hello World»);
```

```
std :: string s2 (s1.begin (), s1.begin () + 5); // Скопируем первые 5 символов s1 в s2
```

замечания

Прежде чем использовать `std::string`, вы должны включить `string` заголовка, так как она включает в себя функции / операторы / перегрузки, которые не включают другие заголовки (например, `iostream`).

Использование конструктора `const char *` с `nullptr` приводит к неопределенному поведению.

```
std::string oops(nullptr);  
std::cout << oops << "\n";
```

Метод `at` `std::out_of_range` если `index >= size()`.

Поведение `operator[]` немного сложнее, во всех случаях оно имеет неопределенное поведение, если `index > size()`, но когда `index == size()`:

C ++ 11

1. В строке non-const поведение не *определено*;
2. В строке `CharT()` ссылка на символ со значением `CharT()` (*нулевой символ*).

C ++ 11

1. `CharT()` ссылка на символ со значением `CharT()` (*нулевой символ*).
2. Изменение этой ссылки - *неопределенное поведение*.

Начиная с C ++ 14 вместо использования `"foo"` рекомендуется использовать `"foo"s`, так как `s` - это **определяемый пользователем буквенный суффикс**, который преобразует `const char* "foo"` в `std::string "foo"`,

Примечание: Вы должны использовать пространство имен `std::string_literals` или `std::literals`, чтобы получить буквенные `s`.

Examples

расщепляющий

Используйте `std::string::substr` чтобы разделить строку. Существует два варианта этой функции-члена.

Первый принимает *начальную позицию*, из которой должна начинаться возвращаемая подстрока. Начальная позиция должна быть действительной в диапазоне `(0, str.length())`

:

```
std::string str = "Hello foo, bar and world!";  
std::string newstr = str.substr(11); // "bar and world!"
```

Второй занимает начальную позицию и общую *длину* новой подстроки. Независимо от *длины* , подстрока никогда не пройдет мимо конца строки источника:

```
std::string str = "Hello foo, bar and world!";  
std::string newstr = str.substr(15, 3); // "and"
```

Обратите внимание, что вы также можете вызывать `substr` без аргументов, в этом случае возвращается точная копия строки

```
std::string str = "Hello foo, bar and world!";  
std::string newstr = str.substr(); // "Hello foo, bar and world!"
```

Сменная замена

Заменить по положению

Чтобы заменить часть `std::string` вы можете использовать метод `replace` из `std::string` .

`replace` есть много полезных перегрузок:

```
//Define string  
std::string str = "Hello foo, bar and world!";  
std::string alternate = "Hello foobar";  
  
//1)  
str.replace(6, 3, "bar"); //"Hello bar, bar and world!"  
  
//2)  
str.replace(str.begin() + 6, str.end(), "nobody!"); //"Hello nobody!"  
  
//3)  
str.replace(19, 5, alternate, 6, 6); //"Hello foo, bar and foobar!"
```

C ++ 14

```
//4)  
str.replace(19, 5, alternate, 6); //"Hello foo, bar and foobar!"
```

```
//5)  
str.replace(str.begin(), str.begin() + 5, str.begin() + 6, str.begin() + 9);  
//"foo foo, bar and world!"
```

```
//6)  
str.replace(0, 5, 3, 'z'); //"zzz foo, bar and world!"
```

```
//7)
str.replace(str.begin() + 6, str.begin() + 9, 3, 'x'); // "Hello xxx, bar and world!"
```

C++ 11

```
//8)
str.replace(str.begin(), str.begin() + 5, { 'x', 'y', 'z' }); // "xyz foo, bar and world!"
```

Заменить вхождения строки другой строкой

Заменить только первое вхождение replace C with B str :

```
std::string replaceString(std::string str,
                          const std::string& replace,
                          const std::string& with){
    std::size_t pos = str.find(replace);
    if (pos != std::string::npos)
        str.replace(pos, replace.length(), with);
    return str;
}
```

Заменить все случаи replace C with на str :

```
std::string replaceStringAll(std::string str,
                             const std::string& replace,
                             const std::string& with) {
    if(!replace.empty()) {
        std::size_t pos = 0;
        while ((pos = str.find(replace, pos)) != std::string::npos) {
            str.replace(pos, replace.length(), with);
            pos += with.length();
        }
    }
    return str;
}
```

конкатенация

Вы можете объединить std::string S с помощью перегруженных операторов + и += .
Использование оператора + :

```
std::string hello = "Hello";
std::string world = "world";
std::string helloworld = hello + world; // "Helloworld"
```

Используя оператор += :

```
std::string hello = "Hello";
```

```
std::string world = "world";  
hello += world; // "Helloworld"
```

Вы также можете добавить строки C, включая строковые литералы:

```
std::string hello = "Hello";  
std::string world = "world";  
const char *comma = ", ";  
std::string newhelloworld = hello + comma + world + "!"; // "Hello, world!"
```

Вы также можете использовать `push_back()` для возврата отдельных `char` :

```
std::string s = "a, b, ";  
s.push_back('c'); // "a, b, c"
```

Существует также `append()` , что в значительной степени напоминает `+=` :

```
std::string app = "test and ";  
app.append("test"); // "test and test"
```

Доступ к персонажу

Существует несколько способов извлечения символов из `std::string` и каждый из них отличается от других.

```
std::string str("Hello world!");
```

Оператор `[]` (п)

Возвращает ссылку на символ с индексом `n`.

`std::string::operator[]` не проверяется границами и не генерирует исключение. Вызывающий отвечает за утверждение, что индекс находится в пределах строки:

```
char c = str[6]; // 'w'
```

В (п)

Возвращает ссылку на символ с индексом `n`.

`std::string::at` *is bounds checked* и будет `std::out_of_range` если индекс не находится в пределах строки:

```
char c = str.at(7); // 'o'
```

Примечание. Оба эти примера приведут к [неопределенному поведению](#), если строка пуста.

фронт()

Возвращает ссылку на первый символ:

```
char c = str.front(); // 'H'
```

назад ()

Возвращает ссылку на последний символ:

```
char c = str.back(); // 'I'
```

токенизировать

Перечислены от наименее дорогостоящих до самых дорогих во время выполнения:

1. `str::strtok` - самый дешевый стандарт, предоставляемый метод токенизации, он также позволяет изменять разделитель между токенами, но он несет 3 трудности с современным C++:
 - `std::strtok` нельзя использовать одновременно для нескольких `strings` (хотя некоторые реализации расширяются для поддержки этого, например: [strtok_s](#))
 - По той же причине `std::strtok` не может использоваться одновременно для нескольких потоков (это может быть определено как реализация, например: реализация [Visual Studio является потокобезопасной](#))
 - Вызов `std::strtok` изменяет `std::string` он работает, поэтому он не может использоваться для `const string S`, `const char* S` или `literal string`, чтобы токенизировать любой из них с помощью `std::strtok` или работать с `std::string`, содержимое которого необходимо сохранить, нужно будет скопировать его, затем копировать можно

Как правило, любая из этих опционных затрат будет скрыта в стоимости распределения токенов, но если требуется самый дешевый алгоритм, и трудности `std::strtok` не могут быть преодолены, рассмотрите решение с [ручным](#) сжатием.

```
// String to tokenize
std::string str{ "The quick brown fox" };
// Vector to store tokens
vector<std::string> tokens;

for (auto i = strtok(&str[0], " "); i != NULL; i = strtok(NULL, " "))
    tokens.push_back(i);
```

Живой пример

2. `std::istream_iterator` использует оператор извлечения потока итеративно. Если входная `std::string` ограничена пробелом, то она может расширяться по опции `std::strtok`, устраняя ее трудности, позволяя встроенную токенизацию, тем самым поддерживая генерацию `const vector<string>` и добавляя поддержку нескольких Обозначение белого пробела:

```
// String to tokenize
const std::string str("The quick \tbrown \nfox");
std::istringstream is(str);
// Vector to store tokens
const std::vector<std::string> tokens = std::vector<std::string>(
    std::istream_iterator<std::string>(is),
    std::istream_iterator<std::string>());
```

Живой пример

3. `std::regex_token_iterator` использует `std::regex` для итеративного tokenize. Он обеспечивает более гибкое определение разделителя. Например, не разделенные запятыми и пробелы:

C ++ 11

```
// String to tokenize
const std::string str{ "The ,qu\, ick ,\tbrown, fox" };
const std::regex re{ "\\s*((?:[^\s\\,]|\\\\\\\\.)*?)\\s*(?:,|$)" };
// Vector to store tokens
const std::vector<std::string> tokens{
    std::sregex_token_iterator(str.begin(), str.end(), re, 1),
    std::sregex_token_iterator()
};
```

Живой пример

Для `regex_token_iterator` дополнительной информации см. [Пример regex_token_iterator](#).

Преобразование в (const) char *

Чтобы получить `const char*` доступ к данным `std::string` вы можете использовать функцию члена `c_str()`. Имейте в виду, что указатель действителен только до тех пор, пока объект `std::string` находится в пределах области действия и остается неизменным, это означает, что на объект могут быть вызваны только методы `const`.

C ++ 17

Функция члена `data()` может использоваться для получения модифицируемого `char*`, который может использоваться для управления данными объекта `std::string`.

C ++ 11

Модифицируемый `char*` также можно получить, взяв адрес первого символа: `&s[0]`. Внутри C++ 11 гарантируется, что это приведет к созданию корректной строки с нулевым завершением. Заметим, что `&s[0]` корректно сформирован, даже если `s` пуст, а `&s.front()` не определено, если `s` пусто.

C++ 11

```
std::string str("This is a string.");
const char* cstr = str.c_str(); // cstr points to: "This is a string.\0"
const char* data = str.data();  // data points to: "This is a string.\0"
```

```
std::string str("This is a string.");

// Copy the contents of str to untie lifetime from the std::string object
std::unique_ptr<char []> cstr = std::make_unique<char[]>(str.size() + 1);

// Alternative to the line above (no exception safety):
// char* cstr_unsafe = new char[str.size() + 1];

std::copy(str.data(), str.data() + str.size(), cstr);
cstr[str.size()] = '\0'; // A null-terminator needs to be added

// delete[] cstr_unsafe;
std::cout << cstr.get();
```

Поиск символа (ов) в строке

Чтобы найти символ или другую строку, вы можете использовать `std::string::find`. Он возвращает позицию первого символа первого совпадения. Если совпадений не найдено, функция возвращает `std::string::npos`

```
std::string str = "Curiosity killed the cat";
auto it = str.find("cat");

if (it != std::string::npos)
    std::cout << "Found at position: " << it << '\n';
else
    std::cout << "Not found!\n";
```

Найдено в позиции: 21

Возможности поиска дополнительно расширяются следующими функциями:

```
find_first_of      // Find first occurrence of characters
find_first_not_of  // Find first absence of characters
find_last_of       // Find last occurrence of characters
find_last_not_of   // Find last absence of characters
```

Эти функции позволяют вам искать символы с конца строки, а также находить отрицательный случай (т. Е. Символы, которые не находятся в строке). Вот пример:

```
std::string str = "dog dog cat cat";
std::cout << "Found at position: " << str.find_last_of("gzx") << '\n';
```

Найдено в позиции: 6

Примечание. Помните, что вышеупомянутые функции не выполняют поиск подстрок, а скорее для символов, содержащихся в строке поиска. В этом случае последнее вхождение 'g' было найдено в позиции 6 (другие символы не были найдены).

Обрезка символов при запуске / окончании

Этот пример требует заголовков `<algorithm>`, `<locale>` и `<utility>`.

C++ 11

Чтобы *обрезать* последовательность или строку, необходимо удалить все ведущие и конечные элементы (или символы), соответствующие определенному предикату. Сначала мы обрезаем конечные элементы, потому что они не связаны с перемещением каких-либо элементов, а затем обрезают ведущие элементы. Обратите внимание, что приведенные ниже обобщения работают для всех типов `std::basic_string` (например, `std::string` и `std::wstring`), а также для контейнеров последовательностей (например, `std::vector` и `std::list`).

```
template <typename Sequence, // any basic_string, vector, list etc.
          typename Pred>     // a predicate on the element (character) type
Sequence& trim(Sequence& seq, Pred pred) {
    return trim_start(trim_end(seq, pred), pred);
}
```

Обрезка конечных элементов включает поиск *последнего* элемента, не соответствующего предикату, и стирание оттуда:

```
template <typename Sequence, typename Pred>
Sequence& trim_end(Sequence& seq, Pred pred) {
    auto last = std::find_if_not(seq.rbegin(),
                                seq.rend(),
                                pred);
    seq.erase(last.base(), seq.end());
    return seq;
}
```

Обрезка ведущих элементов включает в себя поиск *первого* элемента, не соответствующего предикату, и стирание до него:

```
template <typename Sequence, typename Pred>
Sequence& trim_start(Sequence& seq, Pred pred) {
    auto first = std::find_if_not(seq.begin(),
                                  seq.end(),
                                  pred);
    seq.erase(seq.begin(), first);
}
```

```
    return seq;
}
```

Чтобы специализировать выше для обрезки пробелов в `std::string` мы можем использовать функцию `std::isspace()` как предикат:

```
std::string& trim(std::string& str, const std::locale& loc = std::locale()) {
    return trim(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_start(std::string& str, const std::locale& loc = std::locale()) {
    return trim_start(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_end(std::string& str, const std::locale& loc = std::locale()) {
    return trim_end(str, [&loc](const char c){ return std::isspace(c, loc); });
}
```

Аналогично, мы можем использовать функцию `std::iswspace()` для `std::wstring` и т. Д.

Если вы хотите создать *новую* последовательность, которая является обрезанной копией, вы можете использовать отдельную функцию:

```
template <typename Sequence, typename Pred>
Sequence trim_copy(Sequence seq, Pred pred) { // NOTE: passing seq by value
    trim(seq, pred);
    return seq;
}
```

Лексикографическое сравнение

Две `std::string` s можно сравнить лексикографически с помощью операторов `==`, `!=`, `<`, `<=`, `>` и `>=`:

```
std::string str1 = "Foo";
std::string str2 = "Bar";

assert(!(str1 < str2));
assert(str > str2);
assert(!(str1 <= str2));
assert(str1 >= str2);
assert(!(str1 == str2));
assert(str1 != str2);
```

Все эти функции используют метод `std::string::compare()` для выполнения сравнения и возвращают для удобства логические значения. Работа этих функций может быть интерпретирована следующим образом, независимо от фактической реализации:

- оператор `==` :

Если `str1.length() == str2.length()` и каждая пара символов совпадают, то возвращает

`true` , иначе возвращает `false` .

- оператор `!=` :

Если `str1.length() != str2.length()` или одна пара символов не соответствует, возвращает `true` , в противном случае возвращает `false` .

- оператор `<` или оператор `>` :

Находит первую пару символов, сравнивает их, а затем возвращает логический результат.

- оператор `<=` или оператор `>=` :

Находит первую пару символов, сравнивает их, а затем возвращает логический результат.

Примечание . Пара символьная пара означает соответствующие символы в обеих строках одинаковых позиций. Для лучшего понимания, если две строки примера - `str1` и `str2` , а их длины - `n` и `m` соответственно, тогда пары символов обеих строк означают каждую пару `str1[i]` и `str2[i]` где $i = 0, 1, 2 \dots \max(n, m)$. Если для любого i , где соответствующий символ не существует, т. Е. Когда i больше или равно `n` или `m` , он будет считаться самым низким значением.

Ниже приведен пример использования `<` :

```
std::string str1 = "Barr";
std::string str2 = "Bar";

assert(str2 < str1);
```

Эти шаги заключаются в следующем:

1. Сравните первые символы, `'B' == 'B'` - двигаться дальше.
2. Сравните второй символ, `'a' == 'a'` - двигаться дальше.
3. Сравните третий символы, `'r' == 'r'` - перейдите.
4. Диапазон `str2` теперь исчерпан, а в `str1` все еще есть символы. Таким образом, `str2 < str1` .

Преобразование в `std::wstring`

В C++ последовательности символов представлены специализированием `std::basic_string` с нативным типом символов. Две основные коллекции, определенные стандартной библиотекой, - `std::string` и `std::wstring` :

- `std::string` построена с элементами типа `char`

- `std::wstring` построена с элементами типа `wchar_t`

Чтобы преобразовать между двумя типами, используйте `wstring_convert` :

```
#include <string>
#include <codecvt>
#include <locale>

std::string input_str = "this is a -string-, which is a sequence based on the -char- type.";
std::wstring input_wstr = L"this is a -wide- string, which is based on the -wchar_t- type.";

// conversion
std::wstring str_turned_to_wstr =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes(input_str);

std::string wstr_turned_to_str =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().to_bytes(input_wstr);
```

Чтобы улучшить удобство использования и / или читаемость, вы можете определить функции для выполнения преобразования:

```
#include <string>
#include <codecvt>
#include <locale>

using convert_t = std::codecvt_utf8<wchar_t>;
std::wstring_convert<convert_t, wchar_t> strconverter;

std::string to_string(std::wstring wstr)
{
    return strconverter.to_bytes(wstr);
}

std::wstring to_wstring(std::string str)
{
    return strconverter.from_bytes(str);
}
```

Использование образца:

```
std::wstring a_wide_string = to_wstring("Hello World!");
```

Это, безусловно, более `std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes("Hello World!")` чем `std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes("Hello World!")` .

Обратите внимание: `char` и `wchar_t` не подразумевают кодировку и не указывают размер в байтах. Например, `wchar_t` обычно реализуется как 2-байтовый тип данных и обычно содержит кодированные данные UTF-16 под Windows (или UCS-2 в версиях до Windows 2000) и как 4-байтовый тип данных, закодированный с использованием UTF-32 в Linux. Это контрастирует с более новыми типами `char16_t` и `char32_t` , которые были введены в C ++ 11 и гарантированно будут достаточно большими, чтобы содержать любой символ UTF16 или

UTF32 (или, точнее, *кодировую точку*) соответственно.

Использование класса `std::string_view`

C++ 17

В C++ 17 представлен `std::string_view`, который является просто не принадлежащим ему диапазоном `const char S`, реализуемым как пара указателей или указатель и длина. Это отличный тип параметров для функций, которые требуют немодифицируемых строковых данных. До C++ 17 для этого было три варианта:

```
void foo(std::string const& s);           // pre-C++17, single argument, could incur
                                         // allocation if caller's data was not in a string
                                         // (e.g. string literal or vector<char> )

void foo(const char* s, size_t len);     // pre-C++17, two arguments, have to pass them
                                         // both everywhere

void foo(const char* s);                 // pre-C++17, single argument, but need to call
                                         // strlen()

template <class StringT>
void foo(StringT const& s);              // pre-C++17, caller can pass arbitrary char data
                                         // provider, but now foo() has to live in a header
```

Все это можно заменить на:

```
void foo(std::string_view s);            // post-C++17, single argument, tighter coupling
                                         // zero copies regardless of how caller is storing
                                         // the data
```

Обратите внимание, что `std::string_view` *не может* изменять свои базовые данные.

`string_view` полезен, когда вы хотите избежать ненужных копий.

Он предлагает полезное подмножество функций, которые выполняет `std::string`, хотя некоторые из функций ведут себя по-разному:

```
std::string str = "lllllooonnnngggg ssssttttrrrriinnnggg"; //A really long string

//Bad way - 'string::substr' returns a new string (expensive if the string is long)
std::cout << str.substr(15, 10) << '\n';

//Good way - No copies are created!
std::string_view view = str;

// string_view::substr returns a new string_view
std::cout << view.substr(15, 10) << '\n';
```

Пересечение каждого символа

C++ 11

`std::string` поддерживает итераторы, поэтому вы можете использовать цикл, *основанный на диапазоне*, для повторения каждого символа:

```
std::string str = "Hello World!";
for (auto c : str)
    std::cout << c;
```

Вы можете использовать «традиционные» `for` цикла в цикле через каждый символ:

```
std::string str = "Hello World!";
for (std::size_t i = 0; i < str.length(); ++i)
    std::cout << str[i];
```

Преобразование в целые числа / типы с плавающей запятой

`std::string` содержащая число, может быть преобразована в целочисленный тип или тип с плавающей точкой, используя функции преобразования.

Обратите внимание: все эти функции перестают анализировать входную строку, как только они сталкиваются с нечисловым символом, поэтому `"123abc"` будет преобразован в `123`.

Семейство функций `std::ato*` преобразует строки C-стиля (массивы символов) в типы с целыми или с плавающей запятой:

```
std::string ten = "10";

double num1 = std::atof(ten.c_str());
int num2 = std::atoi(ten.c_str());
long num3 = std::atol(ten.c_str());
```

C ++ 11

```
long long num4 = std::atoll(ten.c_str());
```

Однако использование этих функций не рекомендуется, потому что они возвращают `0` если они не могут проанализировать строку. Это плохо, потому что `0` также может быть допустимым результатом, если, например, входная строка была «0», поэтому невозможно определить, действительно ли было выполнено преобразование.

Новое семейство функций `std::sto*` преобразует `std::string s` в типы с целыми числами или с плавающей точкой и генерирует исключения, если они не могут проанализировать их ввод. *Вы должны использовать эти функции, если это возможно :*

C ++ 11

```
std::string ten = "10";
```

```
int num1 = std::stoi(ten);
long num2 = std::stol(ten);
long long num3 = std::stoll(ten);

float num4 = std::stof(ten);
double num5 = std::stod(ten);
long double num6 = std::stold(ten);
```

Кроме того, эти функции также обрабатывают восьмеричные и шестнадцатеричные строки, в отличие от семейства `std::ato*`. Второй параметр - это указатель на первый неотвернутый символ во входной строке (здесь не проиллюстрирован), а третий параметр - база для использования. 0 - автоматическое обнаружение восьмеричного (начиная с 0) и шестнадцатеричного (начиная с 0x или 0X), и любое другое значение является базой для использования

```
std::string ten = "10";
std::string ten_octal = "12";
std::string ten_hex = "0xA";

int num1 = std::stoi(ten, 0, 2); // Returns 2
int num2 = std::stoi(ten_octal, 0, 8); // Returns 10
long num3 = std::stol(ten_hex, 0, 16); // Returns 10
long num4 = std::stol(ten_hex); // Returns 0
long num5 = std::stol(ten_hex, 0, 0); // Returns 10 as it detects the leading 0x
```

Преобразование между кодировками символов

Преобразование между кодировками легко с C++ 11, и большинство компиляторов могут справиться с ним кросс-платформенным способом с помощью `<codecvt>` и `<locale>`.

```
#include <iostream>
#include <codecvt>
#include <locale>
#include <string>
using namespace std;

int main() {
    // converts between wstring and utf8 string
    wstring_convert<codecvt_utf8_utf16<wchar_t>> wchar_to_utf8;
    // converts between ul6string and utf8 string
    wstring_convert<codecvt_utf8_utf16<char16_t>, char16_t> utf16_to_utf8;

    wstring wstr = L"foobar";
    string utf8str = wchar_to_utf8.to_bytes(wstr);
    wstring wstr2 = wchar_to_utf8.from_bytes(utf8str);

    wcout << wstr << endl;
    cout << utf8str << endl;
    wcout << wstr2 << endl;

    ul6string ul6str = u"foobar";
    string utf8str2 = utf16_to_utf8.to_bytes(ul6str);
    ul6string ul6str2 = utf16_to_utf8.from_bytes(utf8str2);

    return 0;
```



```
}
```

Имейте в виду, что Visual Studio 2015 обеспечивает поддержку для этих преобразований, но [ошибка](#) в их реализации библиотеки требует использования другого шаблона для `wstring_convert` при работе с `char16_t` :

```
using utf16_char = unsigned short;
wstring_convert<codecvt_utf8_utf16<utf16_char>, utf16_char> conv_utf8_utf16;

void strings::utf16_to_utf8(const std::u16string& utf16, std::string& utf8)
{
    std::basic_string<utf16_char> tmp;
    tmp.resize(utf16.length());
    std::copy(utf16.begin(), utf16.end(), tmp.begin());
    utf8 = conv_utf8_utf16.to_bytes(tmp);
}

void strings::utf8_to_utf16(const std::string& utf8, std::u16string& utf16)
{
    std::basic_string<utf16_char> tmp = conv_utf8_utf16.from_bytes(utf8);
    utf16.clear();
    utf16.resize(tmp.length());
    std::copy(tmp.begin(), tmp.end(), utf16.begin());
}
```

Проверка, является ли строка префиксом другого

C ++ 14

В C ++ 14 это легко сделать с помощью [std::mismatch](#) которая возвращает первую несоответствующую пару из двух диапазонов:

```
std::string prefix = "foo";
std::string string = "foobar";

bool isPrefix = std::mismatch(prefix.begin(), prefix.end(),
    string.begin(), string.end()).first == prefix.end();
```

Следует отметить , что диапазон-и с половиной версия `mismatch()` существовала до C ++ 14, но это небезопасно в случае, когда вторая строка является короче из двух.

C ++ 14

Мы все еще можем использовать версию `std::mismatch()` в диапазоне от половины до половины, но нам нужно сначала проверить, что первая строка не превосходит вторую:

```
bool isPrefix = prefix.size() <= string.size() &&
    std::mismatch(prefix.begin(), prefix.end(),
        string.begin(), string.end()).first == prefix.end();
```

C ++ 17

С `std::string_view` мы можем написать прямое сравнение, которое мы хотим, не беспокоясь

о накладных `std::string_view` или создании копий:

```
bool isPrefix(std::string_view prefix, std::string_view full)
{
    return prefix == full.substr(0, prefix.size());
}
```

Преобразование в `std::string`

`std::ostringstream` можно использовать для преобразования любого потокового типа в строковое представление, вставив объект в объект `std::ostringstream` (с оператором вставки потока `<<`) и затем преобразуя весь `std::ostringstream` в `std::string`.

Например, для `int`:

```
#include <sstream>

int main()
{
    int val = 4;
    std::ostringstream str;
    str << val;
    std::string converted = str.str();
    return 0;
}
```

Написав свою собственную функцию преобразования, просто:

```
template<class T>
std::string toString(const T& x)
{
    std::ostringstream ss;
    ss << x;
    return ss.str();
}
```

работает, но не подходит для критического кода производительности.

Определяемые пользователем классы могут при необходимости реализовать оператор вставки потока:

```
std::ostream operator<<( std::ostream& out, const A& a )
{
    // write a string representation of a to out
    return out;
}
```

C++ 11

Помимо потоков, начиная с C++ 11, вы также можете использовать `std::to_string` (и `std::to_wstring`), которая перегружена для всех основных типов и возвращает строковое

представление ее параметра.

```
std::string s = to_string(0x12f3); // after this the string s contains "4851"
```

Прочитайте стандарт :: строка онлайн: <https://riptutorial.com/ru/cplusplus/topic/488/станд----строка>

глава 119: Стандарт ISO C ++

Вступление

В 1998 году появилась первая публикация стандарта, в соответствии с которой C ++ - стандартизованный язык. С этого времени C ++ развился, в результате чего появились разные диалекты C ++. На этой странице вы можете найти обзор всех разных стандартов и их изменений по сравнению с предыдущей версией. Подробности о том, как использовать эти функции, описаны на более специализированных страницах.

замечания

Когда упоминается C ++, часто ссылаются на «Стандарт». Но что это за стандарт?

C ++ имеет долгую историю. Начав в качестве небольшого проекта Bjarne Stroustrup в Bell Labs, к началу 90-х годов он стал довольно популярным. Несколько компаний создавали компиляторы C ++, чтобы пользователи могли запускать свои компиляторы C ++ на самых разных компьютерах. Но для того, чтобы облегчить это, все эти конкурирующие компиляторы должны делиться одним определением языка.

В этот момент язык C успешно был стандартизирован. Это означает, что было написано формальное описание языка. Это было представлено Американскому национальному институту стандартов (ANSI), который открыл документ для обзора и впоследствии опубликовал его в 1989 году. Спустя год Международная организация стандартов (поскольку у нее были бы разные акронимы на разных языках, они выбрали одну форму, ISO, полученный от греческого isos, что означает равный.) Принял американский стандарт в качестве международного стандарта.

Для C ++ с самого начала было ясно, что международный интерес. Была начата рабочая группа по ИСО (известна как WG21 в рамках Субкомитета 22). Эта рабочая группа подготовила первый стандарт примерно в 1995 году. Но, как мы знаем программисты, нет ничего более опасного для запланированной доставки, чем функции последней минуты, и это случилось и с C ++. В 1995 году появилась крутая новая библиотека под названием STL, и люди, работающие в WG21, решили добавить сглаженную версию к стандартному проекту C ++. Естественно, это привело к тому, что крайние сроки были упущены, и только 3 года спустя документ стал окончательным. ISO - очень формальная организация, поэтому стандарт C ++ был окрещен не очень товарным именем ISO / IEC 14882. Поскольку стандарты могут быть обновлены, эта точная версия стала известна как 14882: 1998.

И действительно, требовалось обновить Стандарт. Стандарт - очень толстый документ, целью которого является точное описание того, как должны работать компиляторы C ++. Даже небольшая двусмысленность может стоить исправления, поэтому к 2003 году было

выпущено обновление 14882: 2003. Однако это не добавило никакой функции на C ++; новые функции были запланированы для второго обновления.

Неофициально это второе обновление было известно как C ++ 0x, потому что неизвестно, будет ли это требоваться до 2008 или 2009 года. Ну, эта версия также получила небольшую задержку, поэтому стала 14882: 2011.

К счастью, WG21 решила не допускать этого снова. C ++ 11 был хорошо принят и позволил возобновить интерес к C ++. Итак, чтобы сохранить импульс, третье обновление прошло от планирования до публикации через 3 года, чтобы стать 14882: 2014.

Работа там тоже не прекращалась. Был предложен стандарт C ++ 17, и была запущена работа для C ++ 20.

Examples

Текущие рабочие проекты

Все опубликованные стандарты ИСО доступны для продажи в магазине ISO (<http://www.iso.org>). Однако рабочие проекты стандартов C ++ доступны бесплатно.

Различные версии стандарта:

- Предстоящие (иногда называемые C ++ 20 или C ++ 2a): [текущий рабочий черновик \(HTML-версия\)](#))
- Предлагается (иногда называется C ++ 17 или C ++ 1z): [март 2017 рабочий проект N4659](#) .
- C ++ 14 (Иногда упоминается как C ++ 1y): [ноябрь 2014 рабочий проект N4296](#)
- C ++ 11 (иногда называемый C ++ 0x): [февраль 2011 года рабочий проект N3242](#)
- C ++ 03
- C ++ 98

C ++ 11

Стандарт C ++ 11 является основным расширением стандарта C ++. Ниже вы можете ознакомиться с обзором изменений, поскольку они были сгруппированы по [часто задаваемому файлу isocpp](#) с ссылками на более подробную документацию.

Расширения языка

Общие функции

- [авто](#)

- [decltype](#)
- [Оператор Range-for](#)
- Списки инициализаторов
- Унифицированный синтаксис и семантика инициализации
- [Ссылки Rvalue](#) и [семантика перемещения](#)
- [Лямбда](#)
- [noexcept](#) для предотвращения распространения исключений
- [constexpr](#)
- [nullptr](#) - литерал с нулевым указателем
- Копирование и повторное удаление исключений
- Внутренние пространства имен
- Пользовательские литералы

Классы

- `= default` и `= delete`
- Управление перемещением и копированием по умолчанию
- Делегирование конструкторов
- Инициализаторы членов класса
- Унаследованные конструкторы
- Элементы управления переопределением: переопределить
- Элементы управления переопределением: окончательный
- Явные операторы преобразования

Другие типы

- класс `enum`
- `long long` - более длинное целое число
- Расширенные целые типы
- Обобщенные союзы
- Обобщенные POD

Шаблоны

- Внешние шаблоны
- Шаблоны псевдонимы
- Шаблоны Variadic
- Локальные типы в качестве аргументов шаблона

совпадение

- Модель памяти параллелизма

- Динамическая инициализация и уничтожение с параллелизмом
- [Потоковое локальное хранилище](#)

Различные функции языка

- Какова ценность `__cplusplus` для C++ 11?
- Синтаксис типа возврата суффикса
- Предотвращение сужения
- Прямоугольные скобки
- [static_assert](#) утверждения времени компиляции
- Строковые литералы
- Атрибуты
- центровка
- Функции C99

Расширения библиотек

генеральный

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`
- Сбор мусора ABI
- кортеж
- Типовые черты
- функция и привязка
- Регулярные выражения
- Утилиты времени
- Генерация случайных чисел
- Ограниченные распределители

Контейнеры и алгоритмы

- Улучшение алгоритмов
- Улучшение контейнеров
- `unordered_*` контейнеры
- `std::array`
- `forward_list`

совпадение

- [Потоки](#)
- Взаимное исключение
- [Замки](#)
- [Переменные условия](#)
- [атомная энергетика](#)
- [Фьючерсы и обещания](#)
- [асинхронной](#)
- Отказ от процесса

C ++ 14

Стандарт C ++ 14 часто упоминается как исправление для C ++ 11. Он содержит только ограниченный список изменений, большинство из которых являются расширениями новых функций на C ++ 11. Ниже вы можете ознакомиться с обзором изменений, поскольку они были сгруппированы по [часто задаваемому файлу isocpp](#) с ссылками на более подробную документацию.

Расширения языка

- Бинарные литералы
- Вычет обобщенного типа возврата
- `decltype` (авто)
- [Обобщенные лямбда-захваты](#)
- [Родовые лямбды](#)
- Переменные шаблоны
- Расширенный `constexpr`
- [Атрибут `\[\[deprecated\]\]`](#)
- [Цифровые разделители](#)

Расширения библиотек

- Общая блокировка
- Пользовательские литералы для `std::types`
- `std::make_unique`
- Тип преобразования `_t aliases`
- [Адресация кортежей по типу](#) (например, `get<string>(t)`)
- [Прозрачные функциональные операторы](#) (например, `greater<(x)`)
- `std::quoted`

Устаревшее / удалено

- `std::gets` устарел в C ++ 11 и удален из C ++ 14
- `std::random_shuffle` устарел

C ++ 17

Стандарт C ++ 17 является полнофункциональным и был предложен для стандартизации. В компиляторах с экспериментальной поддержкой этих функций он обычно называется C ++ 1z.

Расширения языка

- Сгибание выражений
- декларирование аргументы шаблона не типа `c auto`
- Гарантированное копирование
- Вычисление параметра шаблона для конструкторов
- Структурированные привязки
- Компактные вложенные пространства имен
- Новые атрибуты: `[[fallthrough]]`, `[[nodiscard]]`, `[[maybe_unused]]`
- Сообщение по умолчанию для `static_assert`
- Инициализаторы в `if` и `switch`
- Встроенные переменные
- `if constexpr`
- Порядок гарантий оценки выражений
- Динамическое распределение памяти для перенастроенных данных

Расширения библиотек

- `std::optional`
- `std::variant`
- `std::string_view`
- `merge()` и `extract()` для ассоциативных контейнеров
- Библиотека файловой системы с заголовком `<filesystem>`.
- Параллельные версии большинства стандартных алгоритмов (в заголовке `<algorithm>`).
- Добавление математических специальных функций в заголовок `<cmath>`.
- Перемещение узлов между `map <>`, `unordered_map <>`, `set <>` и `unordered_set <>`

C ++ 03

В стандарте C ++ 03 в основном рассматриваются отчеты о дефектах стандарта C ++ 98. Помимо этих дефектов, он добавляет только одну новую функцию.

Расширения языка

- [Ценность инициализации](#)

C ++ 98

C ++ 98 - первая стандартизованная версия C ++. Поскольку он был разработан как расширение для C, добавлены многие функции, которые выделяют C ++ из C.

Расширения языка (в отношении C89 / C90)

- Классы, Производные классы, виртуальные функции-члены, константные функции-члены
- Перегрузка функций, перегрузка оператора
- Комментарии по одной строке (введена в C-language со стандартом C99)
- Рекомендации
- новый и удалить
- boolean type (был введен в C-language со стандартом C99)
- шаблоны
- Пространства имен
- исключения
- конкретные броски

Расширения библиотек

- Библиотека стандартных шаблонов

C ++ 20

C ++ 20 - это предстоящий стандарт C ++, который в настоящее время находится в разработке, основанный на стандарте C ++ 17. Прогресс можно отслеживать на [официальном сайте ISO cpp](#) .

Следующие функции - это просто то, что было принято для следующей версии стандарта C ++, ориентированного на 2020 год.

Расширения языка

На данный момент никаких языковых расширений не принято.

Расширения библиотек

На данный момент никаких расширений библиотеки не принято.

Прочитайте Стандарт ISO C ++ онлайн: <https://riptutorial.com/ru/cplusplus/topic/2742/стандарт-iso-c-plusplus>

глава 120: Стандартные библиотечные алгоритмы

Examples

станд :: for_each

```
template<class InputIterator, class Function>
    Function for_each(InputIterator first, InputIterator last, Function f);
```

Последствия:

Применяет `f` к результату разыменования каждого итератора в диапазоне `[first, last)` начиная с `first` и заканчивая `last - 1`.

Параметры:

`first, last` - диапазон для применения `f` к.

`f` - вызываемый объект, который применяется к результату разыменования каждого итератора в диапазоне `[first, last)`.

Возвращаемое значение:

`f` (до C++ 11) и `std::move(f)` (начиная с C++ 11).

Сложность:

Применяет `f` точно `last - first` раз.

Пример:

C++ 11

```
std::vector<int> v { 1, 2, 4, 8, 16 };
std::for_each(v.begin(), v.end(), [](int elem) { std::cout << elem << " "; });
```

Применяет данную функцию для каждого элемента вектора `v` печатает этот элемент в `stdout`.

станд :: next_permutation

```
template< class Iterator >
bool next_permutation( Iterator first, Iterator last );
template< class Iterator, class Compare >
```

```
bool next_permutation( Iterator first, Iterator last, Compare cmpFun );
```

Последствия:

Просеять последовательность данных диапазона [первая, последняя] в следующую лексикографически большую перестановку. Если `cmpFun`, настраивается правило перестановки.

Параметры:

`first` - начало диапазона, подлежащего перестановке, включительно

`last` - конец диапазона, который должен быть перестановочен, эксклюзивный

Возвращаемое значение:

Возвращает `true`, если такая перестановка существует.

В противном случае диапазон сводится к лексикографически наименьшей перестановке и возвращает `false`.

Сложность:

O (n), n - расстояние от `first` до `last`.

Пример :

```
std::vector< int > v { 1, 2, 3 };
do
{
    for( int i = 0; i < v.size(); i += 1 )
    {
        std::cout << v[i];
    }
    std::cout << std::endl;
}while( std::next_permutation( v.begin(), v.end() ) );
```

напечатайте все случаи перестановок 1,2,3 в лексикографически возрастающем порядке.
выход:

```
123
132
213
231
312
321
```

станд :: аккумулируют

Определено в заголовке `<numeric>`

```
template<class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init); // (1)

template<class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation f); // (2)
```

Последствия:

`std::accumulate` выполняет операцию **сброса** с использованием функции `f` в диапазоне `[first, last)` начиная с `init` как значение аккумулятора.

Фактически это эквивалентно:

```
T acc = init;
for (auto it = first; first != last; ++it)
    acc = f(acc, *it);
return acc;
```

В версии (1) `operator+` используется вместо `f`, поэтому накопление над контейнером эквивалентно сумме элементов контейнера.

Параметры:

`first, last` - диапазон для применения `f` к.

`init` - начальное значение аккумулятора.

`f` - двоичная функция сгиба.

Возвращаемое значение:

Накопленное значение `f` приложений.

Сложность:

$O(n \times k)$, где n - расстояние от `first` до `last`, $O(k)$ - сложность функции `f`.

Пример:

Пример простой суммы:

```
std::vector<int> v { 2, 3, 4 };
auto sum = std::accumulate(v.begin(), v.end(), 1);
std::cout << sum << std::endl;
```

Выход:

```
10
```

Преобразование цифр в число:

C ++ 11

```
class Converter {
public:
    int operator()(int a, int d) const { return a * 10 + d; }
};
```

и позже

```
const int ds[3] = {1, 2, 3};
int n = std::accumulate(ds, ds + 3, 0, Converter());
std::cout << n << std::endl;
```

C++ 11

```
const std::vector<int> ds = {1, 2, 3};
int n = std::accumulate(ds.begin(), ds.end(),
                        0,
                        [](int a, int d) { return a * 10 + d; });
std::cout << n << std::endl;
```

Выход:

```
123
```

станд :: найти

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val);
```

Последствия

Находит первое вхождение val в диапазоне [первый, последний]

параметры

`first` => итератор, указывающий на начало диапазона `last` => iterator, указывающий на конец диапазона `val` => Значение для поиска в пределах диапазона

Вернуть

Итератор, который указывает на первый элемент в пределах диапазона, который равен (==) в val, итератор указывает на последний, если val не найден.

пример

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {

    //create a vector
    vector<int> intVec {4, 6, 8, 9, 10, 30, 55, 100, 45, 2, 4, 7, 9, 43, 48};

    //define iterators
```

```

vector<int>::iterator itr_9;
vector<int>::iterator itr_43;
vector<int>::iterator itr_50;

//calling find
itr_9 = find(intVec.begin(), intVec.end(), 9); //occurs twice
itr_43 = find(intVec.begin(), intVec.end(), 43); //occurs once

//a value not in the vector
itr_50 = find(intVec.begin(), intVec.end(), 50); //does not occur

cout << "first occurence of: " << *itr_9 << endl;
cout << "only occurence of: " << *itr_43 << endl;

/*
let's prove that itr_9 is pointing to the first occurence
of 9 by looking at the element after 9, which should be 10
not 43
*/
cout << "element after first 9: " << *(itr_9 + 1) << endl;

/*
to avoid dereferencing intVec.end(), lets look at the
element right before the end
*/
cout << "last element: " << *(itr_50 - 1) << endl;

return 0;
}

```

Выход

```

first occurence of: 9
only occurence of: 43
element after first 9: 10
last element: 48

```

станд :: Количество

```

template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count (InputIterator first, InputIterator last, const T& val);

```

Последствия

Подсчитывает количество элементов, равное val

параметры

first => iterator, указывающий на начало диапазона

last => итератор, указывающий на конец диапазона

val => Внесение этого значения в диапазон будет засчитано

Вернуть

Число элементов в диапазоне, равных (==) - val.

пример

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {

    //create vector
    vector<int> intVec{4,6,8,9,10,30,55,100,45,2,4,7,9,43,48};

    //count occurrences of 9, 55, and 101
    size_t count_9 = count(intVec.begin(), intVec.end(), 9); //occurs twice
    size_t count_55 = count(intVec.begin(), intVec.end(), 55); //occurs once
    size_t count_101 = count(intVec.begin(), intVec.end(), 101); //occurs once

    //print result
    cout << "There are " << count_9 << " 9s"<< endl;
    cout << "There is " << count_55 << " 55"<< endl;
    cout << "There is " << count_101 << " 101"<< ends;

    //find the first element == 4 in the vector
    vector<int>::iterator itr_4 = find(intVec.begin(), intVec.end(), 4);

    //count its occurrences in the vector starting from the first one
    size_t count_4 = count(itr_4, intVec.end(), *itr_4); // should be 2

    cout << "There are " << count_4 << " " << *itr_4 << endl;

    return 0;
}
```

Выход

```
There are 2 9s
There is 1 55
There is 0 101
There are 2 4
```

станд :: count_if

```
template <class InputIterator, class UnaryPredicate>
typename iterator_traits<InputIterator>::difference_type
count_if (InputIterator first, InputIterator last, UnaryPredicate red);
```

Последствия

Подсчитывает количество элементов в диапазоне, для которого истинна заданная функция предиката

параметры

`first` => iterator, указывающий на начало диапазона `last` => iterator, указывающий на конец диапазона `red` => предикатная функция (возвращает true или false)

Вернуть

Количество элементов в указанном диапазоне, для которых функция предиката вернулась.

пример

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

/*
   Define a few functions to use as predicates
*/

//return true if number is odd
bool isOdd(int i){
    return i%2 == 1;
}

//functor that returns true if number is greater than the value of the constructor parameter
provided
class Greater {
    int _than;
public:
    Greater(int th): _than(th){}
    bool operator()(int i){
        return i > _than;
    }
};

int main(int argc, const char * argv[]) {

    //create a vector
    vector<int> myvec = {1,5,8,0,7,6,4,5,2,1,5,0,6,9,7};

    //using a lambda function to count even numbers
    size_t evenCount = count_if(myvec.begin(), myvec.end(), [](int i){return i % 2 == 0;}); //
    >= C++11

    //using function pointer to count odd number in the first half of the vector
    size_t oddCount = count_if(myvec.begin(), myvec.end()- myvec.size()/2, isOdd);

    //using a functor to count numbers greater than 5
    size_t greaterCount = count_if(myvec.begin(), myvec.end(), Greater(5));

    cout << "vector size: " << myvec.size() << endl;
    cout << "even numbers: " << evenCount << " found" << endl;
    cout << "odd numbers: " << oddCount << " found" << endl;
    cout << "numbers > 5: " << greaterCount << " found"<< endl;

    return 0;
}
```

```
}
```

Выход

```
vector size: 15
even numbers: 7 found
odd numbers: 4 found
numbers > 5: 6 found
```

станд :: find_if

```
template <class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred);
```

Последствия

Находит первый элемент в диапазоне, для которого предикатная функция `pred` возвращает `true`.

параметры

`first` => iterator, указывающий на начало диапазона `last` => iterator, указывающий на конец диапазона `pred` => предикатная функция (возвращает `true` или `false`)

Вернуть

Итератор, который указывает на первый элемент внутри диапазона, предикатная функция `pred` возвращает `true`. Итератор указывает на последний, если `val` не найден

пример

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

/*
    define some functions to use as predicates
*/

//Returns true if x is multiple of 10
bool multOf10(int x) {
    return x % 10 == 0;
}

//returns true if item greater than passed in parameter
class Greater {
    int _than;

public:
```

```

    Greater(int th):_than(th){

    }
    bool operator()(int data) const
    {
        return data > _than;
    }
};

int main()
{

    vector<int> myvec {2, 5, 6, 10, 56, 7, 48, 89, 850, 7, 456};

    //with a lambda function
    vector<int>::iterator gt10 = find_if(myvec.begin(), myvec.end(), [](int x){return x>10;});
    // >= C++11

    //with a function pointer
    vector<int>::iterator pow10 = find_if(myvec.begin(), myvec.end(), multOf10);

    //with functor
    vector<int>::iterator gt5 = find_if(myvec.begin(), myvec.end(), Greater(5));

    //not Found
    vector<int>::iterator nf = find_if(myvec.begin(), myvec.end(), Greater(1000)); // nf points
to myvec.end()

    //check if pointer points to myvec.end()
    if(nf != myvec.end()) {
        cout << "nf points to: " << *nf << endl;
    }
    else {
        cout << "item not found" << endl;
    }

    cout << "First item > 10: " << *gt10 << endl;
    cout << "First Item n * 10: " << *pow10 << endl;
    cout << "First Item > 5: " << *gt5 << endl;

    return 0;
}

```

Выход

```

item not found
First item > 10: 56
First Item n * 10: 10
First Item > 5: 6

```

станд :: min_element

```

template <class ForwardIterator>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last);

```

```
template <class ForwardIterator, class Compare>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last, Compare comp);
```

Последствия

Находит минимальный элемент в диапазоне

параметры

`first` - итератор, указывающий на начало диапазона

`last` - итератор, указывающий на конец диапазона `comp` - указатель функции или объект функции, который принимает два аргумента и возвращает `true` или `false`, указывающий, является ли аргумент меньше аргумента 2. Эта функция не должна изменять входы

Вернуть

Итератор к минимальному элементу в диапазоне

сложность

Линейный в одном меньше, чем количество сравниваемых элементов.

пример

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <utility> //to use make_pair

using namespace std;

//function compare two pairs
bool pairLessThanFunction(const pair<string, int> &p1, const pair<string, int> &p2)
{
    return p1.second < p2.second;
}

int main(int argc, const char * argv[]) {

    vector<int> intVec {30,200,167,56,75,94,10,73,52,6,39,43};

    vector<pair<string, int>> pairVector = {make_pair("y", 25), make_pair("b", 2),
    make_pair("z", 26), make_pair("e", 5) };

    // default using < operator
    auto minInt = min_element(intVec.begin(), intVec.end());

    //Using pairLessThanFunction
    auto minPairFunction = min_element(pairVector.begin(), pairVector.end(),
    pairLessThanFunction);

    //print minimum of intVector
```

```

cout << "min int from default: " << *minInt << endl;

//print minimum of pairVector
cout << "min pair from PairLessThanFunction: " << (*minPairFunction).second << endl;

return 0;
}

```

Выход

```

min int from default: 6
min pair from PairLessThanFunction: 2

```

Использование `std::nth_element` Чтобы найти медиану (или другие кванты)

Алгоритм `std::nth_element` принимает три итератора: итератор в начало, n -ю позицию и конец. После возвращения функции n -й элемент (по порядку) будет n -м наименьшим элементом. (Функция имеет более сложные перегрузки, например, некоторые из них выполняют функции сравнения, см. Ссылку выше для всех вариантов.)

Примечание. Эта функция очень эффективна - она имеет линейную сложность.

Для этого примера давайте определим медиану последовательности длины n как элемента, который будет находиться в позиции $\lceil n / 2 \rceil$. Например, медиана последовательности длины 5 является 3-м наименьшим элементом, а также медиана последовательности длины 6.

Чтобы использовать эту функцию для поиска медианы, мы можем использовать следующее. Скажем, мы начнем с

```

std::vector<int> v{5, 1, 2, 3, 4};

std::vector<int>::iterator b = v.begin();
std::vector<int>::iterator e = v.end();

std::vector<int>::iterator med = b;
std::advance(med, v.size() / 2);

// This makes the 2nd position hold the median.
std::nth_element(b, med, e);

// The median is now at v[2].

```

Чтобы найти p -й **квантиль**, мы бы изменили некоторые из приведенных выше строк:

```

const std::size_t pos = p * std::distance(b, e);

std::advance(nth, pos);

```

и ищите квантиль в позиции `pos` .

Прочитайте Стандартные библиотечные алгоритмы онлайн:

<https://riptutorial.com/ru/cplusplus/topic/3177/стандартные-библиотечные-алгоритмы>

глава 121: СД :: атомарных

Examples

атомные типы

Каждый экземпляр и полная специализация шаблона `std::atomic` определяют атомный тип. Если один поток записывает на атомный объект, а другой поток читает его, поведение корректно (см. Модель памяти для подробностей о гонках данных)

Кроме того, доступ к атомным объектам может устанавливать `std::memory_order` синхронизацию и упорядочивать неатомные обращения к памяти, как определено `std::memory_order`.

`std::atomic` может быть `TriviallyCopyable` type `T`. `std::atomic` с любым `TriviallyCopyable` type `T`. `std::atomic` является ни копируемым, ни подвижным.

Стандартная библиотека предоставляет специализации шаблона `std::atomic` для следующих типов:

1. Определена одна полная специализация для типа `bool` и его имени `typedef`, который рассматривается как неспециализированный `std::atomic<T>` за исключением того, что он имеет стандартный макет, тривиальный конструктор по умолчанию, тривиальные деструкторы и поддерживает синтаксис синтаксиса агрегации:

Typedef name	Полная специализация
<code>std::atomic_bool</code>	<code>std::atomic<bool></code>

- 2) Полные специализации и `typedefs` для интегральных типов, а именно:

Typedef name	Полная специализация
<code>std::atomic_char</code>	<code>std::atomic<char></code>
<code>std::atomic_char</code>	<code>std::atomic<char></code>
<code>std::atomic_schar</code>	<code>std::atomic<signed char></code>
<code>std::atomic_uchar</code>	<code>std::atomic<unsigned char></code>
<code>std::atomic_short</code>	<code>std::atomic<short></code>
<code>std::atomic_ushort</code>	<code>std::atomic<unsigned short></code>
<code>std::atomic_int</code>	<code>std::atomic<int></code>
<code>std::atomic_uint</code>	<code>std::atomic<unsigned int></code>

Typedef name	Полная специализация
<code>std::atomic_long</code>	<code>std::atomic<long></code>
<code>std::atomic_ulong</code>	<code>std::atomic<unsigned long></code>
<code>std::atomic_llong</code>	<code>std::atomic<long long></code>
<code>std::atomic_ullong</code>	<code>std::atomic<unsigned long long></code>
<code>std::atomic_char16_t</code>	<code>std::atomic<char16_t></code>
<code>std::atomic_char32_t</code>	<code>std::atomic<char32_t></code>
<code>std::atomic_wchar_t</code>	<code>std::atomic<wchar_t></code>
<code>std::atomic_int8_t</code>	<code>std::atomic<std::int8_t></code>
<code>std::atomic_uint8_t</code>	<code>std::atomic<std::uint8_t></code>
<code>std::atomic_int16_t</code>	<code>std::atomic<std::int16_t></code>
<code>std::atomic_uint16_t</code>	<code>std::atomic<std::uint16_t></code>
<code>std::atomic_int32_t</code>	<code>std::atomic<std::int32_t></code>
<code>std::atomic_uint32_t</code>	<code>std::atomic<std::uint32_t></code>
<code>std::atomic_int64_t</code>	<code>std::atomic<std::int64_t></code>
<code>std::atomic_uint64_t</code>	<code>std::atomic<std::uint64_t></code>
<code>std::atomic_int_least8_t</code>	<code>std::atomic<std::int_least8_t></code>
<code>std::atomic_uint_least8_t</code>	<code>std::atomic<std::uint_least8_t></code>
<code>std::atomic_int_least16_t</code>	<code>std::atomic<std::int_least16_t></code>
<code>std::atomic_uint_least16_t</code>	<code>std::atomic<std::uint_least16_t></code>
<code>std::atomic_int_least32_t</code>	<code>std::atomic<std::int_least32_t></code>
<code>std::atomic_uint_least32_t</code>	<code>std::atomic<std::uint_least32_t></code>
<code>std::atomic_int_least64_t</code>	<code>std::atomic<std::int_least64_t></code>
<code>std::atomic_uint_least64_t</code>	<code>std::atomic<std::uint_least64_t></code>
<code>std::atomic_int_fast8_t</code>	<code>std::atomic<std::int_fast8_t></code>
<code>std::atomic_uint_fast8_t</code>	<code>std::atomic<std::uint_fast8_t></code>
<code>std::atomic_int_fast16_t</code>	<code>std::atomic<std::int_fast16_t></code>
<code>std::atomic_uint_fast16_t</code>	<code>std::atomic<std::uint_fast16_t></code>
<code>std::atomic_int_fast32_t</code>	<code>std::atomic<std::int_fast32_t></code>
<code>std::atomic_uint_fast32_t</code>	<code>std::atomic<std::uint_fast32_t></code>
<code>std::atomic_int_fast64_t</code>	<code>std::atomic<std::int_fast64_t></code>
<code>std::atomic_uint_fast64_t</code>	<code>std::atomic<std::uint_fast64_t></code>

Typedef name	Полная специализация
<code>std::atomic_intptr_t</code>	<code>std::atomic<std::intptr_t></code>
<code>std::atomic_uintptr_t</code>	<code>std::atomic<std::uintptr_t></code>
<code>std::atomic_size_t</code>	<code>std::atomic<std::size_t></code>
<code>std::atomic_ptrdiff_t</code>	<code>std::atomic<std::ptrdiff_t></code>
<code>std::atomic_intmax_t</code>	<code>std::atomic<std::intmax_t></code>
<code>std::atomic_uintmax_t</code>	<code>std::atomic<std::uintmax_t></code>

Простой пример использования `std::atomic_int`

```
#include <iostream>           // std::cout
#include <atomic>              // std::atomic, std::memory_order_relaxed
#include <thread>              // std::thread

std::atomic_int foo (0);

void set_foo(int x) {
    foo.store(x, std::memory_order_relaxed);    // set value atomically
}

void print_foo() {
    int x;
    do {
        x = foo.load(std::memory_order_relaxed); // get value atomically
    } while (x==0);
    std::cout << "foo: " << x << '\n';
}

int main ()
{
    std::thread first (print_foo);
    std::thread second (set_foo,10);
    first.join();
    //second.join();
    return 0;
}
//output: foo: 10
```

Прочитайте STD :: атомарных онлайн: <https://riptutorial.com/ru/cplusplus/topic/7475/стд-атомарных>

глава 122: Структуры данных в C ++

Examples

Внедрение Linked List в C ++

Создание узла списка

```
class listNode
{
    public:
    int data;
    listNode *next;
    listNode(int val):data(val),next(NULL){}
};
```

Создание класса List

```
class List
{
    public:
    listNode *head;
    List():head(NULL){}
    void insertAtBegin(int val);
    void insertAtEnd(int val);
    void insertAtPos(int val);
    void remove(int val);
    void print();
    ~List();
};
```

Вставьте новый узел в начале списка.

```
void List::insertAtBegin(int val)//inserting at front of list
{
    listNode *newnode = new listNode(val);
    newnode->next=this->head;
    this->head=newnode;
}
```

Вставьте новый узел в конец списка

```
void List::insertAtEnd(int val) //inserting at end of list
{
    if(head==NULL)
    {
        insertAtBegin(val);
        return;
    }
    listNode *newnode = new listNode(val);
    listNode *ptr=this->head;
```

```

while(ptr->next!=NULL)
{
    ptr=ptr->next;
}
ptr->next=newnode;
}

```

Вставить в определенное положение в списке

```

void List::insertAtPos(int pos,int val)
{
    listNode *newnode=new listNode(val);
    if(pos==1)
    {
        //as head
        newnode->next=this->head;
        this->head=newnode;
        return;
    }
    pos--;
    listNode *ptr=this->head;
    while(ptr!=NULL && --pos)
    {
        ptr=ptr->next;
    }
    if(ptr==NULL)
        return;//not enough elements
    newnode->next=ptr->next;
    ptr->next=newnode;
}

```

Удаление узла из списка

```

void List::remove(int toBeRemoved)//removing an element
{
    if(this->head==NULL)
        return; //empty
    if(this->head->data==toBeRemoved)
    {
        //first node to be removed
        listNode *temp=this->head;
        this->head=this->head->next;
        delete(temp);
        return;
    }
    listNode *ptr=this->head;
    while(ptr->next!=NULL && ptr->next->data!=toBeRemoved)
        ptr=ptr->next;
    if(ptr->next==NULL)
        return;//not found
    listNode *temp=ptr->next;
    ptr->next=ptr->next->next;
    delete(temp);
}

```

Печать списка

```
void List::print()//printing the list
{
    listNode *ptr=this->head;
    while(ptr!=NULL)
    {
        cout<<ptr->data<<" ";
        ptr=ptr->next;
    }
    cout<<endl;
}
```

Деструктор для списка

```
List::~~List()
{
    listNode *ptr=this->head, *next=NULL;
    while(ptr!=NULL)
    {
        next=ptr->next;
        delete(ptr);
        ptr=next;
    }
}
```

Прочитайте Структуры данных в C ++ онлайн: <https://riptutorial.com/ru/cplusplus/topic/7485/структуры-данных-в-c-plusplus>

глава 123: Структуры синхронизации

ПОТОКОВ

Вступление

Для взаимодействия с **потоками** могут потребоваться некоторые методы синхронизации, если потоки взаимодействуют. В этом разделе вы можете найти различные структуры, которые предоставляются стандартной библиотекой для решения этих проблем.

Examples

станд :: shared_lock

Shared_lock можно использовать в сочетании с уникальной блокировкой, позволяющей нескольким читателям и эксклюзивным авторам.

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
public:
    string getPhoneNo( const std::string & name )
    {
        shared_lock<shared_timed_mutex> r(_protect);
        auto it = _phonebook.find( name );
        if ( it == _phonebook.end() )
            return (*it).second;
        return "";
    }
    void addPhoneNo ( const std::string & name, const std::string & phone )
    {
        unique_lock<shared_timed_mutex> w(_protect);
        _phonebook[name] = phone;
    }

    shared_timed_mutex _protect;
    unordered_map<string, string> _phonebook;
};
```

std :: call_once, std :: once_flag

std::call_once обеспечивает выполнение функции ровно один раз с помощью конкурирующих потоков. Он бросает std::system_error в случае, если он не может выполнить свою задачу.

Используется вместе с `std::once_flag`.

```
#include <mutex>
#include <iostream>

std::once_flag flag;
void do_something() {
    std::call_once(flag, []() { std::cout << "Happens once" << std::endl; });

    std::cout << "Happens every time" << std::endl;
}
```

Блокировка объектов для эффективного доступа.

Часто вы хотите заблокировать весь объект во время выполнения нескольких операций над ним. Например, если вам нужно изучить или изменить объект с помощью *итераторов*. Всякий раз, когда вам нужно вызывать несколько функций-членов, обычно более эффективно блокировать весь объект, а не отдельные функции-члены.

Например:

```
class text_buffer
{
    // for readability/maintainability
    using mutex_type = std::shared_timed_mutex;
    using reading_lock = std::shared_lock<mutex_type>;
    using updates_lock = std::unique_lock<mutex_type>;

public:
    // This returns a scoped lock that can be shared by multiple
    // readers at the same time while excluding any writers
    [[nodiscard]]
    reading_lock lock_for_reading() const { return reading_lock(mtx); }

    // This returns a scoped lock that is excluding to one
    // writer preventing any readers
    [[nodiscard]]
    updates_lock lock_for_updates() { return updates_lock(mtx); }

    char* data() { return buf; }
    char const* data() const { return buf; }

    char* begin() { return buf; }
    char const* begin() const { return buf; }

    char* end() { return buf + sizeof(buf); }
    char const* end() const { return buf + sizeof(buf); }

    std::size_t size() const { return sizeof(buf); }

private:
    char buf[1024];
    mutable mutex_type mtx; // mutable allows const objects to be locked
};
```

При вычислении контрольной суммы объект блокируется для чтения, позволяя другим

потокам, которые хотят читать с объекта одновременно, для этого.

```
std::size_t checksum(text_buffer const& buf)
{
    std::size_t sum = 0xA44944A4;

    // lock the object for reading
    auto lock = buf.lock_for_reading();

    for(auto c: buf)
        sum = (sum << 8) | (((unsigned char) ((sum & 0xFF000000) >> 24)) ^ c);

    return sum;
}
```

Очистка объекта обновляет его внутренние данные, поэтому его необходимо выполнить с помощью блокировки исключения.

```
void clear(text_buffer& buf)
{
    auto lock = buf.lock_for_updates(); // exclusive lock
    std::fill(std::begin(buf), std::end(buf), '\0');
}
```

При получении более чем одной блокировки следует следить за тем, чтобы всегда приобретать блокировки в одном порядке для всех потоков.

```
void transfer(text_buffer const& input, text_buffer& output)
{
    auto lock1 = input.lock_for_reading();
    auto lock2 = output.lock_for_updates();

    std::copy(std::begin(input), std::end(input), std::begin(output));
}
```

note: Это лучше всего сделать с помощью `std::deferred::lock` и вызова `std::lock`

`std::condition_variable_any`, `std::cv_status`

Обобщение `std::condition_variable`, `std::condition_variable_any` работает с любым типом структуры `BasicLockable`.

`std::cv_status` как статус возврата для переменной условия имеет два возможных кода возврата:

- `std::cv_status::no_timeout`: не было таймаута, переменная условия была уведомлена
- `std::cv_status::no_timeout`: время ожидания условия

Прочитайте Структуры синхронизации потоков онлайн:

<https://riptutorial.com/ru/cplusplus/topic/9794/структуры-синхронизации-потоков>

глава 124: Тестирование модулей в C ++

Вступление

Модульное тестирование - это уровень тестирования программного обеспечения, который проверяет поведение и правильность единиц кода.

В C ++ «единицы кода» часто относятся к любым классам, функциям или группам. Модульное тестирование часто выполняется с использованием специализированных «тестовых рамок» или «библиотек тестирования», которые часто используют нетривиальные синтаксисы или шаблоны использования.

В этом разделе будут рассмотрены различные стратегии и библиотеки модулей тестирования или фреймворки.

Examples

Тест Google

[Google Test](#) - это платформа тестирования C ++, поддерживаемая Google. Он требует создания библиотеки `gtest` и привязки ее к вашей структуре тестирования при создании файла тестового примера.

Минимальный пример

```
// main.cpp

#include <gtest/gtest.h>
#include <iostream>

// Google Test test cases are created using a C++ preprocessor macro
// Here, a "test suite" name and a specific "test name" are provided.
TEST(module_name, test_name) {
    std::cout << "Hello world!" << std::endl;
    // Google Test will also provide macros for assertions.
    ASSERT_EQ(1+1, 2);
}

// Google Test can be run manually from the main() function
// or, it can be linked to the gtest_main library for an already
// set-up main() function primed to accept Google Test test cases.
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}
```

```
// Build command: g++ main.cpp -lgtest
```

Ловить

Catch - это библиотека только для заголовков, которая позволяет использовать как тестовый стиль блока **TDD**, так и **BDD**.

Следующий фрагмент представлен на странице документации Catch по [этой ссылке](#) :

```
SCENARIO( "vectors can be sized and resized", "[vector]" ) {
    GIVEN( "A vector with some items" ) {
        std::vector v( 5 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );

        WHEN( "the size is increased" ) {
            v.resize( 10 );

            THEN( "the size and capacity change" ) {
                REQUIRE( v.size() == 10 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "the size is reduced" ) {
            v.resize( 0 );

            THEN( "the size changes but not capacity" ) {
                REQUIRE( v.size() == 0 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
        WHEN( "more capacity is reserved" ) {
            v.reserve( 10 );

            THEN( "the capacity changes but not the size" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "less capacity is reserved" ) {
            v.reserve( 0 );

            THEN( "neither size nor capacity are changed" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
    }
}
```

Удобно, эти тесты будут сообщаться следующим образом при запуске:

```
Scenario: vectors can be sized and resized
  Given: A vector with some items
```

```
When: more capacity is reserved  
Then: the capacity changes but not the size
```

Прочитайте Тестирование модулей в С ++ онлайн:

<https://riptutorial.com/ru/cplusplus/topic/9928/тестирование-модулей-в-с-plusplus>

глава 125: Тип Erasure

Вступление

Тип erasure - это набор методов для создания типа, который может обеспечить единый интерфейс для различных базовых типов, при этом скрывая информацию о базовом типе от клиента. `std::function<R(A...)>`, которая имеет возможность удерживать вызываемые объекты разных типов, является, пожалуй, самым известным примером стирания типа в C++.

Examples

Основной механизм

Стирание типа - это способ скрыть тип объекта из его кода, хотя он не является производным от общего базового класса. При этом он обеспечивает мост между мирами статического полиморфизма (шаблоны, в месте использования, точный тип должен быть известен во время компиляции, но его не нужно объявлять соответствующим интерфейсу при определении) и динамический полиморфизм (наследование и виртуальные функции, по месту использования, точный тип не обязательно должен быть известен во время компиляции, но должен быть объявлен соответствующим интерфейсу при определении).

Следующий код показывает основной механизм стирания типа.

```
#include <ostream>

class Printable
{
public:
    template <typename T>
    Printable(T value) : pValue(new Value<T>(value)) {}
    ~Printable() { delete pValue; }
    void print(std::ostream &os) const { pValue->print(os); }

private:
    Printable(Printable const &) /* in C++1x: =delete */; // not implemented
    void operator = (Printable const &) /* in C++1x: =delete */; // not implemented
    struct ValueBase
    {
        virtual ~ValueBase() = default;
        virtual void print(std::ostream &) const = 0;
    };
    template <typename T>
    struct Value : ValueBase
    {
        Value(T const &t) : v(t) {}
        virtual void print(std::ostream &os) const { os << v; }
        T v;
    };
};
```

```
ValueBase *pValue;
};
```

На используемом веб-сайте должно быть видно только указанное выше определение, как и в базовых классах с виртуальными функциями. Например:

```
#include <iostream>

void print_value(Printable const &p)
{
    p.print(std::cout);
}
```

Обратите внимание, что это *не* шаблон, а нормальная функция, которая должна быть объявлена только в файле заголовка и может быть определена в файле реализации (в отличие от шаблонов, определение которых должно быть видимым в месте использования).

При определении конкретного типа ничего не нужно знать о `Printable`, оно просто должно соответствовать интерфейсу, как и с шаблонами:

```
struct MyType { int i; };
ostream& operator << (ostream &os, MyType const &mc)
{
    return os << "MyType {" << mc.i << "}";
}
```

Теперь мы можем передать объект этого класса функции, указанной выше:

```
MyType foo = { 42 };
print_value(foo);
```

Стирание до типа `Regular` с помощью ручной таблицы `vtable`

C++ преуспевает в том, что называется регулярным типом (или, по крайней мере, Pseudo-Regular).

Обычный тип - это тип, который можно создать, назначить и назначить с помощью копирования или перемещения, может быть уничтожен и может сравниваться с равным. Он также может быть построен без аргументов. Наконец, он также поддерживает несколько других операций, которые очень полезны в различных `std` алгоритмах и контейнерах.

[Это корневая бумага](#), но в C++ 11 хотелось бы добавить поддержку `std::hash`.

Здесь я буду использовать ручной подход `vtable` для стирания типа.

```
using dtor_unique_ptr = std::unique_ptr<void, void(*) (void*)>;
```

```

template<class T, class...Args>
dtor_unique_ptr make_dtor_unique_ptr( Args&&... args ) {
    return {new T(std::forward<Args>(args)...), [](void* self){ delete static_cast<T*>(self);
}};
}

struct regular_vtable {
    void(*copy_assign)(void* dest, void const* src); // T&=(T const&)
    void(*move_assign)(void* dest, void* src); // T&=(T&)
    bool(*equals)(void const* lhs, void const* rhs); // T const&==T const&
    bool(*order)(void const* lhs, void const* rhs); // std::less<T>{}(T const&, T const&)
    std::size_t(*hash)(void const* self); // std::hash<T>{}(T const&)
    std::type_info const&(*type)(); // typeid(T)
    dtor_unique_ptr(*clone)(void const* self); // T(T const&)
};

template<class T>
regular_vtable make_regular_vtable() noexcept {
    return {
        [](void* dest, void const* src){ *static_cast<T*>(dest) = *static_cast<T const*>(src); },
        [](void* dest, void* src){ *static_cast<T*>(dest) = std::move(*static_cast<T*>(src)); },
        [](void const* lhs, void const* rhs){ return *static_cast<T const*>(lhs) == *static_cast<T
const*>(rhs); },
        [](void const* lhs, void const* rhs) { return std::less<T>{}(*static_cast<T
const*>(lhs),*static_cast<T const*>(rhs)); },
        [](void const* self){ return std::hash<T>{}(*static_cast<T const*>(self)); },
        []()->decltype(auto){ return typeid(T); },
        [](void const* self){ return make_dtor_unique_ptr<T>(*static_cast<T const*>(self)); }
    };
}

template<class T>
regular_vtable const* get_regular_vtable() noexcept {
    static const regular_vtable vtable=make_regular_vtable<T>();
    return &vtable;
}

struct regular_type {
    using self=regular_type;
    regular_vtable const* vtable = 0;
    dtor_unique_ptr ptr{nullptr, [](void*){}};

    bool empty() const { return !vtable; }

    template<class T, class...Args>
    void emplace( Args&&... args ) {
        ptr = make_dtor_unique_ptr<T>(std::forward<Args>(args)...);
        if (ptr)
            vtable = get_regular_vtable<T>();
        else
            vtable = nullptr;
    }

    friend bool operator==(regular_type const& lhs, regular_type const& rhs) {
        if (lhs.vtable != rhs.vtable) return false;
        return lhs.vtable->equals( lhs.ptr.get(), rhs.ptr.get() );
    }

    bool before(regular_type const& rhs) const {
        auto const& lhs = *this;
        if (!lhs.vtable || !rhs.vtable)
            return std::less<regular_vtable const*>{}(lhs.vtable,rhs.vtable);
        if (lhs.vtable != rhs.vtable)
            return lhs.vtable->type().before(rhs.vtable->type());
        return lhs.vtable->order( lhs.ptr.get(), rhs.ptr.get() );
    }
}

```

```

}
// technically friend bool operator< that calls before is also required

std::type_info const* type() const {
    if (!vtable) return nullptr;
    return &vtable->type();
}
regular_type(regular_type&& o):
    vtable(o.vtable),
    ptr(std::move(o.ptr))
{
    o.vtable = nullptr;
}
friend void swap(regular_type& lhs, regular_type& rhs){
    std::swap(lhs.ptr, rhs.ptr);
    std::swap(lhs.vtable, rhs.vtable);
}
regular_type& operator=(regular_type&& o) {
    if (o.vtable == vtable) {
        vtable->move_assign(ptr.get(), o.ptr.get());
        return *this;
    }
    auto tmp = std::move(o);
    swap(*this, tmp);
    return *this;
}
regular_type(regular_type const& o):
    vtable(o.vtable),
    ptr(o.vtable?o.vtable->clone(o.ptr.get()):dtor_unique_ptr{nullptr, [] (void*){}})
{
    if (!ptr && vtable) vtable = nullptr;
}
regular_type& operator=(regular_type const& o) {
    if (o.vtable == vtable) {
        vtable->copy_assign(ptr.get(), o.ptr.get());
        return *this;
    }
    auto tmp = o;
    swap(*this, tmp);
    return *this;
}
std::size_t hash() const {
    if (!vtable) return 0;
    return vtable->hash(ptr.get());
}
template<class T,
    std::enable_if_t< !std::is_same<std::decay_t<T>, regular_type>{}, int>* =nullptr
>
regular_type(T&& t) {
    emplace<std::decay_t<T>>>(std::forward<T>(t));
}
};

namespace std {
    template<>
    struct hash<regular_type> {
        std::size_t operator()( regular_type const& r )const {
            return r.hash();
        }
    };
    template<>
    struct less<regular_type> {

```

```
bool operator()( regular_type const& lhs, regular_type const& rhs ) const {
    return lhs.before(rhs);
}
};
}
```

живой пример .

Такой регулярный тип может использоваться как ключ для `std::map` или `std::unordered_map` который принимает *что-либо регулярное* для ключа, например:

```
std::map<regular_type, std::any>
```

была бы в основном картой из обычного органа, к чему-либо, скопированному.

В отличие от `any`, мой `regular_type` не оптимизирует мелкие объекты и не поддерживает возврат исходных данных. Получение первоначального типа обратно не сложно.

Небольшая оптимизация объектов требует, чтобы мы храним выровненный буфер для хранения внутри `regular_type`, и тщательно настроить `Deleter` в `ptr` только уничтожить объект, а не удалять его.

Я бы начал с `make_dtor_unique_ptr` и научил его, как иногда хранить данные в буфере, а затем в куче, если в буфере нет места. Этого может быть достаточно.

Только для перемещения `std::function`

`std::function` type стирает до нескольких операций. Одна из вещей, которую он требует, заключается в том, что сохраненное значение можно копировать.

Это вызывает проблемы в нескольких контекстах, таких как `lambdas`, хранящие уникальные `ptrs`. Если вы используете `std::function` в контексте, где копирование не имеет значения, например пул потоков, в котором вы отправляете задачи в потоки, это требование может добавить дополнительные служебные данные.

В частности, `std::packaged_task<Sig>` является вызываемым объектом, который является только перемещением. Вы можете сохранить `std::packaged_task<R(Args...)>` в `std::packaged_task<void(Args...)>`, но это довольно тяжелый и неясный способ создания только для перемещения вызываемый тип-стирание класса.

Таким образом, `task`. Это демонстрирует, как вы можете написать простой тип `std::function`. Я опустил конструктор копирования (который `details::task_pimpl<...>` бы добавление метода `clone` в `details::task_pimpl<...>`).

```
template<class Sig>
struct task;

// putting it in a namespace allows us to specialize it nicely for void return value:
```



```

namespace details {
    template<class R, class...Args>
    struct task_pimpl {
        virtual R invoke(Args&&...args) const = 0;
        virtual ~task_pimpl() {};
        virtual const std::type_info& target_type() const = 0;
    };

    // store an F.  invoke(Args&&...) calls the f
    template<class F, class R, class...Args>
    struct task_pimpl_impl:task_pimpl<R,Args...> {
        F f;
        template<class Fin>
        task_pimpl_impl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
        virtual R invoke(Args&&...args) const final override {
            return f(std::forward<Args>(args)...);
        }
        virtual const std::type_info& target_type() const final override {
            return typeid(F);
        }
    };

    // the void version discards the return value of f:
    template<class F, class...Args>
    struct task_pimpl_impl<F,void,Args...>:task_pimpl<void,Args...> {
        F f;
        template<class Fin>
        task_pimpl_impl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
        virtual void invoke(Args&&...args) const final override {
            f(std::forward<Args>(args)...);
        }
        virtual const std::type_info& target_type() const final override {
            return typeid(F);
        }
    };
};

template<class R, class...Args>
struct task<R(Args...)> {
    // semi-regular:
    task()=default;
    task(task&&)=default;
    // no copy

private:
    // aliases to make some SFINAE code below less ugly:
    template<class F>
    using call_r = std::result_of_t<F const&(Args...)>;
    template<class F>
    using is_task = std::is_same<std::decay_t<F>, task>;
public:
    // can be constructed from a callable F
    template<class F,
        // that can be invoked with Args... and converted-to-R:
        class= decltype( (R)(std::declval<call_r<F>>()) ),
        // and is not this same type:
        std::enable_if_t<!is_task<F>{}, int>* = nullptr
    >
    task(F&& f):
        m_pImpl( make_pimpl(std::forward<F>(f)) )
    {}
};

```

```

// the meat: the call operator
R operator()(Args... args) const {
    return m_pImpl->invoke( std::forward<Args>(args)... );
}
explicit operator bool() const {
    return (bool)m_pImpl;
}
void swap( task& o ) {
    std::swap( m_pImpl, o.m_pImpl );
}
template<class F>
void assign( F&& f ) {
    m_pImpl = make_pimpl(std::forward<F>(f));
}
// Part of the std::function interface:
const std::type_info& target_type() const {
    if (!*this) return typeid(void);
    return m_pImpl->target_type();
}
template< class T >
T* target() {
    return target_impl<T>();
}
template< class T >
const T* target() const {
    return target_impl<T>();
}
// compare with nullptr      :
friend bool operator==( std::nullptr_t, task const& self ) { return !self; }
friend bool operator==( task const& self, std::nullptr_t ) { return !self; }
friend bool operator!=( std::nullptr_t, task const& self ) { return !!self; }
friend bool operator!=( task const& self, std::nullptr_t ) { return !!self; }
private:
template<class T>
using pimpl_t = details::task_pimpl_impl<T, R, Args...>;

template<class F>
static auto make_pimpl( F&& f ) {
    using dF=std::decay_t<F>;
    using pImpl_t = pimpl_t<dF>;
    return std::make_unique<pImpl_t>(std::forward<F>(f));
}
std::unique_ptr<details::task_pimpl<R,Args...>> m_pImpl;

template< class T >
T* target_impl() const {
    return dynamic_cast<pimpl_t<T>*>(m_pImpl.get());
}
};

```

Чтобы сделать эту библиотеку достойной, вы хотите добавить небольшую оптимизацию буфера, поэтому она не хранит все вызываемые в куче.

Добавление SBO потребует нестандартную `task(task&&)` , некоторые `std::aligned_storage_t` в класс, `m_pImpl` `unique_ptr` с `Deleter` , который может быть установлен , чтобы уничтожить только (и не возвращать память в куче), и `emplace_move_to( void* ) = 0` в `task_pimpl` .

[живой пример](#) вышеуказанного кода (без SBO).

Стирание до смежного буфера T

Не все стирание стилей связано с виртуальным наследованием, распределением, размещением новых или даже указателями функций.

Что делает стирание типа стирания стилей, так это то, что он описывает поведение (множество), и принимает любой тип, поддерживающий это поведение и завершающий его. Вся информация, которая не находится в этом наборе поведения, «забыта» или «стерта».

`array_view` принимает свой входящий диапазон или тип контейнера и стирает все, кроме факта, что он является непрерывным буфером T

```
// helper traits for SFINAE:
template<class T>
using data_t = decltype( std::declval<T>().data() );

template<class Src, class T>
using compatible_data = std::integral_constant<bool, std::is_same< data_t<Src>, T* >{} ||
std::is_same< data_t<Src>, std::remove_const_t<T>* >{}>;

template<class T>
struct array_view {
    // the core of the class:
    T* b=nullptr;
    T* e=nullptr;
    T* begin() const { return b; }
    T* end() const { return e; }

    // provide the expected methods of a good contiguous range:
    T* data() const { return begin(); }
    bool empty() const { return begin()==end(); }
    std::size_t size() const { return end()-begin(); }

    T& operator[](std::size_t i)const{ return begin()[i]; }
    T& front()const{ return *begin(); }
    T& back()const{ return *(end()-1); }

    // useful helpers that let you generate other ranges from this one
    // quickly and safely:
    array_view without_front( std::size_t i=1 ) const {
        i = (std::min)(i, size());
        return {begin()+i, end()};
    }
    array_view without_back( std::size_t i=1 ) const {
        i = (std::min)(i, size());
        return {begin(), end()-i};
    }

    // array_view is plain old data, so default copy:
    array_view(array_view const&)=default;
    // generates a null, empty range:
    array_view()=default;

    // final constructor:
```

```

array_view(T* s, T* f):b(s),e(f) {}
// start and length is useful in my experience:
array_view(T* s, std::size_t length):array_view(s, s+length) {}

// SFINAE constructor that takes any .data() supporting container
// or other range in one fell swoop:
template<class Src,
        std::enable_if_t< compatible_data<std::remove_reference_t<Src>&, T >{}, int>* =nullptr,
        std::enable_if_t< !std::is_same<std::decay_t<Src>, array_view >{}, int>* =nullptr
>
array_view( Src&& src ):
    array_view( src.data(), src.size() )
{}

// array constructor:
template<std::size_t N>
array_view( T(&arr)[N] ):array_view(arr, N) {}

// initializer list, allowing {} based:
template<class U,
        std::enable_if_t< std::is_same<const U, T>{}, int>* =nullptr
>
array_view( std::initializer_list<U> il ):array_view(il.begin(), il.end()) {}
};

```

`array_view` принимает любой контейнер, который поддерживает `.data()` возвращающий указатель на `T` и метод `.size()` или массив, и стирает его до диапазона случайного доступа по смежным `T` S.

Он может принимать `std::vector<T>`, `std::string<T>` а `std::array<T, N>` а `T[37]`, список инициализаторов (включая `{}` основанные) или что-то еще вы его поддерживаете (через `T* x.data()` и `size_t x.size()`).

В этом случае данные, которые мы можем извлечь из вещи, которую мы стираем, вместе с нашим «незанятым» представлением «вид» означает, что нам не нужно выделять память или писать пользовательские зависящие от типа функции.

[Живой пример](#) .

Улучшение будет заключаться в использовании `data` не являющихся членами, и `size` не члена в контексте с поддержкой ADL.

Тип стирания стирания стилей с помощью `std::any`

В этом примере используются C ++ 14 и `boost::any` . В C ++ 17 вы можете поменять местами `B std::any` .

Синтаксис, который мы получаем, это:

```

const auto print =
    make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p << "\n"; });

super_any<decltype(print)> a = 7;

```

```
(a->*print) (std::cout);
```

что почти оптимально.

Этот пример основан на работе [@dyp](#) и [@cpplearner](#), а также на моем собственном.

Сначала мы используем тег для пропуска типов:

```
template<class T>struct tag_t{constexpr tag_t(){};};  
template<class T>constexpr tag_t<T> tag{};
```

Этот класс признаков получает подпись, хранящуюся с помощью `any_method`:

Это создает тип указателя функции и фабрику для указанных указателей функций с учетом `any_method`:

```
template<class any_method>  
using any_sig_from_method = typename any_method::signature;  
  
template<class any_method, class Sig=any_sig_from_method<any_method>>  
struct any_method_function;  
  
template<class any_method, class R, class...Args>  
struct any_method_function<any_method, R(Args...)>  
{  
    template<class T>  
    using decorate = std::conditional_t< any_method::is_const, T const, T >;  
  
    using any = decorate<boost::any>;  
  
    using type = R(*) (any&, any_method const*, Args&&...);  
    template<class T>  
    type operator()( tag_t<T> )const{  
        return +[](any& self, any_method const* method, Args&&...args) {  
            return (*method)( boost::any_cast<decorate<T>&>(self), decltype(args)(args)... );  
        };  
    }  
};
```

`any_method_function::type` - это тип указателя на функцию, который мы будем хранить рядом с экземпляром. `any_method_function::operator()` принимает `tag_t<T>` и записывает собственный экземпляр типа `any_method_function::type` который предполагает, что `any&` будет `T`

Мы хотим иметь возможность набирать стирать более одного метода за раз. Поэтому мы объединяем их в кортеж и пишем вспомогательную оболочку, чтобы привязать кортеж к статическому хранилищу для каждого типа и поддерживать указатель на них.

```
template<class...any_methods>  
using any_method_tuple = std::tuple< typename any_method_function<any_methods>::type... >;  
  
template<class...any_methods, class T>
```

```

any_method_tuple<any_methods...> make_vtable( tag_t<T> ) {
    return std::make_tuple(
        any_method_function<any_methods>{}(tag<T>)...
    );
}

template<class...methods>
struct any_methods {
private:
    any_method_tuple<methods...> const* vtable = 0;
    template<class T>
    static any_method_tuple<methods...> const* get_vtable( tag_t<T> ) {
        static const auto table = make_vtable<methods...>(tag<T>);
        return &table;
    }
public:
    any_methods() = default;
    template<class T>
    any_methods( tag_t<T> ): vtable(get_vtable(tag<T>)) {}
    any_methods& operator=(any_methods const&)=default;
    template<class T>
    void change_type( tag_t<T> ={} ) { vtable = get_vtable(tag<T>); }

    template<class any_method>
    auto get_invoker( tag_t<any_method> ={} ) const {
        return std::get<typename any_method_function<any_method>::type>( *vtable );
    }
};

```

Мы могли бы специализировать это для случаев, когда vtable мала (например, 1 элемент), и использовать для этого эффективные указатели с прямым указателем, хранящиеся в классе.

Теперь мы начинаем `super_any`. Я использую `super_any_t` чтобы сделать объявление `super_any` немного проще.

```

template<class...methods>
struct super_any_t;

```

Это ищет методы, которые супер поддерживают для SFINAE и улучшают сообщения об ошибках:

```

template<class super_any, class method>
struct super_method_applies_helper : std::false_type {};

template<class M0, class...Methods, class method>
struct super_method_applies_helper<super_any_t<M0, Methods...>, method> :
    std::integral_constant<bool, std::is_same<M0, method>{} ||
    super_method_applies_helper<super_any_t<Methods...>, method>{}>
{};

template<class...methods, class method>
auto super_method_test( super_any_t<methods...> const&, tag_t<method> )
{
    return std::integral_constant<bool, super_method_applies_helper< super_any_t<methods...>,
    method >{} && method::is_const >{};
}

```

```

}
template<class...methods, class method>
auto super_method_test( super_any_t<methods...>&, tag_t<method> )
{
    return std::integral_constant<bool, super_method_applies_helper< super_any_t<methods...>,
method >{} >{}>;
}

template<class super_any, class method>
struct super_method_applies:
    decltype( super_method_test( std::declval<super_any>(), tag<method> ) )
{};

```

Затем мы создаем тип `any_method`. `any_method` является указателем псевдо-метода. Мы создаем его в глобальном масштабе и `const` ют с использованием синтаксиса , как:

```
const auto print=make_any_method( [](auto&&self, auto&&os){ os << self; } );
```

или в C ++ 17:

```
const any_method print=[](auto&&self, auto&&os){ os << self; };
```

Обратите внимание, что использование не-лямбда может сделать вещи волосатыми, поскольку мы используем тип для этапа поиска. Это можно исправить, но сделать этот пример длиннее, чем он есть. Поэтому всегда инициализируйте любой метод из лямбда или из типа, параметризованного на лямбда.

```

template<class Sig, bool const_method, class F>
struct any_method {
    using signature=Sig;
    enum{is_const=const_method};
private:
    F f;
public:

    template<class Any,
        // SFINAE testing that one of the Anys's matches this type:
        std::enable_if_t< super_method_applies< Any&&, any_method >{}, int>* =nullptr
    >
    friend auto operator->*( Any&& self, any_method const& m ) {
        // we don't use the value of the any_method, because each any_method has
        // a unique type (!) and we check that one of the auto*'s in the super_any
        // already has a pointer to us. We then dispatch to the corresponding
        // any_method_data...

        return [&self, invoke = self.get_invoker(tag<any_method>), m](auto&&...args)-
>decltype(auto)
        {
            return invoke( decltype(self)(self), &m, decltype(args)(args)... );
        };
    }
    any_method( F fin ):f(std::move(fin)) {}

    template<class...Args>
    decltype(auto) operator() (Args&&...args) const {

```

```

    return f(std::forward<Args>(args)...);
}
};

```

Метод фабрики, не нужен в C ++ 17 Я считаю:

```

template<class Sig, bool is_const=false, class F>
any_method<Sig, is_const, std::decay_t<F>>
make_any_method( F&& f ) {
    return {std::forward<F>(f)};
}

```

Это дополненный `any`. Это одновременно `any`, и она несет вокруг пачки указателей функций типа стирания, которые изменяют, когда содержащиеся `any` делают:

```

template<class... methods>
struct super_any_t:boost::any, any_methods<methods...> {
    using vtable=any_methods<methods...>;
public:
    template<class T,
        std::enable_if_t< !std::is_base_of<super_any_t, std::decay_t<T>>{}, int> =0
    >
    super_any_t( T&& t ):
        boost::any( std::forward<T>(t) )
    {
        using dT=std::decay_t<T>;
        this->change_type( tag<dT> );
    }

    boost::any& as_any()&{return *this;}
    boost::any&& as_any()&&{return std::move(*this);}
    boost::any const& as_any()const&{return *this;}
    super_any_t()=default;
    super_any_t(super_any_t&& o):
        boost::any( std::move( o.as_any() ) ),
        vtable(o)
    {}
    super_any_t(super_any_t const& o):
        boost::any( o.as_any() ),
        vtable(o)
    {}
    template<class S,
        std::enable_if_t< std::is_same<std::decay_t<S>, super_any_t>{}, int> =0
    >
    super_any_t( S&& o ):
        boost::any( std::forward<S>(o).as_any() ),
        vtable(o)
    {}
    super_any_t& operator=(super_any_t&&)=default;
    super_any_t& operator=(super_any_t const&)=default;

    template<class T,
        std::enable_if_t< !std::is_same<std::decay_t<T>, super_any_t>{}, int>* =nullptr
    >
    super_any_t& operator=( T&& t ) {
        ((boost::any&)*this) = std::forward<T>(t);
        using dT=std::decay_t<T>;
        this->change_type( tag<dT> );
    }
}

```



```
    return *this;
}
};
```

Поскольку мы сохраняем `any_method` s как объекты `const` , это делает создание `super_any` немного проще:

```
template<class...Ts>
using super_any = super_any_t< std::remove_cv_t<Ts>... >;
```

Тестовый код:

```
const auto print = make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p
<< "\n"; });
const auto wprint = make_any_method<void(std::wostream&)>([](auto&& p, std::wostream& os ){ os
<< p << L"\n"; });

int main()
{
    super_any<decltype(print), decltype(wprint)> a = 7;
    super_any<decltype(print), decltype(wprint)> a2 = 7;

    (a->*print)(std::cout);
    (a->*wprint)(std::wcout);
}
```

[живой пример](#) .

Первоначально опубликовано [здесь](#) в SO вопрос и ответ (и люди, отмеченные выше, помогли с реализацией).

Прочитайте Тип Erasure онлайн: <https://riptutorial.com/ru/cplusplus/topic/2872/тип-erasure>

глава 126: Тип Ключевые слова

Examples

учебный класс

1. Вводит определение типа **класса** .

```
class foo {  
    int x;  
    public:  
    int get_x();  
    void set_x(int new_x);  
};
```

2. Представляет *специфицированный спецификатор типа*, который указывает, что следующее имя - это имя типа класса. Если имя класса уже объявлено, оно может быть найдено, даже если оно скрыто под другим именем. Если имя класса еще не было объявлено, оно объявляется вперед.

```
class foo; // elaborated type specifier -> forward declaration  
class bar {  
    public:  
    bar(foo& f);  
};  
void baz();  
class baz; // another elaborated type specifier; another forward declaration  
           // note: the class has the same name as the function void baz()  
class foo {  
    bar b;  
    friend class baz; // elaborated type specifier refers to the class,  
                     // not the function of the same name  
    public:  
    foo();  
};
```

3. Представляет параметр типа в объявлении **шаблона** .

```
template <class T>  
const T& min(const T& x, const T& y) {  
    return b < a ? b : a;  
}
```

4. В объявлении **параметра шаблона шаблона** `class` ключевого слова предшествует имени параметра. Поскольку аргумент для параметра шаблона шаблона может быть только шаблоном класса, использование `class` здесь избыточно. Однако это требует грамматика C ++.

```
template <template <class T> class U>
```

```
//          ^^^^^ "class" used in this sense here;
//          U is a template template parameter
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

5. Обратите внимание, что смысл 2 и смысл 3 могут быть объединены в одну и ту же декларацию. Например:

```
template <class T>
class foo {
};

foo<class bar> x; // <- bar does not have to have previously appeared.
```

C++ 11

6. В объявлении или определении перечисления объявляет перечисление как [облачное перечисление](#).

```
enum class Format {
    TEXT,
    PDF,
    OTHER,
};
Format f = F::TEXT;
```

структура

Взаимозаменяемый с [class](#), за исключением следующих отличий:

- Если тип класса определяется с помощью ключевого слова `struct`, тогда стандартная доступность баз и членов является `public` а не `private`.
- `struct` не может использоваться для объявления параметра шаблона шаблона или шаблона шаблона; только `class` может.

перечисление

1. Вводит определение типа [перечисления](#).

```
enum Direction {
    UP,
    LEFT,
    DOWN,
    RIGHT
};
Direction d = UP;
```

C++ 11

В C++ 11 `enum` может необязательно сопровождаться `class` или `struct` для определения области с [перечислением](#). Кроме того, как скопированные, так и незанятые перечисления могут иметь свой базовый тип, явно указанный: `T` после имени перечисления, где `T` относится к целочисленному типу.

```
enum class Format : char {
    TEXT,
    PDF,
    OTHER
};
Format f = Format::TEXT;

enum Language : int {
    ENGLISH,
    FRENCH,
    OTHER
};
```

Перечислителям в нормальном `enum` также может предшествовать оператор области видимости, хотя они все еще считаются включенными в область, в которой было указано `enum`.

```
Language l1, l2;

l1 = ENGLISH;
l2 = Language::OTHER;
```

2. Представляет *специфицированный спецификатор типа*, который указывает, что следующее имя - это имя ранее объявленного типа перечисления. (Спецификатор специфицированного типа нельзя использовать для форварда-объявления типа перечисления.) Перечисление можно назвать таким образом, даже если оно скрыто под другим именем.

```
enum Foo { FOO };
void Foo() {}
Foo foo = FOO; // ill-formed; Foo refers to the function
enum Foo foo = FOO; // ok; Foo refers to the enum type
```

C++ 11

3. Вводит *непрозрачную декларацию `enum`*, которая объявляет перечисление без его определения. Он может либо обновить ранее объявленное перечисление, либо переслать-объявить перечисление, которое ранее не было объявлено.

Перечисление, объявленное сначала как область действия, впоследствии не может быть объявлено как незарегистрированное, или *наоборот*. Все объявления перечисления должны согласовываться в базовом типе.

Когда forward-декларирование неперечисленного перечисления, базовый тип должен быть явно указан, поскольку он не может быть выведен до тех пор, пока не будут

известны значения перечислений.

```
enum class Format; // underlying type is implicitly int
void f(Format f);
enum class Format {
    TEXT,
    PDF,
    OTHER,
};

enum Direction;    // ill-formed; must specify underlying type
```

СОЮЗ

1. Вводит определение типа объединения .

```
// Example is from POSIX
union sigval {
    int      sival_int;
    void     *sival_ptr;
};
```

2. Представляет *специфицированный спецификатор типа*, который указывает, что следующее имя - это имя типа объединения. Если имя союза уже объявлено, оно может быть найдено, даже если оно скрыто под другим именем. Если имя профсоюза еще не объявлено, оно объявляется вперед.

```
union foo; // elaborated type specifier -> forward declaration
class bar {
public:
    bar(foo& f);
};
void baz();
union baz; // another elaborated type specifier; another forward declaration
           // note: the class has the same name as the function void baz()
union foo {
    long l;
    union baz* b; // elaborated type specifier refers to the class,
                  // not the function of the same name
};
```

Прочитайте Тип Ключевые слова онлайн: <https://riptutorial.com/ru/cplusplus/topic/7838/тип-ключевые-слова>

глава 127: Типед и псевдонимы типов

Вступление

Параметры `typedef` и (начиная с C ++ 11) `using` **ключевых слов** могут использоваться для обозначения нового имени существующего типа.

Синтаксис

- `typedef type-specifier-seq init-declarator-list ;`
- `attribute-specifier-seq typedef decl-specifier-seq init-declarator-list ;` // с C ++ 11
- используя идентификатор `attribute-specifier-seq ( opt ) = type-id ;` // с C ++ 11

Examples

Основной синтаксис typedef

Объявление `typedef` имеет тот же синтаксис, что и объявление переменной или функции, но содержит слово `typedef`. Присутствие `typedef` заставляет объявление объявлять тип вместо переменной или функции.

```
int T;           // T has type int
typedef int T;   // T is an alias for int

int A[100];      // A has type "array of 100 ints"
typedef int A[100]; // A is an alias for the type "array of 100 ints"
```

Как только псевдоним типа определен, его можно использовать взаимозаменяемо с исходным именем типа.

```
typedef int A[100];
// S is a struct containing an array of 100 ints
struct S {
    A data;
};
```

`typedef` никогда не создает отдельный тип. Это дает только другой способ сослаться на существующий тип.

```
struct S {
    int f(int);
};
typedef int I;
// ok: defines int S::f(int)
I S::f(I x) { return x; }
```

Более сложное использование typedef

Правило о том, что объявления `typedef` имеют тот же синтаксис, что и обычные объявления переменных и функций, могут использоваться для чтения и записи более сложных объявлений.

```
void (*f)(int);           // f has type "pointer to function of int returning void"
typedef void (*f)(int);    // f is an alias for "pointer to function of int returning void"
```

Это особенно полезно для конструкций с запутанным синтаксисом, таких как указатели на нестатические элементы.

```
void (Foo::*pmf)(int);      // pmf has type "pointer to member function of Foo taking int
                             // and returning void"
typedef void (Foo::*pmf)(int); // pmf is an alias for "pointer to member function of Foo
                             // taking int and returning void"
```

Трудно вспомнить синтаксис следующих объявлений функций, даже для опытных программистов:

```
void (Foo::*f(const char*)) (int);
int (&g()) [100];
```

`typedef` может использоваться для упрощения чтения и записи:

```
typedef void (Foo::pmf)(int); // pmf is a pointer to member function type
pmf Foo::f(const char*);      // f is a member function of Foo

typedef int (&ra)[100];        // ra means "reference to array of 100 ints"
ra g();                       // g returns reference to array of 100 ints
```

Объявление нескольких типов с помощью typedef

Ключевое слово `typedef` является спецификатором, поэтому оно применяется отдельно для каждого декларатора. Следовательно, каждое объявленное имя относится к типу, которое это имя имело бы при отсутствии `typedef`.

```
int *x, (*p)();           // x has type int*, and p has type int(*)()
typedef int *x, (*p)();    // x is an alias for int*, while p is an alias for int(*)()
```

Объявление псевдонима с "использованием"

C++ 11

Синтаксис `using` очень прост: имя, которое должно быть определено, идет с левой стороны, а определение - с правой стороны. Не нужно сканировать, чтобы узнать, где это имя.

```
using I = int;
using A = int[100];           // array of 100 ints
using FP = void(*) (int);     // pointer to function of int returning void
using MP = void (Foo::*) (int); // pointer to member function of Foo of int returning void
```

Создание псевдонима типа с `using` имеет тот же эффект, что и создание псевдонима типа с `typedef` . Это просто альтернативный синтаксис для достижения того же.

В отличие от `typedef` , `using` может быть шаблоном. «Шаблон `typedef`», созданный с `using` , называется **шаблоном псевдонимов** .

Прочитайте Типед и псевдонимы типов онлайн: <https://riptutorial.com/ru/cplusplus/topic/9328/типед-и-псевдонимы-типов>

глава 128: Типовые черты

замечания

Типовыми признаками являются шаблонные конструкции, используемые для сравнения и тестирования свойств разных типов во время компиляции. Они могут использоваться для обеспечения условной логики во время компиляции, которая может ограничить или расширить функциональность вашего кода определенным образом. Библиотека признаков типов была введена с помощью стандарта `c++11` который предоставляет несколько разных функций. Также возможно создать собственные шаблоны сравнения типов типов.

Examples

Стандартные типы

C ++ 11

Заголовок `type_traits` содержит набор классов шаблонов и помощников для преобразования и проверки свойств типов во время компиляции.

Эти черты обычно используются в шаблонах для проверки ошибок пользователя, поддержки общего программирования и обеспечения оптимизации.

Большинство типов признаков используются для проверки того, соответствует ли тип определенным критериям. Они имеют следующий вид:

```
template <class T> struct is_foo;
```

Если класс шаблона создается с типом, который удовлетворяет некоторым критериям `foo` , то `is_foo<T>` наследует от `std::integral_constant<bool,true>` (aka `std::true_type`), в противном случае он наследует от `std::integral_constant<bool,false>` (aka `std::false_type`). Это дает признак следующим членам:

Константы

```
static constexpr bool value
```

`true` если `T` удовлетворяет критериям `foo` , `false` противном случае

функции

operator bool

Возвращает value

C ++ 14

bool operator()

Возвращает value

Типы

название	Определение
value_type	bool
type	std::integral_constant<bool,value>

Затем этот признак можно использовать в конструкциях, таких как `static_assert` или `std::enable_if`. Пример с `std::is_pointer`:

```
template <typename T>
void i_require_a_pointer (T t) {
    static_assert(std::is_pointer<T>::value, "T must be a pointer type");
}

//Overload for when T is not a pointer type
template <typename T>
typename std::enable_if<!std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something boring
}

//Overload for when T is a pointer type
template <typename T>
typename std::enable_if<std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something special
}
```

Существуют также различные черты, которые преобразуют типы, такие как `std::add_pointer` и `std::underlying_type` `std::add_pointer`. Эти признаки обычно выставляют `type` типа одного типа, который содержит преобразованный тип. Например, `std::add_pointer<int>::type` is `int*`.

Тип отношений с `std :: is_same`

C ++ 11

Отношение `std::is_same<T, T>` используется для сравнения двух типов. Он будет

оцениваться как логический, true, если типы являются одинаковыми и ложными, если в противном случае.

например

```
// Prints true on most x86 and x86_64 compilers.
std::cout << std::is_same<int, int32_t>::value << "\n";
// Prints false on all compilers.
std::cout << std::is_same<float, int>::value << "\n";
// Prints false on all compilers.
std::cout << std::is_same<unsigned int, int>::value << "\n";
```

Отношение `std::is_same` также будет работать независимо от typedefs. Это фактически продемонстрировано в первом примере при сравнении `int == int32_t` однако это не совсем ясно.

например

```
// Prints true on all compilers.
typedef int MyType
std::cout << std::is_same<int, MyType>::value << "\n";
```

Использование `std::is_same` для предупреждения при неправильном использовании шаблонного класса или функции.

В сочетании со статическим утверждением шаблон `std::is_same` может быть ценным инструментом для обеспечения правильного использования шаблонных классов и функций.

например, функция, которая допускает ввод только от `int` и выбор двух структур.

```
#include <type_traits>
struct foo {
    int member;
    // Other variables
};

struct bar {
    char member;
};

template<typename T>
int AddStructMember(T var1, int var2) {
    // If type T != foo || T != bar then show error message.
    static_assert(std::is_same<T, foo>::value ||
        std::is_same<T, bar>::value,
        "This function does not support the specified type.");
    return var1.member + var2;
}
```

Основные черты характера

C++ 11

Существует несколько различных типов признаков, которые сравнивают более общие типы.

Интеграл:

Оценивает как true для всех целых типов `int` , `char` , `long` , `unsigned int` и т. Д.

```
std::cout << std::is_integral<int>::value << "\n"; // Prints true.
std::cout << std::is_integral<char>::value << "\n"; // Prints true.
std::cout << std::is_integral<float>::value << "\n"; // Prints false.
```

Плавающая точка:

Вычисляет значение true для всех типов с плавающей запятой. `float` , `double` , `long double` и т. Д.

```
std::cout << std::is_floating_point<float>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<double>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<char>::value << "\n"; // Prints false.
```

Is Enum:

Вычисляет значение true для всех перечисленных типов, включая `enum class` .

```
enum fruit {apple, pair, banana};
enum class vegetable {carrot, spinach, leek};
std::cout << std::is_enum<fruit>::value << "\n"; // Prints true.
std::cout << std::is_enum<vegetable>::value << "\n"; // Prints true.
std::cout << std::is_enum<int>::value << "\n"; // Prints false.
```

Является указателем:

Оценивает как истину для всех указателей.

```
std::cout << std::is_pointer<int *>::value << "\n"; // Prints true.
typedef int* MyPTR;
std::cout << std::is_pointer<MyPTR>::value << "\n"; // Prints true.
std::cout << std::is_pointer<int>::value << "\n"; // Prints false.
```

Класс:

Оценивает как true для всех классов и `struct`, за исключением `enum class` .

```
struct FOO {int x, y};
class BAR {
public:
    int x, y;
};
enum class fruit {apple, pair, banana};
std::cout << std::is_class<FOO>::value << "\n"; // Prints true.
std::cout << std::is_class<BAR>::value << "\n"; // Prints true.
```

```
std::cout << std::is_class<fruit>::value << "\n"; // Prints false.
std::cout << std::is_class<int>::value << "\n"; // Prints false.
```

Свойства типа

C ++ 11

Свойства типа сравнивают модификаторы, которые могут быть помещены на разные переменные. Полезность этих черт типа не всегда очевидна.

Примечание. В приведенном ниже примере можно было бы предложить улучшения для не оптимизирующего компилятора. Это просто доказательство концепции, а не сложный пример.

например, Быстрое разделение на четыре.

```
template<typename T>
inline T FastDivideByFour(const T &var) {
    // Will give an error if the inputted type is not an unsigned integral type.
    static_assert(std::is_unsigned<T>::value && std::is_integral<T>::value,
        "This function is only designed for unsigned integral types.");
    return (var >> 2);
}
```

Постоянно:

Это будет считаться истинным, если тип является постоянным.

```
std::cout << std::is_const<const int>::value << "\n"; // Prints true.
std::cout << std::is_const<int>::value << "\n"; // Prints false.
```

Неустойчиво:

Это будет считаться истинным, когда тип изменчивый.

```
std::cout << std::is_volatile<static volatile int>::value << "\n"; // Prints true.
std::cout << std::is_const<const int>::value << "\n"; // Prints false.
```

Подписан:

Это будет считаться истинным для всех подписанных типов.

```
std::cout << std::is_signed<int>::value << "\n"; // Prints true.
std::cout << std::is_signed<float>::value << "\n"; // Prints true.
std::cout << std::is_signed<unsigned int>::value << "\n"; // Prints false.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints false.
```

Неподписано:

Будет оцениваться как истинное для всех неподписанных типов.

```
std::cout << std::is_unsigned<unsigned int>::value << "\n"; // Prints true.  
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints true.  
std::cout << std::is_unsigned<int>::value << "\n"; // Prints false.  
std::cout << std::is_signed<float>::value << "\n"; // Prints false.
```

Прочитайте Типовые черты онлайн: <https://riptutorial.com/ru/cplusplus/topic/4750/типовые-черты>

глава 129: указатели

Вступление

Указатель - это адрес, который относится к местоположению в памяти. Они обычно используются, чтобы позволить функциям или структурам данных знать и изменять память без необходимости копировать упомянутую память. Указатели можно использовать как с примитивными (встроенными), так и с пользовательскими типами.

Указатели используют «разыменование» *, «адрес операторов» & и «arrow» -> . Операторы '*' и '->' используются для доступа к выделенной памяти, а оператор & используется для получения адреса в памяти.

Синтаксис

- <Тип данных> * <Имя переменной>;
- <Тип данных> * <Имя переменной> = & <Переменное имя одного и того же типа данных>;
- <Тип данных> * <Имя переменной> = <Значение одного и того же типа данных>;
- int * foo; // Указатель, который указывает на целочисленное значение
- int * bar = & myIntVar;
- long * bar [2];
- long * bar [] = {& myLongVar1, & myLongVar2}; // Равно: long * bar [2]

замечания

Помните о проблемах при объявлении нескольких указателей в одной строке.

```
int* a, b, c; //Only a is a pointer, the others are regular ints.

int* a, *b, *c; //These are three pointers!

int *foo[2]; //Both *foo[0] and *foo[1] are pointers.
```

Examples

Основы указателя

C ++ 11

Примечание: во всех следующих случаях предполагается существование константы `nullptr` C ++ 11 ++. Для более ранних версий замените `nullptr` на `NULL` , константу, которая использовалась для аналогичной роли.

Создание указательной переменной

Переменная указателя может быть создана с использованием специального синтаксиса `*`, например `int *pointer_to_int;`,

Когда переменная имеет *тип указателя* (`int *`), она просто содержит адрес памяти. Адрес памяти - это местоположение, в котором хранятся данные *базового типа* (`int`).

Разница очевидна при сравнении размера переменной с размером указателя на один и тот же тип:

```
// Declare a struct type `big_struct` that contains
// three long long ints.
typedef struct {
    long long int foo1;
    long long int foo2;
    long long int foo3;
} big_struct;

// Create a variable `bar` of type `big_struct`
big_struct bar;
// Create a variable `p_bar` of type `pointer to big_struct`.
// Initialize it to `nullptr` (a null pointer).
big_struct *p_bar0 = nullptr;

// Print the size of `bar`
std::cout << "sizeof(bar) = " << sizeof(bar) << std::endl;
// Print the size of `p_bar`.
std::cout << "sizeof(p_bar0) = " << sizeof(p_bar0) << std::endl;

/* Produces:
    sizeof(bar) = 24
    sizeof(p_bar0) = 8
*/
```

Принимая адрес другой переменной

Указатели могут быть назначены между собой так же, как и обычные переменные; в этом случае это **адрес памяти**, который копируется из одного указателя в другой, а **не фактические данные**, на которые указывает указатель.

Более того, они могут принимать значение `nullptr` которое представляет собой нулевую ячейку памяти. Указатель, равный `nullptr` содержит недопустимую ячейку памяти и, следовательно, не относится к действительным данным.

Вы можете получить адрес памяти переменной данного типа, префиксной переменной с *адресом* оператора `&`. Значение, возвращаемое символом `&` является указателем на базовый тип, который содержит адрес памяти переменной (который является допустимыми данными **до тех пор, пока переменная не выходит за рамки**).


```
// Copy `p_bar0` into `p_bar1`.
big_struct *p_bar1 = p_bar0;

// Take the address of `bar` into `p_bar2`
big_struct *p_bar2 = &bar;

// p_bar1 is now nullptr, p_bar2 is &bar.

p_bar0 = p_bar2;

// p_bar0 is now &bar.

p_bar2 = nullptr;

// p_bar0 == &bar
// p_bar1 == nullptr
// p_bar2 == nullptr
```

В отличие от ссылок:

- назначение двух указателей не перезаписывает память, к которой относится назначенный указатель;
- указатели могут быть нулевыми.
- *адрес* оператора требуется явно.

Доступ к контенту указателя

Поскольку для принятия адреса требуется `&`, а для доступа к контенту требуется использование *оператора разыменования* `*` в качестве префикса. Когда указатель разыменовывается, он становится переменной базового типа (фактически, ссылкой на него). Затем он может быть прочитан и изменен, если не `const`.

```
(*p_bar0).foo1 = 5;

// `p_bar0` points to `bar`. This prints 5.
std::cout << "bar.foo1 = " << bar.foo1 << std::endl;

// Assign the value pointed to by `p_bar0` to `baz`.
big_struct baz;
baz = *p_bar0;

// Now `baz` contains a copy of the data pointed to by `p_bar0`.
// Indeed, it contains a copy of `bar`.

// Prints 5 as well
std::cout << "baz.foo1 = " << baz.foo1 << std::endl;
```

Сочетание `*` и оператора `.` сокращенно `->`:

```
std::cout << "bar.foo1 = " << (*p_bar0).foo1 << std::endl; // Prints 5
std::cout << "bar.foo1 = " << p_bar0->foo1 << std::endl; // Prints 5
```

Выделение недействительных указателей

При разыменовании указателя вы должны убедиться, что он указывает на действительные данные. Разделение недопустимого указателя (или нулевого указателя) может привести к нарушению доступа к памяти или чтению или записи данных мусора.

```
big_struct *never_do_this() {
    // This is a local variable. Outside `never_do_this` it doesn't exist.
    big_struct retval;
    retval.foo1 = 11;
    // This returns the address of `retval`.
    return &retval;
    // `retval` is destroyed and any code using the value returned
    // by `never_do_this` has a pointer to a memory location that
    // contains garbage data (or is inaccessible).
}
```

В таком сценарии `g++` и `clang++` корректно выдают предупреждения:

```
(Clang) warning: address of stack memory associated with local variable 'retval' returned [-Wreturn-stack-address]
(Gcc)   warning: address of local variable `retval' returned [-Wreturn-local-addr]
```

Следовательно, следует проявлять осторожность, когда указатели являются аргументами функций, поскольку они могут быть нулевыми:

```
void naive_code(big_struct *ptr_big_struct) {
    // ... some code which doesn't check if `ptr_big_struct` is valid.
    ptr_big_struct->foo1 = 12;
}

// Segmentation fault.
naive_code(nullptr);
```

Операции указателя

Для указателей есть два оператора: Address-of operator (&): возвращает адрес памяти своего операнда. Оператор Contents-of (Dereference) (*): Возвращает значение переменной, расположенную по адресу, указанному его оператором.

```
int var = 20;
int *ptr;
ptr = &var;

cout << var << endl;
//Outputs 20 (The value of var)

cout << ptr << endl;
//Outputs 0x234f119 (var's memory location)

cout << *ptr << endl;
```

```
//Outputs 20(The value of the variable stored in the pointer ptr
```

Звездочка (*) используется при объявлении указателя для простой цели указывать, что это указатель. Не путайте это с оператором **разыменования**, который используется для получения значения, расположенного по указанному адресу. Это просто две разные вещи, представленные одним и тем же знаком.

Арифметика указателей

Приращение / уменьшение

Указатель может быть увеличен или уменьшен (префикс и постфикс). Приращение указателя продвигает значение указателя к элементу в массиве одним элементом за текущий элемент, указанный в данный момент. Уменьшение указателя перемещает его в предыдущий элемент массива.

Арифметика указателя не допускается, если тип, на который указывает указатель, не завершен. `void` всегда является неполным типом.

```
char* str = new char[10]; // str = 0x010
++str;                    // str = 0x011 in this case sizeof(char) = 1 byte

int* arr = new int[10];   // arr = 0x00100
++arr;                    // arr = 0x00104 if sizeof(int) = 4 bytes

void* ptr = (void*)new char[10];
++ptr;    // void is incomplete.
```

Если указатель на конечный элемент увеличивается, то указатель указывает на один элемент за конец массива. Такой указатель не может быть разыменован, но он может быть уменьшен.

Увеличение указателя на один из прошедших элементов в массиве или уменьшение указателя на первый элемент в массиве приводит к неопределенному поведению.

Указатель на объект без массива можно обрабатывать для арифметики указателя, как если бы это был массив размером 1.

Добавление / вычитание

Целочисленные значения могут быть добавлены к указателям; они действуют как приращение, а не на определенное число, а не на 1. Целочисленные значения также можно вычесть из указателей, действуя как декремент указателя. Как и при добавлении / уменьшении, указатель должен указывать на полный тип.

```
char* str = new char[10]; // str = 0x010
str += 2;                // str = 0x010 + 2 * sizeof(char) = 0x012

int* arr = new int[10];  // arr = 0x100
arr += 2;                // arr = 0x100 + 2 * sizeof(int) = 0x108, assuming sizeof(int) == 4.
```

Различие в указателях

Можно вычислить разницу между двумя указателями на один и тот же тип. Два указателя должны быть в пределах одного объекта массива; в противном случае результаты неопределенного поведения.

Учитывая два указателя P и Q в одном массиве, если P - i й элемент в массиве, а Q - j й элемент, то $P - Q$ будет $i - j$. Тип результата: `std::ptrdiff_t`, from `<cstdlib>`.

```
char* start = new char[10]; // str = 0x010
char* test = &start[5];
std::ptrdiff_t diff = test - start; //Equal to 5.
std::ptrdiff_t diff = start - test; //Equal to -5; ptrdiff_t is signed.
```

Прочитайте указатели онлайн: <https://riptutorial.com/ru/cplusplus/topic/3056/указатели>

глава 130: Указатели для участников

Синтаксис

- Предполагая класс с именем `Class` ...
 - `type * ptr = & Class :: member;` // Указывать только на статические члены
 - `type Class :: * ptr = & Class :: member;` // Указываем нестатические члены класса
- Для указателей на нестатические члены класса, учитывая следующие два определения:
 - Экземпляр класса;
 - Класс `* p = & instance;`
- Указатели на переменные класса
 - `ptr = & Class :: i;` // Указывать на переменную `i` внутри каждого класса
 - `instance. * ptr = 1;` // Доступ к экземпляру экземпляра `i`
 - `p -> * ptr = 1;` // Доступ к `p i`
- Указатели на функции члена класса
 - `ptr = & Класс :: F;` // Точка к функции 'F' внутри каждого класса
 - `(. * Экземпляр PTR) (5);` // Вызов экземпляра F
 - `(P -> * PTR) (6);` // Вызов `p's F`

Examples

Указатели на статические функции-члены

`static` функция-член аналогична обычной функции C / C ++, за исключением области:

- Он находится внутри `class`, поэтому ему нужно его имя, украшенное именем класса;
- Он имеет доступ к `public`, `protected` или `private`.

Итак, если у вас есть доступ к `static` функции-члену и правильно ее украсить, вы можете указать на функцию, как на любую нормальную функцию вне `class`:

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

// Note that MyFn() is of type 'Fn'
int MyFn(int i) { return 2*i; }

class Class {
public:
```

```

    // Note that Static() is of type 'Fn'
    static int Static(int i) { return 3*i; }
}; // Class

int main() {
    Fn *fn;    // fn is a pointer to a type-of Fn

    fn = &MyFn;        // Point to one function
    fn(3);              // Call it
    fn = &Class::Static; // Point to the other function
    fn(4);              // Call it
} // main()

```

Указатели на функции-члены

Чтобы получить доступ к функции-члену класса, вам нужно иметь «дескриптор» для конкретного экземпляра, как самого экземпляра, так и указателя или ссылки на него. Учитывая экземпляр класса, вы можете указать на различные его члены с указателем на член, если вы правильно поняли синтаксис! Конечно, указатель должен быть объявлен того же типа, что и указатель на ...

```

typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

class Class {
public:
    // Note that A() is of type 'Fn'
    int A(int a) { return 2*a; }
    // Note that B() is of type 'Fn'
    int B(int b) { return 3*b; }
}; // Class

int main() {
    Class c;          // Need a Class instance to play with
    Class *p = &c;    // Need a Class pointer to play with

    Fn Class::*fn;    // fn is a pointer to a type-of Fn within Class

    fn = &Class::A;   // fn now points to A within any Class
    (c.*fn)(5);       // Pass 5 to c's function A (via fn)
    fn = &Class::B;   // fn now points to B within any Class
    (p->*fn)(6);       // Pass 6 to c's (via p) function B (via fn)
} // main()

```

В отличие от указателей на переменные-члены (в предыдущем примере), связь между экземпляром класса и указателем-членом должна быть тесно связана с круглыми скобками, что выглядит немного странным (как будто .* И ->* не странны довольно!)

Указатели на переменные-члены

Чтобы получить доступ к члену `class`, вам нужно иметь «дескриптор» для конкретного экземпляра, как самого экземпляра, так и указателя или ссылки на него. Учитывая экземпляр `class`, вы можете указать на различные его члены с указателем на член, если вы правильно поняли синтаксис! Конечно, указатель должен быть объявлен того же типа, что

и указатель на ...

```
class Class {
public:
    int x, y, z;
    char m, n, o;
}; // Class

int x; // Global variable

int main() {
    Class c;           // Need a Class instance to play with
    Class *p = &c;     // Need a Class pointer to play with

    int *p_i;          // Pointer to an int

    p_i = &x;           // Now pointing to x
    p_i = &c.x;         // Now pointing to c's x

    int Class::*p_C_i; // Pointer to an int within Class

    p_C_i = &Class::x; // Point to x within any Class
    int i = c.*p_C_i;  // Use p_C_i to fetch x from c's instance
    p_C_i = &Class::y; // Point to y within any Class
    i = c.*p_C_i;      // Use p_C_i to fetch y from c's instance

    p_C_i = &Class::m; // ERROR! m is a char, not an int!

    char Class::*p_C_c = &Class::m; // That's better...
} // main()
```

Синтаксис указателя на член требует некоторых дополнительных синтаксических элементов:

- Чтобы определить тип указателя, вам нужно указать базовый тип, а также тот факт, что он находится внутри класса: `int Class::*ptr; ,`
- Если у вас есть класс или ссылка и хотите использовать его с указателем на члене, вы должны использовать `.*` Оператор (родственный `.` Оператор).
- Если у вас есть указатель на класс и вы хотите использовать его с указателем на член, вам нужно использовать оператор `->*` (сродни оператору `->`).

Указатели на статические переменные-члены

`static` переменная-член как обычная переменная C / C ++, за исключением области:

- Он находится внутри `class` , поэтому ему нужно его имя, украшенное именем класса;
- Он имеет доступ к `public` , `protected` или `private` .

Итак, если у вас есть доступ к `static` переменной-члену и правильно ее украсить, вы можете указать на переменную как на любую нормальную переменную вне `class` :

```
class Class {
public:
```

```
    static int i;
}; // Class

int Class::i = 1; // Define the value of i (and where it's stored!)

int j = 2;    // Just another global variable

int main() {
    int k = 3; // Local variable

    int *p;

    p = &k;    // Point to k
    *p = 2;    // Modify it
    p = &j;    // Point to j
    *p = 3;    // Modify it
    p = &Class::i; // Point to Class::i
    *p = 4;    // Modify it
} // main()
```

Прочитайте Указатели для участников онлайн: <https://riptutorial.com/ru/cplusplus/topic/2130/указатели-для-участников>

глава 131: Умные указатели

Синтаксис

- `std::shared_ptr<ClassType> variableName = std::make_shared<ClassType>(arg1, arg2, ...);`
- `std::shared_ptr<ClassType> variableName (new ClassType(arg1, arg2, ...));`
- `std::weak_ptr<ClassType> variableName = std::make_weak_ptr<ClassType>(arg1, arg2, ...); // С ++ 14`
- `std::weak_ptr<ClassType> variableName (new ClassType(arg1, arg2, ...));`

замечания

С ++ не является языком, управляемым памятью. Динамически выделенная память (т. Е. Объекты, созданные `new`) будет «утечка», если она явно не освобождена (с `delete`). Ответственность программиста заключается в том, чтобы освободить динамически выделенную память, прежде чем отбрасывать последний указатель на этот объект.

Умные указатели могут использоваться для автоматического управления областью динамически распределенной памяти (т. Е. Когда последняя ссылка указателя выходит за пределы области, она удаляется).

В большинстве случаев интеллектуальные указатели предпочтительнее «сырых» указателей. Они делают семантику владения динамически выделенной явной памятью, передавая в своих именах, должен ли объект быть общим или уникальным.

Используйте `#include <memory>` , чтобы использовать интеллектуальные указатели.

Examples

Совместное использование собственности (`std::shared_ptr`)

Шаблон класса `std::shared_ptr` определяет общий указатель, который может делиться правами собственности на объект с другими общими указателями. Это контрастирует с `std::weak_ptr` который представляет собой эксклюзивное право собственности.

Поведение совместного доступа реализуется с помощью метода, известного как подсчет ссылок, при котором количество общих указателей, указывающих на объект, сохраняется вместе с ним. Когда это число достигает нуля, либо путем уничтожения или переназначения последнего экземпляра `std::shared_ptr` , объект автоматически уничтожается.

```
// Creation: 'firstShared' is a shared pointer for a new instance of 'Foo'
std::shared_ptr<Foo> firstShared = std::make_shared<Foo>(/*args*/);
```

Чтобы создать несколько интеллектуальных указателей, совместно использующих один и тот же объект, нам нужно создать еще один `shared_ptr` который сглаживает первый общий указатель. Вот два способа сделать это:

```
std::shared_ptr<Foo> secondShared(firstShared); // 1st way: Copy constructing
std::shared_ptr<Foo> secondShared;
secondShared = firstShared; // 2nd way: Assigning
```

Любой из указанных способов делает `secondShared` общим указателем, который разделяет право собственности на наш экземпляр `Foo` с `firstShared`.

Умный указатель работает так же, как исходный указатель. Это означает, что вы можете использовать `*` чтобы разыменовать их. Оператор `regular ->` работает также:

```
secondShared->test(); // Calls Foo::test()
```

Наконец, когда последний `aliased shared_ptr` выходит из области видимости, вызывается деструктор нашего экземпляра `Foo`.

Предупреждение. При создании `shared_ptr` может `bad_alloc` исключение `bad_alloc` когда необходимо выделить дополнительные данные для семантики совместного использования. Если конструктор передается регулярным указателем, он предполагает, что ему принадлежит объект, на который указывает, и вызывает `delete`, если генерируется исключение. Это означает, что `shared_ptr<T>(new T(args))` не будет утечка объекта `T` если выделение `shared_ptr<T>` не выполняется. Тем не менее, рекомендуется использовать `make_shared<T>(args)` или `make_shared<T>(args) allocate_shared<T>(alloc, args)`, которые позволяют реализации оптимизировать распределение памяти.

Выделение массивов ([]) с использованием `shared_ptr`

C++11 C++17

К сожалению, нет прямого способа выделения массивов с использованием `make_shared<>`.

Возможно создание массивов для `shared_ptr<>` с использованием `new` и `std::default_delete`.

Например, чтобы выделить массив из 10 целых чисел, мы можем написать код как

```
shared_ptr<int> sh(new int[10], std::default_delete<int[]>());
```

Указание `std::default_delete` является обязательным здесь, чтобы убедиться, что выделенная память правильно очищена с помощью `delete[]`.

Если мы знаем размер во время компиляции, мы можем сделать это следующим образом:

```

template<class Arr>
struct shared_array_maker {};
template<class T, std::size_t N>
struct shared_array_maker<T[N]> {
    std::shared_ptr<T> operator()const{
        auto r = std::make_shared<std::array<T,N>>();
        if (!r) return {};
        return {r.data(), r};
    }
};
template<class Arr>
auto make_shared_array()
-> decltype( shared_array_maker<Arr>{}() )
{ return shared_array_maker<Arr>{}(); }

```

То `make_shared_array<int[10]>` возвращает `shared_ptr<int>` указывающий на 10 ints по умолчанию.

C ++ 17

С C ++ 17 `shared_ptr` получила особую поддержку для типов массивов. Больше нет необходимости явно указывать элемент управления массивом, а общий указатель может быть разыменован с помощью оператора индекса массива `[]` :

```

std::shared_ptr<int[]> sh(new int[10]);
sh[0] = 42;

```

Общие указатели могут указывать на под-объект объекта, которому он владеет:

```

struct Foo { int x; };
std::shared_ptr<Foo> p1 = std::make_shared<Foo>();
std::shared_ptr<int> p2(p1, &p1->x);

```

Оба `p2` и `p1` принадлежат объекту типа `Foo` , но `p2` указывает на его `int` member `x` . Это означает, что если `p1` выходит за пределы области действия или переназначается, основной объект `Foo` будет по-прежнему оставаться в живых, гарантируя, что `p2` не болтается.

Важно: `shared_ptr` знает только о себе и обо всех других `shared_ptr` которые были созданы с помощью конструктора псевдонимов. Он не знает о каких-либо других указателях, включая все другие `shared_ptr` s, созданные со ссылкой на тот же экземпляр `Foo` :

```

Foo *foo = new Foo;
std::shared_ptr<Foo> shared1(foo);
std::shared_ptr<Foo> shared2(foo); // don't do this

shared1.reset(); // this will delete foo, since shared1
                // was the only shared_ptr that owned it

shared2->test(); // UNDEFINED BEHAVIOR: shared2's foo has been

```

```
// deleted already!!
```

Передача собственности `shared_ptr`

По умолчанию `shared_ptr` увеличивает счетчик ссылок и не передает права собственности. Тем не менее, это может быть сделано для передачи права собственности с помощью `std::move`:

```
shared_ptr<int> up = make_shared<int>();  
// Transferring the ownership  
shared_ptr<int> up2 = move(up);  
// At this point, the reference count of up = 0 and the  
// ownership of the pointer is solely with up2 with reference count = 1
```

Совместное использование с временным участием (`std::weak_ptr`)

Экземпляры `std::weak_ptr` могут указывать на объекты, принадлежащие экземплярам `std::shared_ptr` а сами становятся самими временными собственниками. Это означает, что слабые указатели не изменяют счетчик ссылок объекта и поэтому не препятствуют удалению объекта, если все общие указатели объекта переназначены или уничтожены.

В следующем примере экземпляры `std::weak_ptr` используются так, что уничтожение древовидного объекта не блокируется:

```
#include <memory>  
#include <vector>  
  
struct TreeNode {  
    std::weak_ptr<TreeNode> parent;  
    std::vector< std::shared_ptr<TreeNode> > children;  
};  
  
int main() {  
    // Create a TreeNode to serve as the root/parent.  
    std::shared_ptr<TreeNode> root(new TreeNode);  
  
    // Give the parent 100 child nodes.  
    for (size_t i = 0; i < 100; ++i) {  
        std::shared_ptr<TreeNode> child(new TreeNode);  
        root->children.push_back(child);  
        child->parent = root;  
    }  
  
    // Reset the root shared pointer, destroying the root object, and  
    // subsequently its child nodes.  
    root.reset();  
}
```

Как дочерние узлы добавляются к детям корневого узла, их `std::weak_ptr` член `parent` устанавливается в корневой узел. `parent` член объявляется как слабый указатель, а не

общий указатель, так что счетчик ссылок корневого узла не увеличивается. Когда корневой узел сбрасывается в конце `main()`, корень уничтожается. Так как единственные оставшиеся ссылки `std::shared_ptr` на дочерние узлы содержались в дочерних элементах корневого `children`, все дочерние узлы впоследствии также уничтожаются.

Из-за деталей реализации блока управления выделенная память `shared_ptr` может не быть выпущена до тех пор, пока счетчик ссылок `shared_ptr` и `weak_ptr` ссылок `weak_ptr` достигнут нуля.

```
#include <memory>
int main()
{
    {
        std::weak_ptr<int> wk;
        {
            // std::make_shared is optimized by allocating only once
            // while std::shared_ptr<int>(new int(42)) allocates twice.
            // Drawback of std::make_shared is that control block is tied to our integer
            std::shared_ptr<int> sh = std::make_shared<int>(42);
            wk = sh;
            // sh memory should be released at this point...
        }
        // ... but wk is still alive and needs access to control block
    }
    // now memory is released (sh and wk)
}
```

Поскольку `std::weak_ptr` не сохраняет свой ссылочный объект в живых, прямой доступ к данным через `std::weak_ptr` невозможен. Вместо этого он предоставляет функцию-член `lock()` которая пытается получить `std::shared_ptr` к указанному объекту:

```
#include <cassert>
#include <memory>
int main()
{
    {
        std::weak_ptr<int> wk;
        std::shared_ptr<int> sp;
        {
            std::shared_ptr<int> sh = std::make_shared<int>(42);
            wk = sh;
            // calling lock will create a shared_ptr to the object referenced by wk
            sp = wk.lock();
            // sh will be destroyed after this point, but sp is still alive
        }
        // sp still keeps the data alive.
        // At this point we could even call lock() again
        // to retrieve another shared_ptr to the same data from wk
        assert(*sp == 42);
        assert(!wk.expired());
        // resetting sp will delete the data,
        // as it is currently the last shared_ptr with ownership
        sp.reset();
        // attempting to lock wk now will return an empty shared_ptr,
        // as the data has already been deleted
    }
}
```

```
    sp = wk.lock();
    assert(!sp);
    assert(wk.expired());
}
}
```

Уникальное право собственности (std::unique_ptr)

C++ 11

`std::unique_ptr` - это шаблон класса, который управляет временем жизни динамически хранимого объекта. В отличие от `std::shared_ptr`, динамический объект принадлежит только *одному экземпляру* `std::unique_ptr` в любое время,

```
// Creates a dynamic int with value of 20 owned by a unique pointer
std::unique_ptr<int> ptr = std::make_unique<int>(20);
```

(Примечание: `std::unique_ptr` доступен с C++ 11 и `std::make_unique` с C++ 14.)

Только переменная `ptr` содержит указатель на динамически выделенный `int`. Когда уникальный указатель, который владеет объектом, выходит из области видимости, принадлежащий ему объект удаляется, т. Е. Его деструктор вызывается, если объект имеет тип класса, и память для этого объекта освобождается.

Чтобы использовать `std::unique_ptr` и `std::make_unique` с типами массивов, используйте их специализации массивов:

```
// Creates a unique_ptr to an int with value 59
std::unique_ptr<int> ptr = std::make_unique<int>(59);

// Creates a unique_ptr to an array of 15 ints
std::unique_ptr<int[]> ptr = std::make_unique<int[]>(15);
```

Вы можете получить доступ к `std::unique_ptr` как к необработанному указателю, потому что он перегружает эти операторы.

Вы можете передать право собственности на содержимое смарт-указателя на другой указатель, используя `std::move`, что приведет к тому, что оригинальный интеллектуальный указатель укажет на `nullptr`.

```
// 1. std::unique_ptr
std::unique_ptr<int> ptr = std::make_unique<int>();

// Change value to 1
*ptr = 1;

// 2. std::unique_ptr (by moving 'ptr' to 'ptr2', 'ptr' doesn't own the object anymore)
std::unique_ptr<int> ptr2 = std::move(ptr);
```

```
int a = *ptr2; // 'a' is 1
int b = *ptr;  // undefined behavior! 'ptr' is 'nullptr'
               // (because of the move command above)
```

Передача `unique_ptr` в функции как параметр:

```
void foo(std::unique_ptr<int> ptr)
{
    // Your code goes here
}

std::unique_ptr<int> ptr = std::make_unique<int>(59);
foo(std::move(ptr))
```

Возвращение `unique_ptr` из функций. Это предпочтительный способ написания фабричных функций на C++ 11, поскольку он ясно передает семантику права собственности: вызывающий пользователь получает результирующий `unique_ptr` и несет за это ответственность.

```
std::unique_ptr<int> foo()
{
    std::unique_ptr<int> ptr = std::make_unique<int>(59);
    return ptr;
}

std::unique_ptr<int> ptr = foo();
```

Сравните это с:

```
int* foo_cpp03();

int* p = foo_cpp03(); // do I own p? do I have to delete it at some point?
                     // it's not readily apparent what the answer is.
```

C++ 14

Шаблон класса `make_unique` предоставляется с C++ 14. Его легко добавить вручную в код C++ 11:

```
template<typename T, typename... Args>
typename std::enable_if<!std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(Args&&... args)
{ return std::unique_ptr<T>(new T(std::forward<Args>(args) ...)); }

// Use make_unique for arrays
template<typename T>
typename std::enable_if<std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(size_t n)
{ return std::unique_ptr<T>(new typename std::remove_extent<T>::type[n]()); }
```

C++ 11

В отличие от *немного* умного указателя (`std::auto_ptr`), `unique_ptr` также может быть создан с векторным распределением (*не* `std::vector`). Ранние примеры были для *скалярных* распределений. Например, чтобы иметь динамически выделенный целочисленный массив для 10 элементов, вы должны указать `int[]` как тип шаблона (а не просто `int`):

```
std::unique_ptr<int[]> arr_ptr = std::make_unique<int[]>(10);
```

Что можно упростить с помощью:

```
auto arr_ptr = std::make_unique<int[]>(10);
```

Теперь вы используете `arr_ptr` как будто это массив:

```
arr_ptr[2] = 10; // Modify third element
```

Вам не нужно беспокоиться о де-распределении. Эта специализированная версия шаблона вызывает конструкторы и деструкторы соответственно. Использование векторной версии `unique_ptr` или самого `vector` - это личный выбор.

В версиях до C ++ 11 был доступен `std::auto_ptr` . В отличие от `unique_ptr` разрешено копировать `auto_ptr` s, после чего исходный `ptr` потеряет право собственности на содержащий указатель и получает его.

Использование пользовательских удалений для создания оболочки для интерфейса C

Многие интерфейсы C, такие как [SDL2](#), имеют свои собственные функции удаления. Это означает, что вы не можете использовать интеллектуальные указатели напрямую:

```
std::unique_ptr<SDL_Surface> a; // won't work, UNSAFE!
```

Вместо этого вам нужно определить свой собственный дебит. В примерах здесь используется структура `SDL_Surface` которую следует освободить с помощью функции `SDL_FreeSurface()` , но они должны быть адаптированы ко многим другим C-интерфейсам.

Делектор должен быть вызван с аргументом указателя и, следовательно, может быть, например, простым указателем функции:

```
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);
```

Любой другой вызываемый объект также будет работать, например, класс с `operator()` :

```
struct SurfaceDeleter {
    void operator()(SDL_Surface* surf) {
        SDL_FreeSurface(surf);
    }
};
```



```

    }
};

std::unique_ptr<SDL_Surface, SurfaceDeleter> a(pointer, SurfaceDeleter{}); // safe
std::unique_ptr<SDL_Surface, SurfaceDeleter> b(pointer); // equivalent to the above
// as the deleter is value-
initialized

```

Это не только обеспечивает безопасное, новое накладные расходы (если вы используете `unique_ptr`) автоматическое управление памятью, вы также получаете безопасность исключений.

Обратите внимание, что делтер является частью типа `unique_ptr`, и реализация может использовать [пустую оптимизацию базы](#), чтобы избежать изменения размера для пустых пользовательских удалений. Таким образом, в то время как `std::unique_ptr<SDL_Surface, SurfaceDeleter>` и `std::unique_ptr<SDL_Surface, void(*) (SDL_Surface*)>` решает одну и ту же проблему аналогичным образом, прежний тип по-прежнему является только размером указателя, в то время как последний тип должен содержать *два* указателя: и `SDL_Surface*` и указатель функции! При использовании произвольных пользовательских удалений функциональных возможностей предпочтительнее обернуть функцию в пустой тип.

В случаях, когда подсчет ссылок важен, можно использовать `shared_ptr` вместо `unique_ptr`. В `shared_ptr` всегда хранится `deleter`, это стирает тип делтера, который может быть полезен в API. Недостатки использования `shared_ptr` сравнению с `unique_ptr` включают в себя более высокую стоимость памяти для хранения делтера и стоимость выполнения для поддержания счетчика ссылок.

```

// deleter required at construction time and is part of the type
std::unique_ptr<SDL_Surface, void(*) (SDL_Surface*)> a(pointer, SDL_FreeSurface);

// deleter is only required at construction time, not part of the type
std::shared_ptr<SDL_Surface> b(pointer, SDL_FreeSurface);

```

C++ 17

С `template auto` мы можем сделать еще проще обернуть наши пользовательские удалители:

```

template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) {
        DeleteFn(ptr);
    }
};

template <class T, auto DeleteFn>
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;

```

С помощью приведенного выше примера просто:

```
unique_ptr_deleter<SDL_Surface, SDL_FreeSurface> c(pointer);
```

Здесь цель `auto` - обрабатывать все свободные функции, независимо от того, возвращают ли они `void` (например, `SDL_FreeSurface`) или нет (например, `fclose`).

Уникальная собственность без семантики перемещения (`auto_ptr`)

C ++ 11

ПРИМЕЧАНИЕ: `std::auto_ptr` устарел на C ++ 11 и будет удален на C ++ 17. Вы должны использовать это только в том случае, если вы вынуждены использовать C ++ 03 или ранее и готовы быть осторожными. Рекомендуется перейти к `unique_ptr` в сочетании с `std::move` для замены поведения `std::auto_ptr`.

До того, как мы установили `std::unique_ptr`, перед семантикой перемещения была `std::auto_ptr`. `std::auto_ptr` предоставляет уникальное право собственности, но передает право собственности на копию.

Как и во всех умных указателях, `std::auto_ptr` автоматически очищает ресурсы (см. [RAII](#)):

```
{
    std::auto_ptr<int> p(new int(42));
    std::cout << *p;
} // p is deleted here, no memory leaked
```

но допускает только одного владельца:

```
std::auto_ptr<X> px = ...;
std::auto_ptr<X> py = px;
// px is now empty
```

Это позволяет использовать `std::auto_ptr`, чтобы сохранить явное и уникальное право собственности на риск потери права собственности:

```
void f(std::auto_ptr<X> ) {
    // assumes ownership of X
    // deletes it at end of scope
};

std::auto_ptr<X> px = ...;
f(px); // f acquires ownership of underlying X
      // px is now empty
px->foo(); // NPE!
// px.~auto_ptr() does NOT delete
```

Передача права собственности произошла в конструкторе «сору». `auto_ptr` конструктор копирования «s и копировать оператор присваивания принимают операнды неисполнением `const` ссылки, так что они могут быть изменены. Пример реализации может быть:

```

template <typename T>
class auto_ptr {
    T* ptr;
public:
    auto_ptr(auto_ptr& rhs)
    : ptr(rhs.release())
    { }

    auto_ptr& operator=(auto_ptr& rhs) {
        reset(rhs.release());
        return *this;
    }

    T* release() {
        T* tmp = ptr;
        ptr = nullptr;
        return tmp;
    }

    void reset(T* tmp = nullptr) {
        if (ptr != tmp) {
            delete ptr;
            ptr = tmp;
        }
    }

    /* other functions ... */
};

```

Это нарушает семантику копирования, которая требует, чтобы копирование объекта оставило вас с двумя эквивалентными версиями. Для любого типа копирования `T`, я должен уметь писать:

```

T a = ...;
T b(a);
assert(b == a);

```

Но для `auto_ptr` это не так. В результате небезопасно вставлять `auto_ptr` в контейнеры.

Получение `shared_ptr`, ссылающегося на это

`enable_shared_from_this` позволяет получить правильный `shared_ptr` экземпляра `this`.

При выводе своего класса из шаблона класса `enable_shared_from_this`, наследование метода `shared_from_this`, который возвращает `shared_ptr` экземпляра `this`.

Обратите внимание, что объект должен быть создан как `shared_ptr` на первом месте:

```

#include <memory>
class A: public enable_shared_from_this<A> {
};
A* ap1 = new A();
shared_ptr<A> ap2(ap1); // First prepare a shared pointer to the object and hold it!
// Then get a shared pointer to the object from the object itself
shared_ptr<A> ap3 = ap1->shared_from_this();

```

```
int c3 =ap3.use_count(); // =2: pointing to the same object
```

Примечание (2) вы не можете вызвать `enable_shared_from_this` внутри конструктора.

```
#include <memory> // enable_shared_from_this

class Widget : public std::enable_shared_from_this< Widget >
{
public:
    void DoSomething()
    {
        std::shared_ptr< Widget > self = shared_from_this();
        someEvent -> Register( self );
    }
private:
    ...
};

int main()
{
    ...
    auto w = std::make_shared< Widget >();
    w -> DoSomething();
    ...
}
```

Если вы используете `shared_from_this()` для объекта, не принадлежащего `shared_ptr`, такого как локальный автоматический объект или глобальный объект, тогда поведение не определено. Так как C++ 17 вместо этого выбрасывает `std::bad_alloc`.

Использование `shared_from_this()` из конструктора эквивалентно использованию его на объекте, не принадлежащем `shared_ptr`, потому что объекты обладают `shared_ptr` после возврата конструктора.

Кастинг `std::shared_ptr` указатели

Это не представляется возможным непосредственно использовать `static_cast`, `const_cast`, `dynamic_cast` и `reinterpret_cast` на `std::shared_ptr` для извлечения собственности обмена указатель с указателем передается в качестве аргумента. Вместо этого следует использовать функции `std::static_pointer_cast`, `std::const_pointer_cast`, `std::dynamic_pointer_cast` и `std::reinterpret_pointer_cast`:

```
struct Base { virtual ~Base() noexcept {} };
struct Derived: Base {};
auto derivedPtr(std::make_shared<Derived>());
auto basePtr(std::static_pointer_cast<Base>(derivedPtr));
auto constBasePtr(std::const_pointer_cast<Base const>(basePtr));
auto constDerivedPtr(std::dynamic_pointer_cast<Derived const>(constBasePtr));
```

Обратите внимание, что `std::reinterpret_pointer_cast` недоступен в C++ 11 и C++ 14, поскольку он был предложен только [N3920](#) и принят в TS Basic Library в [феврале 2014](#)

года . Однако его можно реализовать следующим образом:

```
template <typename To, typename From>
inline std::shared_ptr<To> reinterpret_pointer_cast(
    std::shared_ptr<From> const & ptr) noexcept
{ return std::shared_ptr<To>(ptr, reinterpret_cast<To *>(ptr.get())); }
```

Написание умного указателя: value_ptr

value_ptr - это умный указатель, который ведет себя как значение. При копировании копируется его содержимое. При создании создается его содержимое.

```
// Like std::default_delete:
template<class T>
struct default_copier {
    // a copier must handle a null T const* in and return null:
    T* operator()(T const* tin)const {
        if (!tin) return nullptr;
        return new T(*tin);
    }
    void operator()(void* dest, T const* tin)const {
        if (!tin) return;
        return new(dest) T(*tin);
    }
};
// tag class to handle empty case:
struct empty_ptr_t {};
constexpr empty_ptr_t empty_ptr{};
// the value pointer type itself:
template<class T, class Copier=default_copier<T>, class Deleter=std::default_delete<T>,
        class Base=std::unique_ptr<T, Deleter>
>
struct value_ptr:Base, private Copier {
    using copier_type=Copier;
    // also typedefs from unique_ptr

    using Base::Base;

    value_ptr( T const& t ):
        Base( std::make_unique<T>(t) ),
        Copier()
    {}
    value_ptr( T && t ):
        Base( std::make_unique<T>(std::move(t)) ),
        Copier()
    {}
    // almost-never-empty:
    value_ptr():
        Base( std::make_unique<T>() ),
        Copier()
    {}
    value_ptr( empty_ptr_t ) {}

    value_ptr( Base b, Copier c={} ):
        Base(std::move(b)),
        Copier(std::move(c))
    {}
};
```

```

Copier const& get_copier() const {
    return *this;
}

value_ptr clone() const {
    return {
        Base(
            get_copier() (this->get()),
            this->get_deleter()
        ),
        get_copier()
    };
}

value_ptr(value_ptr&&)=default;
value_ptr& operator=(value_ptr&&)=default;

value_ptr(value_ptr const& o):value_ptr(o.clone()) {}
value_ptr& operator=(value_ptr const&o) {
    if (o && *this) {
        // if we are both non-null, assign contents:
        **this = *o;
    } else {
        // otherwise, assign a clone (which could itself be null):
        *this = o.clone();
    }
    return *this;
}

value_ptr& operator=( T const& t ) {
    if (*this) {
        **this = t;
    } else {
        *this = value_ptr(t);
    }
    return *this;
}

value_ptr& operator=( T && t ) {
    if (*this) {
        **this = std::move(t);
    } else {
        *this = value_ptr(std::move(t));
    }
    return *this;
}

T& get() { return **this; }
T const& get() const { return **this; }
T* get_pointer() {
    if (!*this) return nullptr;
    return std::addressof(get());
}

T const* get_pointer() const {
    if (!*this) return nullptr;
    return std::addressof(get());
}

// operator-> from unique_ptr
};

template<class T, class...Args>
value_ptr<T> make_value_ptr( Args&&... args ) {
    return {std::make_unique<T>(std::forward<Args>(args)...) };
}

```

Этот конкретный value_ptr пуст, если вы создаете его с помощью empty_ptr_t или если вы

перейдете от него. Он раскрывает тот факт, что это `unique_ptr`, поэтому на нем работает `explicit operator bool()` `const .get()` был изменен, чтобы вернуть ссылку (поскольку она почти никогда не пуста), а `.get_pointer()` вместо этого возвращает указатель.

Этот интеллектуальный указатель может быть полезен для случаев `pImpl`, где нам нужна семантика значений, но мы также не хотим раскрывать содержимое `pImpl` вне файла реализации.

С помощью нестандартного `Copier` он может обрабатывать виртуальные базовые классы, которые знают, как создавать экземпляры своих производных и превращать их в типы значений.

Прочитайте Умные указатели онлайн: <https://riptutorial.com/ru/cplusplus/topic/509/умные-указатели>

глава 132: Управление памятью

Синтаксис

- `::( opt ) new ( выражение-list ) ( opt ) new-initial-initial-initializer ( opt )`
- `::( opt ) new ( expression-list ) ( opt ) ( type-id ) new-initializer ( opt )`
- `::( opt ) delete cast-expression`
- `::( opt ) delete [] cast-expression`
- `std :: unique_ptr < type-id > var_name (новый тип-идентификатор ( opt )); // C ++ 11`
- `std :: shared_ptr < type-id > var_name (новый тип-идентификатор ( opt )); // C ++ 11`
- `std :: shared_ptr < type-id > var_name = std :: make_shared < type-id > ( opt ); // C ++ 11`
- `std :: unique_ptr < type-id > var_name = std :: make_unique < type-id > ( opt ); // C ++ 14`

замечания

Ведущий `::` заставляет новый или удаляемый оператор искать глобальную область, переопределяя любые перегруженные новые или удаленные операторы класса.

Необязательные аргументы, следующие за `new` ключевым словом, обычно используются для вызова **нового места размещения**, но могут также использоваться для передачи дополнительной информации в распределитель, например тега, запрашивающего выделение памяти из выбранного пула.

Выделенный тип обычно явно указан, *например*, `new Foo`, но также может быть записан как `auto` (начиная с C ++ 11) или `decltype(auto)` (начиная с C ++ 14), чтобы вывести его из инициализатора.

Инициализация выделенного объекта происходит по тем же правилам, что и инициализация локальных переменных. В частности, объект будет инициализирован по умолчанию, если инициализатор `iso` опущен, а при динамическом распределении скалярного типа или массива скалярного типа нет гарантии, что память будет обнулена.

Объект массива, созданный *новым выражением*, должен быть уничтожен с помощью `delete[]`, независимо от того, было ли *новое выражение* написано с помощью `[]` или нет. Например:

```
using IA = int[4];
int* pIA = new IA;
delete[] pIA; // right
// delete pIA; // wrong
```

Examples

стек

Стек представляет собой небольшую область памяти, в которую временные значения помещаются во время выполнения. Выделение данных в стек очень быстро по сравнению с распределением кучи, так как вся память уже назначена для этой цели.

```
int main() {
    int a = 0; //Stored on the stack
    return a;
}
```

Стек назван так, потому что цепочки вызовов функций будут иметь временную память, расположенную друг над другом, каждая из которых использует отдельный небольшой раздел памяти.

```
float bar() {
    //f will be placed on the stack after anything else
    float f = 2;
    return f;
}

double foo() {
    //d will be placed just after anything within main()
    double d = bar();
    return d;
}

int main() {
    //The stack has no user variables stored in it until foo() is called
    return (int)foo();
}
```

Данные, хранящиеся в стеке, действительны только до тех пор, пока область, которая распределяет переменную, все еще активна.

```
int* pA = nullptr;

void foo() {
    int b = *pA;
    pA = &b;
}

int main() {
    int a = 5;
    pA = &a;
    foo();
    //Undefined behavior, the value pointed to by pA is no longer in scope
    a = *pA;
}
```

Свободное хранение (куча, динамическое распределение ...)

Термин **«куча»** представляет собой общий вычислительный термин, означающий область

памяти, из которой части могут быть выделены и освобождены независимо от памяти, предоставленной **стеком** .

В C++ *Стандарт* относится к этой области как к **свободному магазину**, который считается более точным.

Области памяти, выделенные из **Free Store**, могут жить дольше, чем исходная область, в которой она была выделена. Данные, слишком большие для хранения в стеке, также могут быть выделены из **Free Store** .

Сырая память может быть выделена и освобождена *новыми* и *удаляемыми* ключевыми словами.

```
float *foo = nullptr;
{
    *foo = new float; // Allocates memory for a float
    float bar;        // Stack allocated
} // End lifetime of bar, while foo still alive

delete foo;           // Deletes the memory for the float at pF, invalidating the pointer
foo = nullptr;        // Setting the pointer to nullptr after delete is often considered good
practice
```

Также можно назначить массивы с фиксированным размером с *новыми* и *удалить* , с немного отличающимся синтаксисом. Выделение массива несовместимо с распределением без массива, и смешивание двух приведет к повреждению кучи. Выделение массива также выделяет память для отслеживания размера массива для последующего удаления определенным образом.

```
// Allocates memory for an array of 256 ints
int *foo = new int[256];
// Deletes an array of 256 ints at foo
delete[] foo;
```

При использовании *new* и *delete* вместо *malloc* и *free* , конструктор и деструктор будут выполняться (подобно объектам на основе стека). Поэтому *новые* и *удалить* , предпочитаемые над *таНосом* и *бесплатно*.

```
struct ComplexType {
    int a = 0;

    ComplexType() { std::cout << "Ctor" << std::endl; }
    ~ComplexType() { std::cout << "Dtor" << std::endl; }
};

// Allocates memory for a ComplexType, and calls its constructor
ComplexType *foo = new ComplexType();
//Calls the destructor for ComplexType() and deletes memory for a ComplexType at pC
delete foo;
```

Начиная с C ++ 11, для указания владельца рекомендуется использовать [интеллектуальные указатели](#) .

C ++ 14

C ++ 14 добавил `std::make_unique` в STL, изменив рекомендацию в пользу `std::make_unique` или `std::make_shared` вместо того, чтобы использовать голый *новый* и *удалить* .

Размещение новых

Бывают ситуации, когда мы не хотим полагаться на Free Store для выделения памяти, и мы хотим использовать пользовательские выделения памяти, используя `new` .

В этих ситуациях мы можем использовать Placement New , где мы можем сказать оператору «`new`» выделить память из предварительно выделенной ячейки памяти

Например

```
int a4byteInteger;  
  
char *a4byteChar = new (&a4byteInteger) char[4];
```

В этом примере память, `a4byteChar` представляет собой 4 байта, выделенных для «стека» с помощью целочисленной переменной `a4byteInteger` .

Преимущество такого распределения памяти заключается в том, что программисты контролируют распределение. В приведенном выше примере, поскольку `a4byteInteger` выделяется в стеке, нам не нужно делать явный вызов «`delete a4byteChar`».

Такое же поведение может быть достигнуто и для динамически распределенной памяти. Например

```
int *a8byteDynamicInteger = new int[2];  
  
char *a8byteChar = new (a8byteDynamicInteger) char[8];
```

В этом случае указатель на память `a8byteChar` будет ссылаться на динамическую память, выделенную `a8byteDynamicInteger` . В этом случае, однако, нам нужно явно вызвать `delete a8byteDynamicInteger` для освобождения памяти

Другой пример для класса C ++

```
struct ComplexType {  
    int a;  
  
    ComplexType() : a(0) {}  
    ~ComplexType() {}  
};
```

```

int main() {
    char* dynArray = new char[256];

    //Calls ComplexType's constructor to initialize memory as a ComplexType
    new((void*)dynArray) ComplexType();

    //Clean up memory once we're done
    reinterpret_cast<ComplexType*>(dynArray)->~ComplexType();
    delete[] dynArray;

    //Stack memory can also be used with placement new
    alignas(ComplexType) char localArray[256]; //alignas() available since C++11

    new((void*)localArray) ComplexType();

    //Only need to call the destructor for stack memory
    reinterpret_cast<ComplexType*>(localArray)->~ComplexType();

    return 0;
}

```

Прочитайте Управление памятью онлайн: <https://riptutorial.com/ru/cplusplus/topic/2873/управление-памятью>

глава 133: Управление потоком

замечания

Проверьте [тему циклов](#) для разных типов циклов.

Examples

дело

Вводит метку case оператора switch. Операнд должен быть постоянным выражением и соответствовать условию переключения в типе. Когда оператор switch выполняется, он переходит к метке case с операндом, равным условию, если оно есть.

```
char c = getchar();
bool confirmed;
switch (c) {
    case 'y':
        confirmed = true;
        break;
    case 'n':
        confirmed = false;
        break;
    default:
        std::cout << "invalid response!\n";
        abort();
}
```

переключатель

Согласно стандарту C ++,

Оператор `switch` приводит к тому, что управление передается одному из нескольких операторов в зависимости от значения условия.

Ключевое слово `switch` следует в скобках условие и блок, который может содержать `case` метки и необязательное по `default` ярлык. Когда оператор `switch` выполняется, управление будет передано либо на метку `case` либо на значение, соответствующее значению условия, если таковое имеется, или на метку по `default` , если таковая имеется.

Условие должно быть выражением или объявлением, которое имеет либо целочисленный, либо тип перечисления, либо тип класса с функцией преобразования для целочисленного или перечисляемого типа.

```
char c = getchar();
bool confirmed;
```

```

switch (c) {
    case 'y':
        confirmed = true;
        break;
    case 'n':
        confirmed = false;
        break;
    default:
        std::cout << "invalid response!\n";
        abort();
}

```

ЛОВИТЬ

Ключевое слово `catch` вводит обработчик исключений, то есть блок, в который будет передаваться элемент управления, когда выбрано исключение совместимого типа. Ключевое слово `catch` следует за *объявлением исключения* в скобках, которое по форме похоже на объявление параметра функции: имя параметра может быть опущено, а разрешен эллипс `...`, который соответствует любому типу. Обработчик исключений будет обрабатывать исключение только в том случае, если его объявление совместимо с типом исключения. Дополнительные сведения см. В разделе [catching exceptions](#).

```

try {
    std::vector<int> v(N);
    // do something
} catch (const std::bad_alloc&) {
    std::cout << "failed to allocate memory for vector!" << std::endl;
} catch (const std::runtime_error& e) {
    std::cout << "runtime error: " << e.what() << std::endl;
} catch (...) {
    std::cout << "unexpected exception!" << std::endl;
    throw;
}

```

дефолт

В операторе `switch` вводится метка, на которую будет вклиниваться, если значение условия не равно ни одному из значений меток `case`.

```

char c = getchar();
bool confirmed;
switch (c) {
    case 'y':
        confirmed = true;
        break;
    case 'n':
        confirmed = false;
        break;
    default:
        std::cout << "invalid response!\n";
        abort();
}

```

Определяет конструктор по умолчанию, конструктор копирования, конструктор перемещения, деструктор, оператор присваивания копий или оператор переноса назначения, чтобы иметь поведение по умолчанию.

```
class Base {
    // ...
    // we want to be able to delete derived classes through Base*,
    // but have the usual behaviour for Base's destructor.
    virtual ~Base() = default;
};
```

если

Вводит оператор if. Ключевое слово `if` должно сопровождаться условием в скобках, которое может быть выражением или объявлением. Если условие является правдоподобным, выполнение после условия будет выполнено.

```
int x;
std::cout << "Please enter a positive number." << std::endl;
std::cin >> x;
if (x <= 0) {
    std::cout << "You didn't enter a positive number!" << std::endl;
    abort();
}
```

еще

В первом подзадаче оператора if может следовать ключевое слово `else`. Подкрепление после ключевого слова `else` будет выполняться, когда условие ложно (то есть, когда первое подзадание не выполняется).

```
int x;
std::cin >> x;
if (x%2 == 0) {
    std::cout << "The number is even\n";
} else {
    std::cout << "The number is odd\n";
}
```

идти к

Переход к помеченной инструкции, которая должна находиться в текущей функции.

```
bool f(int arg) {
    bool result = false;
    hWidget widget = get_widget(arg);
    if (!g()) {
        // we can't continue, but must do cleanup still
        goto end;
    }
    // ...
}
```

```

    result = true;
end:
    release_widget(widget);
    return result;
}

```

вернуть

Возвращает управление от функции к вызывающему.

Если `return` имеет операнд, операнд преобразуется в тип возвращаемого значения функции, а преобразованное значение возвращается вызывающему.

```

int f() {
    return 42;
}
int x = f(); // x is 42
int g() {
    return 3.14;
}
int y = g(); // y is 3

```

Если `return` не имеет операнда, функция должна иметь тип возврата `void`. В качестве специального случая функция `void`-returning также может возвращать выражение, если выражение имеет тип `void`.

```

void f(int x) {
    if (x < 0) return;
    std::cout << sqrt(x);
}
int g() { return 42; }
void h() {
    return f(); // calls f, then returns
    return g(); // ill-formed
}

```

Когда `main` возвращает, `std::exit` неявно вызывается с возвращаемым значением, и поэтому значение возвращается в среду выполнения. (Однако возврат из `main` уничтожает автоматические локальные переменные, а вызов `std::exit` напрямую не выполняется).

```

int main(int argc, char** argv) {
    if (argc < 2) {
        std::cout << "Missing argument\n";
        return EXIT_FAILURE; // equivalent to: exit(EXIT_FAILURE);
    }
}

```

бросать

1. Когда `throw` происходит в выражении с операндом, его эффект заключается в том, чтобы выдать **исключение**, которое является копией операнда.


```
void print_asterisks(int count) {
    if (count < 0) {
        throw std::invalid_argument("count cannot be negative!");
    }
    while (count-- > 0) { putchar('*'); }
}
```

2. Когда `throw` происходит в выражении без операнда, его эффект заключается в том, чтобы **перебросить текущее исключение**. Если не существует текущего исключения, вызывается `std::terminate`.

```
try {
    // something risky
} catch (const std::bad_alloc&) {
    std::cerr << "out of memory" << std::endl;
} catch (...) {
    std::cerr << "unexpected exception" << std::endl;
    // hope the caller knows how to handle this exception
    throw;
}
```

3. Когда `throw` происходит в деклараторе функции, он вводит спецификацию динамических исключений, в которой перечислены типы исключений, которые функция разрешает распространять.

```
// this function might propagate a std::runtime_error,
// but not, say, a std::logic_error
void risky() throw(std::runtime_error);
// this function can't propagate any exceptions
void safe() throw();
```

Спецификации динамических исключений устаревают с C++ 11.

Обратите внимание, что первые два использования `throw` перечисленные выше, представляют собой выражения, а не выражения. (Тип выражения `throw void`.) Это позволяет вложить их в выражения, например:

```
unsigned int predecessor(unsigned int x) {
    return (x > 0) ? (x - 1) : (throw std::invalid_argument("0 has no predecessor"));
}
```

пытаться

Ключевое слово `try` сопровождается блоком или списком инициализатора конструктора, а затем блоком (см. [Здесь](#)). За блоком `try` следует один или несколько **блоков `catch`**. Если **исключение** распространяется из блока `try`, каждый из соответствующих блоков `catch` после блока `try` имеет возможность обрабатывать исключение, если типы совпадают.

```
std::vector<int> v(N); // if an exception is thrown here,
                     // it will not be caught by the following catch block
```

```
try {
    std::vector<int> v(N); // if an exception is thrown here,
                        // it will be caught by the following catch block
    // do something with v
} catch (const std::bad_alloc&) {
    // handle bad_alloc exceptions from the try block
}
```

Условные структуры: if, if..else

если и еще:

он использовал, чтобы проверить, возвращает ли данное выражение true или false и действует как таковое:

```
if (condition) statement
```

условием может быть любое допустимое выражение C ++, которое возвращает то, что проверяется на правду / ложь, например:

```
if (true) { /* code here */ } // evaluate that true is true and execute the code in the
brackets
if (false) { /* code here */ } // always skip the code since false is always false
```

условие может быть любым, функцией, переменной или сравнением, например

```
if(istrue()) { } // evaluate the function, if it returns true, the if will execute the code
if(isTrue(var)) { } //evaluate the return of the function after passing the argument var
if(a == b) { } // this will evaluate the return of the experssion (a==b) which will be true if
equal and false if unequal
if(a) { } //if a is a boolean type, it will evaluate for its value, if it's an integer, any
non zero value will be true,
```

если мы хотим проверить несколько выражений, мы можем сделать это двумя способами:

используя двоичные операторы :

```
if (a && b) { } // will be true only if both a and b are true (binary operators are outside
the scope here
if (a || b ) { } //true if a or b is true
```

используя if / ifelse / else :

для простого переключателя, если или иначе

```
if (a== "test") {
    //will execute if a is a string "test"
} else {
    // only if the first failed, will execute
}
```

для нескольких вариантов:

```
if (a=='a') {  
    // if a is a char valued 'a'  
} else if (a=='b') {  
    // if a is a char valued 'b'  
} else if (a=='c') {  
    // if a is a char valued 'c'  
} else {  
    //if a is none of the above  
}
```

однако следует отметить, что вместо этого вы должны использовать « **switch** », если ваш код проверяет значение той же переменной

Операторы перехода: break, continue, goto, exit.

Инструкция по разрыву:

Используя break, мы можем оставить цикл, даже если условие для его конца не выполняется. Его можно использовать для завершения бесконечного цикла или для его завершения до его естественного конца

Синтаксис

```
break;
```

Пример : мы часто используем break в случаях switch , т. Е. После того, как переключатель i удовлетворяется, выполняется блок кода этого условия.

```
switch(conditon){  
    case 1: block1;  
    case 2: block2;  
    case 3: block3;  
    default: blockdefault;  
}
```

в этом случае, если выполняется случай 1, выполняется блок 1, то, что мы действительно хотим, это только обработанный блок1, но вместо этого, когда блок1 обрабатывается оставшимися блоками, блок2, block3 и blockdefault также обрабатываются, хотя только 1-й случай был насыщен . Чтобы избежать этого, мы используем break в конце каждого блока, например:

```
switch(condition){  
    case 1: block1;  
        break;  
    case 2: block2;  
        break;  
    case 3: block3;  
        break;
```

```
default: blockdefault;
    break;
}
```

поэтому обрабатывается только один блок, и управление выходит из цикла переключателя.

`break` также может использоваться в других условных и не условных циклах, например, `if`, `while` `for` `T. Д .;`

пример:

```
if(condition1){
    ....
    if(condition2){
        .....
        break;
    }
    ...
}
```

Инструкция продолжения:

Инструкция `continue` приводит к тому, что программа пропускает остальную часть цикла в текущей итерации, как если бы конец блока оператора был достигнут, заставляя его перейти к следующей итерации.

Синтаксис

```
continue;
```

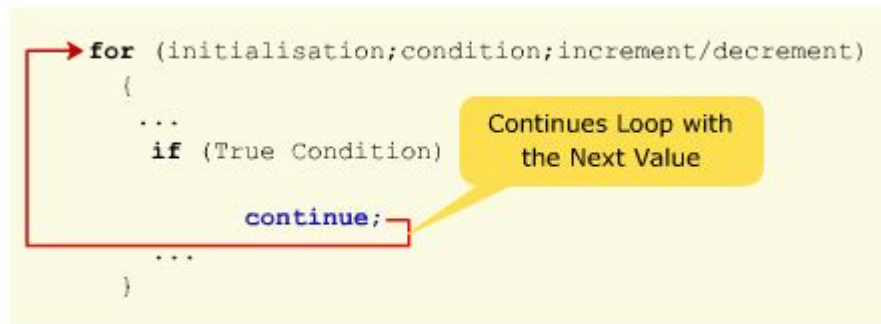
Пример рассмотрим следующее:

```
for(int i=0;i<10;i++){
    if(i%2==0)
        continue;
    cout<<"\n @"<<i;
}
```

который производит выход:

```
@1
@3
@5
@7
@9
```

в этот код всякий раз, когда выполняется условие `i%2==0` `continue` обрабатывается, это заставляет компилятор пропустить весь оставшийся код (печать `@` и `i`), а инструкция `increment / decment` цикла выполняется.



Инструкция goto:

Это позволяет совершать абсолютный переход к другой точке программы. Вы должны использовать эту функцию тщательно, так как ее выполнение игнорирует ограничения типа вложения. Точка назначения идентифицируется меткой, которая затем используется как аргумент для инструкции goto. Метка состоит из допустимого идентификатора, за которым следует двоеточие (:)

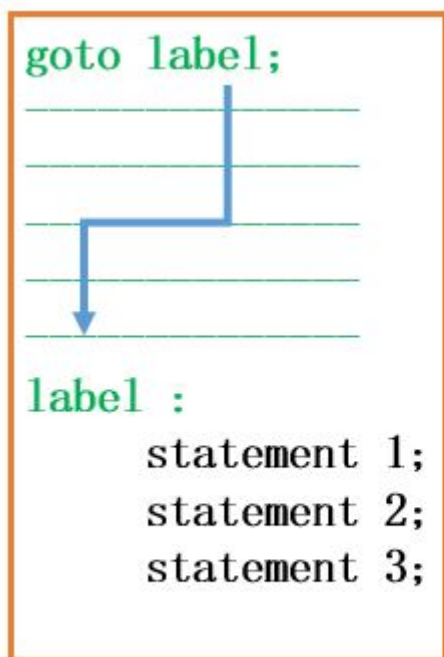
Синтаксис

```

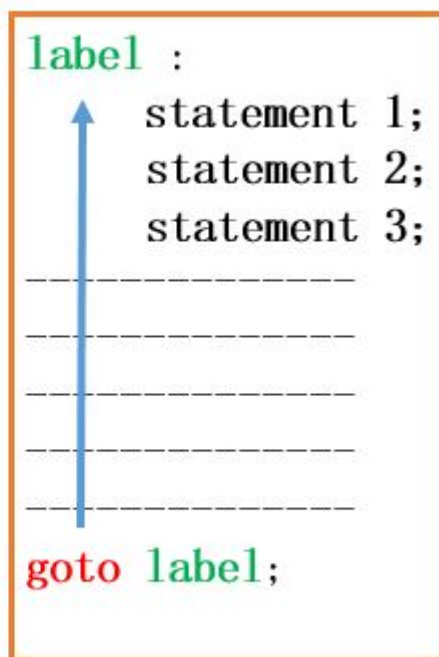
goto label;
..
.
label: statement;

```

Примечание. Использование инструкции goto сильно обескуражено, поскольку сложно отслеживать поток управления программой, что затрудняет понимание программы и ее трудно изменить.



Forward Reference



Backward Reference

Пример :

```
int num = 1;
STEP:
do{

    if( num%2==0 )
    {
        num = num + 1;
        goto STEP;
    }

    cout << "value of num : " << num << endl;
    num = num + 1;
}while( num < 10 );
```

ВЫХОД :

```
value of num : 1
value of num : 3
value of num : 5
value of num : 7
value of num : 9
```

всякий раз, когда выполняется условие `num%2==0` , `goto` отправляет управление исполнением в начало цикла `do-while` `while`.

Функция выхода:

`exit` - это функция, определенная в `cstdlib` . Цель `exit` - завершить работу программы с определенным кодом выхода. Его прототип:

```
void exit (int exit code);
```

`cstdlib` определяет стандартные коды выхода `EXIT_SUCCESS` и `EXIT_FAILURE` .

Прочитайте Управление потоком онлайн: <https://riptutorial.com/ru/cplusplus/topic/7837/управление-потоком>

глава 134: Управление ресурсами

Вступление

Одним из самых сложных действий на C и C++ является управление ресурсами. К счастью, на C++ у нас есть много способов разработки управления ресурсами в наших программах. Эта статья надеется объяснить некоторые из идиом и методов, используемых для управления выделенными ресурсами.

Examples

Инициализация ресурсов

Приобретение ресурсов - это инициализация (RAII) - распространенная идиома в управлении ресурсами. В случае динамической памяти он использует [интеллектуальные указатели](#) для выполнения управления ресурсами. При использовании RAII приобретенный ресурс сразу получает право владения интеллектуальному указателю или эквивалентному менеджеру ресурсов. Доступ к ресурсу осуществляется только через этого менеджера, поэтому менеджер может отслеживать различные операции. Например, `std::auto_ptr` автоматически освобождает свой соответствующий ресурс, когда он выпадает из области действия или иным образом удаляется.

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    {
        auto_ptr ap(new int(5)); // dynamic memory is the resource
        cout << *ap << endl; // prints 5
    } // auto_ptr is destroyed, its resource is automatically freed
}
```

C++ 11

Основная проблема `std::auto_ptr` заключается в том, что он не может копироваться без передачи права собственности:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    auto_ptr ap1(new int(5));
    cout << *ap1 << endl; // prints 5
    auto_ptr ap2(ap1); // copy ap2 from ap1; ownership now transfers to ap2
    cout << *ap2 << endl; // prints 5
}
```

```
cout << ap1 == nullptr << endl; // prints 1; ap1 has lost ownership of resource
}
```

Из-за этой странной семантики копирования, `std::auto_ptr` нельзя использовать в контейнерах, между прочим. Причина этого заключается в том, чтобы предотвратить удаление памяти дважды: если есть два `auto_ptr`s с правом собственности на один и тот же ресурс, они обе пытаются освободить его, когда они будут уничтожены. Освобождение уже освобожденного ресурса может вызвать проблемы, поэтому важно предотвратить его. Тем не менее, `std::shared_ptr` имеет способ избежать этого, не передавая право собственности при копировании:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    shared_ptr sp2;
    {
        shared_ptr sp1(new int(5)); // give ownership to sp1
        cout << *sp1 << endl; // prints 5
        sp2 = sp1; // copy sp2 from sp1; both have ownership of resource
        cout << *sp1 << endl; // prints 5
        cout << *sp2 << endl; // prints 5
    } // sp1 goes out of scope and is destroyed; sp2 has sole ownership of resource
    cout << *sp2 << endl;
} // sp2 goes out of scope; nothing has ownership, so resource is freed
```

Мьютексы и безопасность потоков

Проблемы могут возникать, когда несколько потоков пытаются получить доступ к ресурсу. Для простого примера предположим, что у нас есть поток, который добавляет его к переменной. Он делает это, сначала прочитав переменную, добавив ее к ней, а затем сохранит ее. Предположим, что мы инициализируем эту переменную до 1, затем создаем два экземпляра этого потока. По завершении обоих потоков интуиция предполагает, что эта переменная должна иметь значение 3. Однако приведенная ниже таблица иллюстрирует, что может пойти не так:

	Тема 1	Тема 2
Время Шаг 1	Чтение 1 из переменной	
Время Шаг 2		Чтение 1 из переменной
Время Шаг 3	Добавьте 1 плюс 1, чтобы получить 2	
Время Шаг		Добавьте 1 плюс 1, чтобы

	Тема 1	Тема 2
4		получить 2
Время Шаг 5	Хранить 2 в переменной	
Время Шаг 6		Хранить 2 в переменной

Как вы можете видеть, в конце операции 2 находится в переменной, а не 3. Причина в том, что Thread 2 прочитал переменную до того, как был обновлен Thread 1. Решение?

Мьютексы.

Взаимная блокировка (контаминация **Мут** UAL **экс** ключения) является объектом управления ресурсами предназначен для решения такого рода проблем. Когда поток хочет получить доступ к ресурсу, он «приобретает» мьютекс ресурса. Как только это делается, доступ к ресурсу, поток «освобождает» мьютекс. Пока мьютекс будет получен, все вызовы для получения мьютекса не вернутся, пока мьютекс не будет отпущен. Чтобы лучше понять это, подумайте о мьютексе как очереди ожидания в супермаркете: потоки идут в линию, пытаются получить мьютексы, а затем ожидают, что потоки впереди них закончатся, а затем используя ресурс, а затем выйдут из путем освобождения мьютекса. Было бы полное пандемониум, если бы каждый попытался получить доступ к ресурсу сразу.

C ++ 11

`std::mutex` - это реализация `std::mutex` C ++ 11.

```
#include <thread>
#include <mutex>
#include <iostream>
using namespace std;

void add_1(int& i, const mutex& m) { // function to be run in thread
    m.lock();
    i += 1;
    m.unlock();
}

int main() {
    int var = 1;
    mutex m;

    cout << var << endl; // prints 1

    thread t1(add_1, var, m); // create thread with arguments
    thread t2(add_1, var, m); // create another thread
    t1.join(); t2.join(); // wait for both threads to finish

    cout << var << endl; // prints 3
}
```

Прочитайте Управление ресурсами онлайн: <https://riptutorial.com/ru/cplusplus/topic/8336/управление-ресурсами>

глава 135: Файловый ввод-вывод

Вступление

С ++-файл ввода-вывода выполняется через *потоки* . Ключевыми абстракциями являются:

`std::istream` для чтения текста.

`std::ostream` для `std::ostream` текста.

`std::streambuf` для чтения или записи символов.

В *форматированном вводе* используется `operator>>` .

Форматированный вывод использует `operator<<` .

В потоках используется `std::locale` , например, для подробностей форматирования и для перевода между внешними кодировками и внутренней кодировкой.

Подробнее о потоках: [<iostream> Library](#)

Examples

Открытие файла

Открытие файла выполняется одинаково для всех 3 потоков файлов (`ifstream` , `ofstream` и `fstream`).

Вы можете открыть файл непосредственно в конструкторе:

```
std::ifstream ifs("foo.txt"); // ifstream: Opens file "foo.txt" for reading only.
std::ofstream ofs("foo.txt"); // ofstream: Opens file "foo.txt" for writing only.
std::fstream iofs("foo.txt"); // fstream: Opens file "foo.txt" for reading and writing.
```

Кроме того, вы можете использовать функцию члена потока файлов `open()` :

```
std::ifstream ifs;
ifs.open("bar.txt");           // ifstream: Opens file "bar.txt" for reading only.

std::ofstream ofs;
ofs.open("bar.txt");           // ofstream: Opens file "bar.txt" for writing only.

std::fstream iofs;
iofs.open("bar.txt");          // fstream: Opens file "bar.txt" for reading and writing.
```

Вы **всегда** должны проверить, был ли файл успешно открыт (даже при написании). Ошибки могут включать: файл не существует, файл не имеет прав доступа, файл уже используется, произошли ошибки диска, отключен диск ... Проверка может быть выполнена следующим образом:

```
// Try to read the file 'foo.txt'.
std::ifstream ifs("fooo.txt"); // Note the typo; the file can't be opened.

// Check if the file has been opened successfully.
if (!ifs.is_open()) {
    // The file hasn't been opened; take appropriate actions here.
    throw CustomException(ifs, "File could not be opened");
}
```

Когда путь к файлу содержит обратную косую черту (например, в системе Windows), вы должны правильно их избежать:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs("c:\\folder\\foo.txt"); // using escaped backslashes
```

C ++ 11

или использовать исходный литерал:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs(R"(c:\folder\foo.txt)"); // using raw literal
```

или вместо этого используйте косые черты:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs("c:/folder/foo.txt");
```

C ++ 11

Если вы хотите открыть файл с не-ASCII-символами в пути в Windows, в настоящее время вы можете использовать **нестандартный** аргумент пути к широкому символу:

```
// Open the file 'пример\foo.txt' on Windows.
std::ifstream ifs(LR"(пример\foo.txt)"); // using wide characters with raw literal
```

Чтение из файла

Существует несколько способов чтения данных из файла.

Если вы знаете, как форматируются данные, вы можете использовать оператор извлечения потока (>>). Предположим, у вас есть файл с именем *foo.txt*, который содержит следующие данные:

```
John Doe 25 4 6 1987
```

```
Jane Doe 15 5 24 1976
```

Затем вы можете использовать следующий код для чтения этих данных из файла:

```
// Define variables.
std::ifstream is("foo.txt");
std::string firstname, lastname;
int age, bmonth, bday, byear;

// Extract firstname, lastname, age, bday month, bday day, and bday year in that order.
// Note: '>>' returns false if it reached EOF (end of file) or if the input data doesn't
// correspond to the type of the input variable (for example, the string "foo" can't be
// extracted into an 'int' variable).
while (is >> firstname >> lastname >> age >> bmonth >> bday >> byear)
    // Process the data that has been read.
```

Оператор извлечения потока `>>` извлекает каждый символ и останавливается, если он находит символ, который нельзя сохранить, или если он является особым символом:

- Для типов строк оператор останавливается в пробеле () или в новой строке (`\n`).
- Для чисел оператор останавливается с символом, отличным от числа.

Это означает, что следующая версия файла *foo.txt* также будет успешно прочитана предыдущим кодом:

```
John
Doe 25
4 6 1987
```

```
Jane
Doe
15 5
24
1976
```

Оператор извлечения потока `>>` всегда возвращает переданный ему поток. Поэтому несколько операторов могут быть соединены друг с другом, чтобы читать данные последовательно. Тем не менее, поток также может быть использован в качестве логического выражения (как показано в `while` петле в предыдущем коде). Это связано с тем, что классы потоков имеют оператор преобразования для типа `bool`. Этот оператор `bool()` вернет `true` пока поток не имеет ошибок. Если поток переходит в состояние ошибки (например, поскольку больше не может быть извлечено данных), то оператор `bool()` вернет `false`. Таким образом, в `while` цикл в предыдущем коде будет после того, как вышел из входного файла был прочитан до конца.

Если вы хотите прочитать весь файл в виде строки, вы можете использовать следующий код:

```
// Opens 'foo.txt'.
```

```
std::ifstream is("foo.txt");
std::string whole_file;

// Sets position to the end of the file.
is.seekg(0, std::ios::end);

// Reserves memory for the file.
whole_file.reserve(is.tellg());

// Sets position to the start of the file.
is.seekg(0, std::ios::beg);

// Sets contents of 'whole_file' to all characters in the file.
whole_file.assign(std::istreambuf_iterator<char>(is),
    std::istreambuf_iterator<char>());
```

Этот код резервирует пространство для `string`, чтобы сократить ненужные распределения памяти.

Если вы хотите прочитать файл по строкам, вы можете использовать функцию `getline()` :

```
std::ifstream is("foo.txt");

// The function getline returns false if there are no more lines.
for (std::string str; std::getline(is, str);) {
    // Process the line that has been read.
}
```

Если вы хотите прочитать фиксированное количество символов, вы можете использовать функцию члена потока `read()` :

```
std::ifstream is("foo.txt");
char str[4];

// Read 4 characters from the file.
is.read(str, 4);
```

После выполнения команды чтения, вы всегда должны проверить, если состояние ошибки флаг `failbit` был установлен, поскольку он указывает на то удалось ли операция или нет. Это можно сделать, вызвав функцию члена файлового потока `fail()` :

```
is.read(str, 4); // This operation might fail for any reason.

if (is.fail())
    // Failed to read!
```

Запись в файл

Существует несколько способов записи в файл. Самый простой способ - использовать поток выходных файлов (`ofstream`) вместе с оператором вставки потока (`<<`):

```
std::ofstream os("foo.txt");
```

```
if(os.is_open()){
    os << "Hello World!";
}
```

Вместо « << вы также можете использовать функцию члена потока выходного файла `write()` :

```
std::ofstream os("foo.txt");
if(os.is_open()){
    char data[] = "Foo";

    // Writes 3 characters from data -> "Foo".
    os.write(data, 3);
}
```

После записи в поток, вы всегда должны проверить , если состояние ошибки флаг `badbit` был установлен, поскольку он указывает на то удалось ли операция или нет. Это можно сделать, вызвав функцию члена потока выходного файла `bad()` :

```
os << "Hello Badbit!"; // This operation might fail for any reason.
if (os.bad())
    // Failed to write!
```

Режимы открытия

При создании потока файлов вы можете указать режим открытия. Режим открытия - это, в основном, параметр для управления тем, как поток открывает файл.

(Все режимы можно найти в пространстве имен `std::ios` .)

Режим открытия может быть предоставлен в качестве второго параметра конструктору файлового потока или его функции `open()` :

```
std::ofstream os("foo.txt", std::ios::out | std::ios::trunc);

std::ifstream is;
is.open("foo.txt", std::ios::in | std::ios::binary);
```

Следует отметить, что вам нужно установить `ios::in` или `ios::out` если вы хотите установить другие флаги, поскольку они неявно устанавливаются членами `iostream`, хотя они имеют правильное значение по умолчанию.

Если вы не укажете режим открытия, используются следующие режимы по умолчанию:

- `ifstream` - `in`
- `ofstream` - `out`
- `fstream` - `in` И `out`

Режимы открытия файла, которые вы можете указать по дизайну:

Режим	Имея в виду	За	Описание
app	присоединять	Выход	Добавляет данные в конец файла.
binary	двоичный	Ввод, вывод	Вход и выход выполняются в двоичном формате.
in	вход	вход	Открывает файл для чтения.
out	выход	Выход	Открывает файл для записи.
trunc	усекать	Ввод, вывод	Удаляет содержимое файла при открытии.
ate	в конце	вход	Открывается до конца файла.

Примечание. Установка `binary` режима позволяет считывать / записывать данные в точности как есть; не устанавливая его, это позволяет переносить символ новой строки `'\n' / \` на определенный конец строки последовательности.

Например, в Windows конец строки - CRLF (`"\r\n"`).

Напишите: `"\n" => "\r\n"`

Читайте: `"\r\n" => "\n"`

Заккрытие файла

Явное закрытие файла редко необходимо в C ++, так как поток файлов автоматически закрывает связанный файл в своем деструкторе. Тем не менее, вы должны попытаться ограничить время жизни объекта потока файлов, чтобы он не закрывал дескриптор файла дольше, чем необходимо. Например, это можно сделать, поместив все операции с файлами в собственную область (`{ }`):

```
std::string const prepared_data = prepare_data();
{
    // Open a file for writing.
    std::ofstream output("foo.txt");

    // Write data.
    output << prepared_data;
} // The ofstream will go out of scope here.
// Its destructor will take care of closing the file properly.
```

Вызов `close()` явно необходимо , только если вы хотите использовать один и тот же `fstream` объект позже, но не хотите , чтобы сохранить файл открыть между ними:

```
// Open the file "foo.txt" for the first time.
std::ofstream output("foo.txt");

// Get some data to write from somewhere.
```



```

std::string const prepared_data = prepare_data();

// Write data to the file "foo.txt".
output << prepared_data;

// Close the file "foo.txt".
output.close();

// Preparing data might take a long time. Therefore, we don't open the output file stream
// before we actually can write some data to it.
std::string const more_prepared_data = prepare_complex_data();

// Open the file "foo.txt" for the second time once we are ready for writing.
output.open("foo.txt");

// Write the data to the file "foo.txt".
output << more_prepared_data;

// Close the file "foo.txt" once again.
output.close();

```

Промывка потока

Потоки файлов по умолчанию буферизуются, как и многие другие типы потоков. Это означает, что запись в поток не может привести к немедленному изменению базового файла. Чтобы заставить все буферизованные записи совершать сразу, вы можете *очистить* поток. Вы можете сделать это напрямую, вызывая метод `flush()` или с помощью манипулятора `std::flush` stream:

```

std::ofstream os("foo.txt");
os << "Hello World!" << std::flush;

char data[3] = "Foo";
os.write(data, 3);
os.flush();

```

Существует поток манипулятор `std::endl` который объединяет запись новой строки с потоком потока:

```

// Both following lines do the same thing
os << "Hello World!\n" << std::flush;
os << "Hello world!" << std::endl;

```

Буферизация может улучшить производительность записи в поток. Поэтому приложения, выполняющие много писем, должны избегать излишней очистки. Напротив, если операции ввода-вывода выполняются нечасто, приложения должны часто промывать, чтобы избежать застревания данных в объекте потока.

Чтение файла ASCII в строку std ::

```

std::ifstream f("file.txt");

```

```

if (f)
{
    std::stringstream buffer;
    buffer << f.rdbuf();
    f.close();

    // The content of "file.txt" is available in the string `buffer.str()`
}

```

Метод `rdbuf()` возвращает указатель на `streambuf` который может быть `streambuf` в `buffer` через `stringstream::operator<< member`.

Другая возможность (популярная в [Effective STL](#) от [Scott Meyers](#)):

```

std::ifstream f("file.txt");

if (f)
{
    std::string str((std::istreambuf_iterator<char>(f),
                std::istreambuf_iterator<char>()));

    // Operations on `str`...
}

```

Это хорошо, потому что требует небольшого кода (и позволяет читать файл непосредственно в любом контейнере STL, а не только в строках), но может быть медленным для больших файлов.

ПРИМЕЧАНИЕ . Дополнительные скобки вокруг первого аргумента конструктору строк необходимы для предотвращения *наиболее неприятной* проблемы *синтаксического анализа* .

Последний, но тем не менее важный:

```

std::ifstream f("file.txt");

if (f)
{
    f.seekg(0, std::ios::end);
    const auto size = f.tellg();

    std::string str(size, ' ');
    f.seekg(0);
    f.read(&str[0], size);
    f.close();

    // Operations on `str`...
}

```

что, вероятно, является самым быстрым вариантом (из трех предложенных).

Чтение файла в контейнер

В приведенном ниже примере мы используем `std::string` и `operator>>` для чтения элементов из файла.

```
std::ifstream file("file3.txt");

std::vector<std::string> v;

std::string s;
while(file >> s) // keep reading until we run out
{
    v.push_back(s);
}
```

В приведенном выше примере мы просто итерации через файл, читающий один «элемент» за раз, используя `operator>>`. Этот же `std::istream_iterator` может быть достигнут с помощью `std::istream_iterator` который является итератором ввода, который считывает один «элемент» за раз из потока. Также большинство контейнеров можно построить с использованием двух итераторов, чтобы мы могли упростить приведенный выше код:

```
std::ifstream file("file3.txt");

std::vector<std::string> v(std::istream_iterator<std::string>{file},
                          std::istream_iterator<std::string>{});
```

Мы можем расширить это, чтобы прочитать любые типы объектов, которые нам нравятся, просто указав объект, который мы хотим прочитать как параметр шаблона, для `std::istream_iterator`. Таким образом, мы можем просто расширить сказанное выше, чтобы читать строки (а не слова) следующим образом:

```
// Unfortunately there is no built in type that reads line using >>
// So here we build a simple helper class to do it. That will convert
// back to a string when used in string context.
struct Line
{
    // Store data here
    std::string data;
    // Convert object to string
    operator std::string const&() const {return data;}
    // Read a line from a stream.
    friend std::istream& operator>>(std::istream& stream, Line& line)
    {
        return std::getline(stream, line.data);
    }
};

std::ifstream file("file3.txt");

// Read the lines of a file into a container.
std::vector<std::string> v(std::istream_iterator<Line>{file},
                          std::istream_iterator<Line>{});
```

Чтение `struct` из форматированного текстового файла.

C++ 11

```
struct info_type
{
    std::string name;
    int age;
    float height;

    // we define an overload of operator>> as a friend function which
    // gives in privileged access to private data members
    friend std::istream& operator>>(std::istream& is, info_type& info)
    {
        // skip whitespace
        is >> std::ws;
        std::getline(is, info.name);
        is >> info.age;
        is >> info.height;
        return is;
    }
};

void func4()
{
    auto file = std::ifstream("file4.txt");

    std::vector<info_type> v;

    for(info_type info; file >> info;) // keep reading until we run out
    {
        // we only get here if the read succeeded
        v.push_back(info);
    }

    for(auto const& info: v)
    {
        std::cout << "  name: " << info.name << '\n';
        std::cout << "    age: " << info.age << " years" << '\n';
        std::cout << "height: " << info.height << "lbs" << '\n';
        std::cout << '\n';
    }
}
```

file4.txt

```
Wogger Wabbit
2
6.2
Bilbo Baggins
111
81.3
Mary Poppins
29
154.8
```

Выход:

```
name: Wogger Wabbit
age: 2 years
height: 6.2lbs

name: Bilbo Baggins
age: 111 years
height: 81.3lbs

name: Mary Poppins
age: 29 years
height: 154.8lbs
```

Копирование файла

```
std::ifstream src("source_filename", std::ios::binary);
std::ofstream dst("dest_filename", std::ios::binary);
dst << src.rdbuf();
```

C ++ 17

С C ++ 17 стандартным способом копирования файла является заголовок `<filesystem>` и использование `copy_file`:

```
std::filesystem::copy_file("source_filename", "dest_filename");
```

Библиотека файловой системы была первоначально разработана как `boost.filesystem` и, наконец, объединена с ISO C ++ с C ++ 17.

Проверка конца файла внутри условия цикла, неправильная практика?

`eof` возвращает `true` только **после** чтения конца файла. Он НЕ указывает, что следующее чтение будет концом потока.

```
while (!f.eof())
{
    // Everything is OK

    f >> buffer;

    // What if *only* now the eof / fail bit is set?

    /* Use `buffer` */
}
```

Вы можете правильно написать:

```
while (!f.eof())
{
    f >> buffer >> std::ws;

    if (f.fail())
        break;
```

```
/* Use `buffer` */
}
```

НО

```
while (f >> buffer)
{
    /* Use `buffer` */
}
```

проще и меньше подверженности ошибкам.

Дальнейшие ссылки:

- `std::ws` : отбрасывает ведущие пробелы из входного потока
- `std::basic_ios::fail` : возвращает `true` если произошла ошибка в связанном потоке

Запись файлов с нестандартными языковыми настройками

Если вам нужно записать файл с использованием разных настроек языкового стандарта по умолчанию, вы можете использовать `std::locale` и `std::basic_ios::imbue()` чтобы сделать это для определенного потока файлов:

Руководство для использования:

- Перед открытием файла вы всегда должны применять локальный поток.
- Как только поток будет пропитан, вы не должны изменять языковой стандарт.

Причины ограничений: при создании файлового потока с локалью имеет неопределенное поведение, если текущая локаль не является независимой от состояния или не указывает на начало файла.

Потоки UTF-8 (и другие) не являются независимыми от состояния. Также поток файлов с локалью UTF-8 может попробовать и прочитать маркер спецификации из файла при его открытии; поэтому просто открытие файла может читать символы из файла, и оно не будет в начале.

```
#include <iostream>
#include <fstream>
#include <locale>

int main()
{
    std::cout << "User-preferred locale setting is "
               << std::locale("").name().c_str() << std::endl;

    // Write a floating-point value using the user's preferred locale.
    std::ofstream ofsl;
    ofsl.imbue(std::locale(""));
}
```

```

ofsf1.open("file1.txt");
ofsf1 << 78123.456 << std::endl;

// Use a specific locale (names are system-dependent)
std::ofstream ofs2;
ofs2.imbue(std::locale("en_US.UTF-8"));
ofs2.open("file2.txt");
ofs2 << 78123.456 << std::endl;

// Switch to the classic "C" locale
std::ofstream ofs3;
ofs3.imbue(std::locale::classic());
ofs3.open("file3.txt");
ofs3 << 78123.456 << std::endl;
}

```

Явное переключение на классический язык «C» полезно, если ваша программа использует другой стандарт по умолчанию, и вы хотите обеспечить фиксированный стандарт для чтения и записи файлов. С предпочтительной локалью «C» пример записывает

```

78,123.456
78,123.456
78123.456

```

Если, например, предпочтительный языковой стандарт является немецким и, следовательно, использует другой формат чисел, пример записывает

```

78 123,456
78,123.456
78123.456

```

(обратите внимание на десятичную запятую в первой строке).

Прочитайте **Файловый ввод-вывод онлайн**: <https://riptutorial.com/ru/cplusplus/topic/496/файловый-ввод-вывод>

глава 136: Файлы заголовков

замечания

В C++, как и в C, компилятор C++ и процесс компиляции используют препроцессор C. Как указано в руководстве для препроцессора GNU C, заголовочный файл определяется следующим образом:

Заголовочный файл представляет собой файл, содержащий объявления C и определения макросов (см. Макросы) для совместного использования нескольких исходных файлов. Вы запрашиваете использование файла заголовка в своей программе, включив его, с директивой предварительной обработки C «`#include`».

Файлы заголовков служат для двух целей.

- Файлы заголовков системы объявляют интерфейсы к частям операционной системы. Вы включаете их в свою программу для предоставления определений и деклараций, необходимых для вызова системных вызовов и библиотек.
- Ваши собственные заголовочные файлы содержат декларации для интерфейсов между исходными файлами вашей программы. Каждый раз, когда у вас есть группа связанных деклараций и макроопределений, все или большинство из которых необходимы в нескольких разных исходных файлах, рекомендуется создать для них заголовочный файл.

Однако для самого препроцессора C заголовочный файл не отличается от исходного файла.

Схема организации заголовка / исходного файла представляет собой просто строгое и стандартное соглашение, устанавливаемое различными проектами программного обеспечения, чтобы обеспечить разделение между интерфейсом и реализацией.

Несмотря на то, что сам стандарт C++ официально не применяется, рекомендуется использовать правило заголовка / исходного файла, и на практике оно уже почти повсеместно.

Обратите внимание, что заголовочные файлы могут быть заменены как соглашение о структуре файла проекта с помощью предстоящей функции модулей, которая по-прежнему должна рассматриваться для включения в будущий стандарт C++ на момент написания (например, C++ 20).

Examples

Основной пример

Следующий пример будет содержать блок кода, который должен быть разделен на несколько исходных файлов, как обозначается `// filename`.

Исходные файлы

```
// my_function.h

/* Note how this header contains only a declaration of a function.
 * Header functions usually do not define implementations for declarations
 * unless code must be further processed at compile time, as in templates.
 */

/* Also, usually header files include preprocessor guards so that every header
 * is never included twice.
 *
 * The guard is implemented by checking if a header-file unique preprocessor
 * token is defined, and only including the header if it hasn't been included
 * once before.
 */
#ifndef MY_FUNCTION_H
#define MY_FUNCTION_H

// global_value and my_function() will be
// recognized as the same constructs if this header is included by different files.
const int global_value = 42;
int my_function();

#endif // MY_FUNCTION_H
```

```
// my_function.cpp

/* Note how the corresponding source file for the header includes the interface
 * defined in the header so that the compiler is aware of what the source file is
 * implementing.
 *
 * In this case, the source file requires knowledge of the global constant
 * global_value only defined in my_function.h. Without inclusion of the header
 * file, this source file would not compile.
 */
#include "my_function.h" // or #include "my_function.hpp"
int my_function() {
    return global_value; // return 42;
}
```

Заголовочные файлы затем включаются в другие исходные файлы, которые хотят использовать функциональные возможности, определенные интерфейсом заголовка, но не требуют знания его реализации (таким образом, уменьшая связь кода). Следующая программа использует заголовок `my_function.h` как указано выше:

```
// main.cpp

#include <iostream>          // A C++ Standard Library header.
#include "my_function.h"    // A personal header

int main(int argc, char** argv) {
    std::cout << my_function() << std::endl;
    return 0;
}
```

Процесс компиляции

Поскольку файлы заголовков часто являются частью процесса обработки процесса компиляции, типичный процесс компиляции, использующий соглашение заголовка / исходного файла, обычно выполняет следующее.

Предполагая, что файл заголовка и файл исходного кода уже находятся в одном каталоге, программист выполнит следующие команды:

```
g++ -c my_function.cpp      # Compiles the source file my_function.cpp
                             # --> object file my_function.o

g++ main.cpp my_function.o  # Links the object file containing the
                             # implementation of int my_function()
                             # to the compiled, object version of main.cpp
                             # and then produces the final executable a.out
```

В качестве альтернативы, если вы хотите сначала скомпилировать `main.cpp` в объектный файл, а затем связать только файлы объектов вместе в качестве последнего шага:

```
g++ -c my_function.cpp
g++ -c main.cpp

g++ main.o my_function.o
```

Шаблоны в файлах заголовков

Шаблонам требуется генерация кода во время компиляции: например, шаблонная функция будет эффективно превращена в несколько различных функций, как только параметризованная функция будет параметризоваться, используя исходный код.

Это означает, что функция шаблона, функция-член и определения классов не могут быть делегированы в отдельный файл исходного кода, так как любой код, который будет использовать любую шаблонную конструкцию, требует знания своего определения, как правило, генерирующего любой производный код.

Таким образом, шаблонный код, если он помещен в заголовки, также должен содержать

его определение. Ниже приведен пример:

```
// templated_function.h

template <typename T>
T* null_T_pointer() {
    T* type_point = NULL; // or, alternatively, nullptr instead of NULL
                          // for C++11 or later
    return type_point;
}
```

Прочитайте Файлы заголовков онлайн: <https://riptutorial.com/ru/cplusplus/topic/7211/файлы-заголовков>

глава 137: Функции нестатического пользователя

Синтаксис

- // Вызов:
 - `variable.member_function ()`;
 - `variable_pointer-> member_function ()`;
- // Определение:
 - `ret_type class_name :: member_function () cv-qualifiers {`
 - тело;
 - `}`
- // Прототип:
 - `class class_name {`
 - `virt-specifier ret_type member_function () cv-qualifiers virt-specifier-seq;`
 - // virt-specifier: «virtual», если применимо.
 - // cv-qualifiers: «const» и / или «volatile», если применимо.
 - // virt-specifier-seq: «переопределить» и / или «final», если применимо.
 - `}`

замечания

Не-`static` функция-член является `class / struct / union` функция, которая вызывается на конкретном случае, и действует на указанный экземпляр. В отличие от `static` функций-членов, он не может быть вызван без указания экземпляра.

Информацию о классах, структурах и объединениях см. [В родительской теме](#).

Examples

Нестатические функции-члены

`class` или `struct` могут иметь функции-члены, а также переменные-члены. Эти функции имеют синтаксис, в основном похожий на автономные функции, и могут быть определены либо внутри, либо вне определения класса; если определено вне определения класса, имя функции имеет префикс имени класса и оператора области (`::`).

```
class CL {
public:
    void definedInside() {}
    void definedOutside();
};
void CL::definedOutside() {}
```

Эти функции вызываются в экземпляре (или ссылке на экземпляр) класса с помощью оператора точки (.) Или указателя на экземпляр с помощью оператора стрелки (->), и каждый вызов привязывается к экземпляру, был вызван; когда функция-член вызывается в экземпляре, он имеет доступ ко всем полям этого экземпляра (через [this указатель](#)), но может обращаться только к полям других экземпляров, если эти экземпляры поставляются в качестве параметров.

```
struct ST {
    ST(const std::string& ss = "Wolf", int ii = 359) : s(ss), i(ii) { }

    int get_i() const { return i; }
    bool compare_i(const ST& other) const { return (i == other.i); }

private:
    std::string s;
    int i;
};
ST st1;
ST st2("Species", 8472);

int i = st1.get_i(); // Can access st1.i, but not st2.i.
bool b = st1.compare_i(st2); // Can access st1 & st2.
```

Этим функциям разрешен доступ к переменным-членам и / или другим функциям-членам, независимо от модификаторов доступа к переменной или функции. Они также могут быть записаны вне порядка, доступа к переменным-членам и / или вызовам функций-членов, объявленных перед ними, поскольку все определение класса должно анализироваться до того, как компилятор сможет начать компилировать класс.

```
class Access {
public:
    Access(int i_ = 8088, int j_ = 8086, int k_ = 6502) : i(i_), j(j_), k(k_) {}

    int i;
    int get_k() const { return k; }
    bool private_no_more() const { return i_be_private(); }
protected:
    int j;
    int get_i() const { return i; }
private:
    int k;
    int get_j() const { return j; }
    bool i_be_private() const { return ((i > j) && (k < j)); }
};
```

Инкапсуляция

Обычное использование функций-членов заключается в инкапсуляции, используя *аксессуар* (обычно известный как геттер) и *мутатор* (обычно известный как сеттер) вместо прямого доступа к полям.

```
class Encapsulator {
    int encapsulated;

public:
    int get_encapsulated() const { return encapsulated; }
    void set_encapsulated(int e) { encapsulated = e; }

    void some_func() {
        do_something_with(encapsulated);
    }
};
```

Внутри класса `encapsulated` могут быть свободно доступны любой нестатической функцией-членом; вне класса доступ к нему регулируется функциями-членами, используя `get_encapsulated()` для его чтения и `set_encapsulated()` для его изменения. Это предотвращает непреднамеренные изменения переменной, поскольку для ее чтения и записи используются отдельные функции. [Есть много дискуссий о том, предоставляют ли геттеры и сеттеры или прерывают инкапсуляцию, с хорошими аргументами для обеих претензий; такие жаркие споры выходят за рамки этого примера.]

Имя Скрытие и импорт

Когда базовый класс предоставляет набор перегруженных функций, а производный класс добавляет еще одну перегрузку в набор, это скрывает все перегрузки, предоставляемые базовым классом.

```
struct HiddenBase {
    void f(int) { std::cout << "int" << std::endl; }
    void f(bool) { std::cout << "bool" << std::endl; }
    void f(std::string) { std::cout << "std::string" << std::endl; }
};

struct HidingDerived : HiddenBase {
    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HiddenBase hb;
HidingDerived hd;
std::string s;

hb.f(1); // Output: int
hb.f(true); // Output: bool
hb.f(s); // Output: std::string;

hd.f(1.f); // Output: float
hd.f(3); // Output: float
hd.f(true); // Output: float
```

```
hd.f(s);    // Error: Can't convert from std::string to float.
```

Это связано с правилами разрешения имен: во время поиска имени, как только правильное имя найдено, мы перестаем искать, даже если мы явно не нашли правильную *версию* объекта с таким именем (например, с `hd.f(s)`); из-за этого перегрузка функции в производном классе предотвращает поиск имени из обнаружения перегрузок в базовом классе. Чтобы избежать этого, использование-объявления можно использовать для «импорта» имен из базового класса в производный класс, чтобы они были доступны во время поиска имени.

```
struct HidingDerived : HiddenBase {
    // All members named HiddenBase::f shall be considered members of HidingDerived for
    lookup.
    using HiddenBase::f;

    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HidingDerived hd;

hd.f(1.f); // Output: float
hd.f(3);   // Output: int
hd.f(true); // Output: bool
hd.f(s);   // Output: std::string
```

Если производный класс импортирует имена с использованием объявления-объявления, но также объявляет функции с той же сигнатурой, что и функции в базовом классе, функции базового класса молча будут переопределены или скрыты.

```
struct NamesHidden {
    virtual void hide_me()      {}
    virtual void hide_me(float) {}
    void hide_me(int)           {}
    void hide_me(bool)          {}
};

struct NameHider : NamesHidden {
    using NamesHidden::hide_me;

    void hide_me()      {} // Overrides NamesHidden::hide_me().
    void hide_me(int) {} // Hides NamesHidden::hide_me(int).
};
```

Использование-декларация также может использоваться для изменения модификаторов доступа, если импортированный объект был `public` или `protected` в базовом классе.

```
struct ProMem {
    protected:
    void func() {}
};
```

```

struct BecomesPub : ProMem {
    using ProMem::func;
};

// ...

ProMem pm;
BecomesPub bp;

pm.func(); // Error: protected.
bp.func(); // Good.

```

Аналогично, если мы явно хотим вызвать функцию-член из определенного класса в иерархии наследования, мы можем квалифицировать имя функции при вызове функции, указав этот класс по имени.

```

struct One {
    virtual void f() { std::cout << "One." << std::endl; }
};

struct Two : One {
    void f() override {
        One::f(); // this->One::f();
        std::cout << "Two." << std::endl;
    }
};

struct Three : Two {
    void f() override {
        Two::f(); // this->Two::f();
        std::cout << "Three." << std::endl;
    }
};

// ...

Three t;

t.f();          // Normal syntax.
t.Two::f();     // Calls version of f() defined in Two.
t.One::f();     // Calls version of f() defined in One.

```

Виртуальные функции участника

Функции-члены также могут быть объявлены `virtual`. В этом случае, если вы вызываете указатель или ссылаетесь на экземпляр, они не будут доступны напрямую; скорее, они будут искать функцию в таблице виртуальных функций (список указателей на `vtable` - функции для виртуальных функций, более широко известный как `vtable` или `vftable`), и использовать это для вызова версии, соответствующей динамике экземпляра (фактического) типа. Если функция вызывается напрямую, из переменной класса не выполняется поиск.

```

struct Base {
    virtual void func() { std::cout << "In Base." << std::endl; }
}

```



```
};

struct Derived : Base {
    void func() override { std::cout << "In Derived." << std::endl; }
};

void slicer(Base x) { x.func(); }

// ...

Base b;
Derived d;

Base *pb = &b, *pd = &d; // Pointers.
Base &rb = b, &rd = d;   // References.

b.func();    // Output:  In Base.
d.func();    // Output:  In Derived.

pb->func();   // Output:  In Base.
pd->func();   // Output:  In Derived.

rb.func();   // Output:  In Base.
rd.func();   // Output:  In Derived.

slicer(b);   // Output:  In Base.
slicer(d);   // Output:  In Base.
```

Обратите внимание, что в то время как `pd Base*` , а `rd - Base&` , вызывающий `func()` для любого из двух вызовов `Derived::func()` вместо `Base::func()` ; это связано с тем, что `vtable` для `Derived` обновляет запись `Base::func()` вместо этого указывает на `Derived::func()` . И наоборот, обратите внимание, что передача экземпляра `slicer()` всегда приводит к `slicer()` `Base::func()` , даже когда переданный экземпляр является `Derived` ; это происходит из-за того, что называется *срезом данных* , где передача экземпляра `Derived` в `Base` параметр по значению делает часть экземпляра `Derived` которая не является недоступной для `Base` экземпляра.

Когда функция-член определена как виртуальная, все функции-члены производного класса с одной и той же сигнатурой переопределяют ее, независимо от того, указана ли функция переопределения как `virtual` или нет. Это может сделать производные классы более трудными для программистов, чтобы разобраться, однако, поскольку нет никаких указаний относительно того, какие функции (ы) являются `virtual` .

```
struct B {
    virtual void f() {}
};

struct D : B {
    void f() {} // Implicitly virtual, overrides B::f.
                // You'd have to check B to know that, though.
};
```

Обратите внимание, однако, что производная функция только переопределяет базовую

функцию, если их подписи совпадают; даже если производная функция явно объявлена `virtual`, она вместо этого создаст новую виртуальную функцию, если сигнатуры будут несогласованными.

```
struct BadB {
    virtual void f() {}
};

struct BadD : BadB {
    virtual void f(int i) {} // Does NOT override BadB::f.
};
```

C ++ 11

На C ++ 11, намерение переопределения может быть явно с контекстно-зависимым ключевым словом `override`. Это говорит компилятору, что программист ожидает, что он переопределит функцию базового класса, что заставляет компилятор пропускать ошибку, если она *не* отменяет ничего.

```
struct CPP11B {
    virtual void f() {}
};

struct CPP11D : CPP11B {
    void f() override {}
    void f(int i) override {} // Error: Doesn't actually override anything.
};
```

Это также дает возможность говорить программистам, что функция является виртуальной, а также объявляется, по крайней мере, в одном базовом классе, что упрощает анализ сложных классов.

Когда функция объявляется `virtual` и определяется вне определения класса, `virtual` спецификатор должен быть включен в объявление функции и не повторяться в определении.

C ++ 11

Это также справедливо для `override`.

```
struct VB {
    virtual void f(); // "virtual" goes here.
    void g();
};
/* virtual */ void VB::f() {} // Not here.
virtual void VB::g() {} // Error.
```

Если базовый класс перегружает `virtual` функцию, то только перегрузки, явно указанные как `virtual` будут виртуальными.

```
struct BOverload {
```

```

    virtual void func() {}
    void func(int) {}
};

struct DOverload : BOverload {
    void func() override {}
    void func(int) {}
};

// ...

BOverload* bo = new DOverload;
bo->func(); // Calls DOverload::func().
bo->func(1); // Calls BOverload::func(int).

```

Для получения дополнительной информации см. [Соответствующую тему](#).

Const Correctness

Одним из основных применений для `this` CV-классификаторов является *const корректности*. Это практика гарантирования того, что только доступ, который *должен* изменить объект, *способен* модифицировать объект и что любая (членная или нечлена) функция, которая не нуждается в изменении объекта, не имеет права на запись к этому объекту (прямо или косвенно). Это предотвращает непреднамеренные изменения, делая код менее ошибочным. Он также позволяет любой функции, которая не нуждается в изменении состояния, чтобы иметь возможность принимать объект `const` или не-`const`, без необходимости переписывать или перегружать функцию.

`const` по своей природе начинается снизу вверх: любая функция-член класса, которая не нуждается в изменении состояния, *объявляется как const*, поэтому ее можно вызвать в экземплярах `const`. Это, в свою очередь, позволяет объявлять `const`-by-reference, когда они не нуждаются в модификации, что позволяет функциям принимать либо `const` либо не-`const` объекты без жалобы, а `const`-ness может распространяться наружу в этом манера. Из-за этого геттеры часто являются `const`, как и любые другие функции, которые не нуждаются в изменении логического состояния.

```

class ConstIncorrect {
    Field fld;

public:
    ConstIncorrect(const Field& f) : fld(f) {} // Modifies.

    const Field& get_field() { return fld; } // Doesn't modify; should be const.
    void set_field(const Field& f) { fld = f; } // Modifies.

    void do_something(int i) { // Modifies.
        fld.insert_value(i);
    }
    void do_nothing() { } // Doesn't modify; should be const.
};

class ConstCorrect {

```

```

    Field fld;

public:
    ConstCorrect(const Field& f) : fld(f) {}           // Not const: Modifies.

    const Field& get_field() const { return fld; } // const: Doesn't modify.
    void set_field(const Field& f) { fld = f; }       // Not const: Modifies.

    void do_something(int i) {                         // Not const: Modifies.
        fld.insert_value(i);
    }
    void do_nothing() const { }                       // const: Doesn't modify.
};

// ...

const ConstIncorrect i_cant_do_anything(make_me_a_field());
// Now, let's read it...
Field f = i_cant_do_anything.get_field();
// Error: Loses cv-qualifiers, get_field() isn't const.
i_cant_do_anything.do_nothing();
// Error: Same as above.
// Oops.

const ConstCorrect but_i_can(make_me_a_field());
// Now, let's read it...
Field f = but_i_can.get_field(); // Good.
but_i_can.do_nothing();          // Good.

```

Как показано в комментариях к `ConstIncorrect` и `ConstCorrect`, надлежащим образом функции cv-qualifying также служат документацией. Если класс `const` корректен, любая функция, которая не является `const` можно смело предположить, чтобы изменить состояние, и любая функция, которая является `const` можно смело предположить, что она не изменяет состояние.

Прочитайте [Функции нестатического пользователя онлайн](https://riptutorial.com/ru/cplusplus/topic/5661/функции-нестатического-пользователя):

<https://riptutorial.com/ru/cplusplus/topic/5661/функции-нестатического-пользователя>

глава 138: Функция C ++ «вызов по значению» против «вызов по ссылке»

Вступление

Сфера применения этого раздела заключается в объяснении различий в теории и реализации того, что происходит с параметрами функции при вызове.

Подробно параметры можно рассматривать как переменные перед вызовом функции и внутри функции, где видимое поведение и доступность этих переменных различаются с методом, используемым для их передачи.

Кроме того, повторное использование переменных и их соответствующих значений после вызова функции также объясняется этой темой.

Examples

Вызов по значению

При вызове функции в стеке программы появляются новые элементы. К ним относятся некоторая информация о функции, а также о пространстве (ячейках памяти) для параметров и возвращаемом значении.

При передаче параметра функции значение используемой переменной (или литерала) копируется в ячейку памяти параметра функции. Это означает, что теперь есть две ячейки памяти с одинаковым значением. Внутри функции мы работаем только в области памяти параметров.

После выхода из функции выпадающая (удаленная) память в стеке программы удаляет все данные вызова функции, включая расположение памяти параметров, которые мы использовали внутри. Таким образом, значения, измененные внутри функции, не влияют на значения внешних переменных.

```
int func(int f, int b) {  
    //new variables are created and values from the outside copied  
    //f has a value of 0  
    //inner_b has a value of 1  
    f = 1;  
    //f has a value of 1  
    b = 2;  
    //inner_b has a value of 2  
    return f+b;  
}  
  
int main(void) {
```

```

int a = 0;
int b = 1; //outer_b
int c;

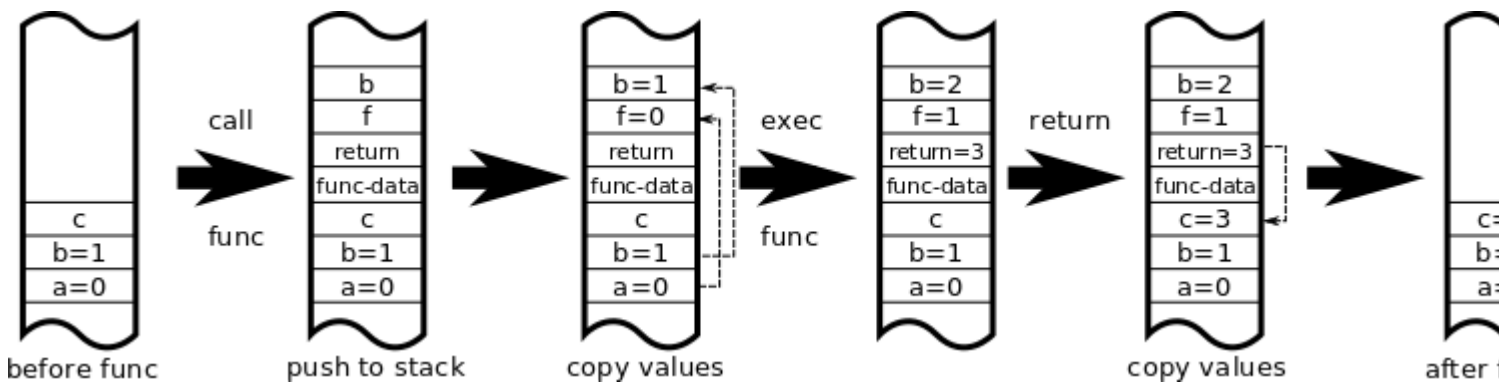
c = func(a,b);
//the return value is copied to c

//a has a value of 0
//outer_b has a value of 1    <--- outer_b and inner_b are different variables
//c has a value of 3
}

```

В этом коде мы создаем переменные внутри основной функции. Они получают назначенные значения. При вызове функций создаются две новые переменные: `f` и `inner_b` где `b` разделяет имя с внешней переменной, которая не разделяет память. Поведение `a<->f` и `b<->b` идентично.

Следующий рисунок символизирует, что происходит в стеке и почему нет изменений в `variabale b`. Графика не полностью точна, но подчеркивает пример.



Он называется «вызов по значению», потому что мы не передаем переменные, а только значения этих переменных.

Прочитайте [Функция C ++ «вызов по значению» против «вызов по ссылке» онлайн: https://riptutorial.com/ru/cplusplus/topic/10669/функция-c-plusplus--вызов-по-значению--против--вызов-по-ссылке-](https://riptutorial.com/ru/cplusplus/topic/10669/функция-c-plusplus--вызов-по-значению--против--вызов-по-ссылке-)

глава 139: Фьючерсы и обещания

Вступление

Обещания и фьючерсы используются для пересылки одного объекта из одного потока в другой.

Объект `std::promise` задается потоком, который генерирует результат.

Объект `std::future` можно использовать для извлечения значения, для проверки наличия доступного значения или для остановки выполнения до тех пор, пока значение не будет доступно.

Examples

`std::future` и `std::prom`

В следующем примере обещает быть использовано другим потоком:

```
{
    auto promise = std::promise<std::string>();

    auto producer = std::thread([&]
    {
        promise.set_value("Hello World");
    });

    auto future = promise.get_future();

    auto consumer = std::thread([&]
    {
        std::cout << future.get();
    });

    producer.join();
    consumer.join();
}
```

Пример отложенного асинхронного

Этот код реализует версию `std::async`, но он ведет себя так, как будто `async` всегда вызывался с `deferred` политикой запуска. Эта функция также не имеет специального поведения `async future`; возвращенное `future` может быть уничтожено без его приобретения.

```
template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
```

```

using result_type = decltype(func());

auto promise = std::promise<result_type>();
auto future  = promise.get_future();

std::thread(std::bind( [=] (std::promise<result_type>& promise)
{
    try
    {
        promise.set_value(func());
        // Note: Will not work with std::promise<void>. Needs some meta-template
programming which is out of scope for this example.
    }
    catch(...)
    {
        promise.set_exception(std::current_exception());
    }
}, std::move(promise))).detach();

return future;
}

```

std::packaged_task и std::future

std::packaged_task **связывает** функцию и связанное с ней обещание для возвращаемого типа:

```

template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    auto task    = std::packaged_task<decltype(func()) ()>(std::forward<F>(func));
    auto future = task.get_future();

    std::thread(std::move(task)).detach();

    return std::move(future);
}

```

Поток запускается немедленно. Мы можем либо отделить его, либо присоединиться к нему в конце сферы действия. Когда вызов функции в std::thread завершается, результат будет готов.

Обратите внимание, что это немного отличается от std::async где возвращаемый std::future при уничтожении фактически **блокируется** до тех пор, пока поток не будет завершен.

std::future_error и std::future_errc

Если ограничения для std::prom и std::future не выполняются, возникает исключение типа std::future_error.

Член кода ошибки в исключении относится к типу std::future_errc, а значения приведены ниже, а также некоторые тестовые примеры:


```
enum class future_errc {
    broken_promise           = /* the task is no longer shared */,
    future_already_retrieved = /* the answer was already retrieved */,
    promise_already_satisfied = /* the answer was stored already */,
    no_state                  = /* access to a promise in non-shared state */
};
```

Неактивное обещание:

```
int test()
{
    std::promise<int> pr;
    return 0; // returns ok
}
```

Активные обещания, неиспользованные:

```
int test()
{
    std::promise<int> pr;
    auto fut = pr.get_future(); //blocks indefinitely!
    return 0;
}
```

Двойной поиск:

```
int test()
{
    std::promise<int> pr;
    auto fut1 = pr.get_future();

    try{
        auto fut2 = pr.get_future();    //    second attempt to get future
        return 0;
    }
    catch(const std::future_error& e)
    {
        cout << e.what() << endl;    //    Error: "The future has already been retrieved
from the promise or packaged_task."
        return -1;
    }
    return fut2.get();
}
```

Установка значения std :: prom дважды:

```
int test()
{
    std::promise<int> pr;
    auto fut = pr.get_future();
    try{
        std::promise<int> pr2(std::move(pr));
        pr2.set_value(10);
        pr2.set_value(10); // second attempt to set promise throws exception
    }
    catch(const std::future_error& e)
```

```

{
    cout << e.what() << endl;          //   Error: "The state of the promise has already been
set."
    return -1;
}
return fut.get();
}

```

std::future и std::async

В следующем примере наивного параллельного сопоставления слияния `std::async` используется для запуска нескольких параллельных задач `merge_sort`. `std::future` используется для ожидания результатов и их синхронизации:

```

#include <iostream>
using namespace std;

void merge(int low,int mid,int high, vector<int>&num)
{
    vector<int> copy(num.size());
    int h,i,j,k;
    h=low;
    i=low;
    j=mid+1;

    while( (h<=mid) && (j<=high) )
    {
        if (num[h]<=num[j])
        {
            copy[i]=num[h];
            h++;
        }
        else
        {
            copy[i]=num[j];
            j++;
        }
        i++;
    }
    if(h>mid)
    {
        for(k=j;k<=high;k++)
        {
            copy[i]=num[k];
            i++;
        }
    }
    else
    {
        for(k=h;k<=mid;k++)
        {
            copy[i]=num[k];
            i++;
        }
    }
    for(k=low;k<=high;k++)
        swap(num[k], copy[k]);
}

```

```

}

void merge_sort(int low,int high,vector<int>& num)
{
    int mid;
    if(low<high)
    {
        mid = low + (high-low)/2;
        auto future1    =  std::async(std::launch::deferred, [&]()
                                {
                                    merge_sort(low,mid,num);
                                });
        auto future2    =  std::async(std::launch::deferred, [&]()
                                {
                                    merge_sort(mid+1,high,num) ;
                                });

        future1.get();
        future2.get();
        merge(low,mid,high,num);
    }
}

```

Примечание. В примере `std::async` запускается с политикой `std::launch_deferred` . Это делается для предотвращения создания нового потока в каждом вызове. В случае нашего примера вызовы `std::async` выходят из строя, они синхронизируются при вызовах для `std::future::get()` .

`std::launch_async` заставляет новый поток создаваться в каждом вызове.

Политика по умолчанию: `std::launch::deferred` | `std::launch::async` , то есть реализация определяет политику для создания новых потоков.

Асинхронные классы операций

- `std::async`: выполняет асинхронную операцию.
- `std::future`: обеспечивает доступ к результатам асинхронной операции.
- `std::prom`: упаковывает результат асинхронной операции.
- `std::packaged_task`: связывает функцию и связанное с ней обещание для возвращаемого типа.

Прочитайте **Фьючерсы и обещания онлайн**: <https://riptutorial.com/ru/cplusplus/topic/9840/фьючерсы-и-обещания>

глава 140: Характеристики связи

Вступление

Спецификация привязки сообщает компилятору компилировать объявления таким образом, чтобы их можно было связывать вместе с декларациями, написанными на другом языке, например C.

Синтаксис

- `extern string-literal { declaration-seq ( opt ) }`
- `extern строкового литерала декларация`

замечания

Стандарт требует, чтобы все компиляторы поддерживали `extern "C"`, чтобы позволить C++ быть совместимым с C и `extern "C++"`, который может использоваться для переопределения спецификации привязки и восстановления значения по умолчанию. Другие поддерживаемые спецификации привязки определены [реализацией](#).

Examples

Обработчик сигналов для Unix-подобной операционной системы

Поскольку обработчик сигнала будет вызываться ядром с использованием соглашения о вызове C, мы должны сообщить компилятору использовать соглашение C-вызова при компиляции функции.

```
volatile sig_atomic_t death_signal = 0;
extern "C" void cleanup(int signum) {
    death_signal = signum;
}
int main() {
    bind(...);
    listen(...);
    signal(SIGTERM, cleanup);
    while (int fd = accept(...)) {
        if (fd == -1 && errno == EINTR && death_signal) {
            printf("Caught signal %d; shutting down\n", death_signal);
            break;
        }
        // ...
    }
}
```

Создание заголовка библиотеки C, совместимого с C ++

Заголовок библиотеки C обычно можно включить в программу на C ++, так как большинство объявлений действительны как для C, так и для C ++. Например, рассмотрим следующий `foo.h`:

```
typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);
```

Определение `make_foo` отдельно компилируется и распределяется с заголовком в виде объекта.

Программа C ++ может `#include <foo.h>`, но компилятор не будет знать, что функция `make_foo` определена как символ C и, вероятно, попытается найти ее с искаженным именем и не сможет ее найти. Даже если он может найти определение `make_foo` в библиотеке, не все платформы используют одни и те же соглашения о вызовах для C и C ++, а компилятор C ++ будет использовать соглашение о вызове C ++ при вызове `make_foo`, что может вызвать ошибку сегментации, если `make_foo` ожидается, что его вызывают с помощью соглашения о вызове C.

Способ устранения этой проблемы состоит в том, чтобы обернуть почти все объявления в заголовке во `extern "C"` блоке `extern "C"`.

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);

#ifdef __cplusplus
} /* end of "extern C" block */
#endif
```

Теперь, когда `foo.h` включен в программу C, он будет просто отображаться как обычные объявления, но когда `foo.h` будет включен из C ++-программы, `make_foo` будет находиться внутри `extern "C"` блока `extern "C"` и компилятор будет знать, что искать unmangled name и использовать соглашение вызова C.

Прочитайте Характеристики связи онлайн: <https://riptutorial.com/ru/cplusplus/topic/9268/характеристики-связи>

глава 141: центровка

Вступление

Все типы в C++ имеют выравнивание. Это ограничение на адрес памяти, в котором объекты этого типа могут быть созданы внутри. Адрес памяти действителен для создания объекта, если деление этого адреса на выравнивание объекта является целым числом.

Выравнивание типов всегда имеет силу двух (включая 1).

замечания

Стандарт гарантирует следующее:

- Требование к выравниванию типа является делителем его размера. Например, класс с размером 16 байт может иметь выравнивание 1, 2, 4, 8 или 16, но не 32. (Если члены класса имеют всего 14 байтов, но класс должен иметь требование выравнивания из 8, компилятор будет вставлять 2 байта заполнения, чтобы размер класса был равен 16.)
- Подписанные и неподписанные версии целочисленного типа имеют одинаковое требование выравнивания.
- Указатель на `void` имеет такое же требование выравнивания, как и указатель на `char`.
- Квалифицированные и cv-неквалифицированные версии типа имеют одинаковое требование выравнивания.

Обратите внимание, что, хотя выравнивание существует в C++ 03, только C++ 11 стало возможным запрашивать выравнивание (используя `alignof`) и выравнивание управления (используя `alignas`).

Examples

Запрос выравнивания типа

C++ 11

Требование выравнивания типа можно запросить с помощью `alignof` **слова** `alignof` в качестве унарного оператора. Результатом является постоянное выражение типа `std::size_t`, *то есть* оно может быть оценено во время компиляции.

```
#include <iostream>
int main() {
    std::cout << "The alignment requirement of int is: " << alignof(int) << '\n';
}
```

Возможный выход

Требование выравнивания `int`: 4

Если применяется к массиву, он дает требование выравнивания типа элемента. Если применяется к ссылочному типу, он дает требование выравнивания ссылочного типа. (У самих ссылок нет выравнивания, поскольку они не являются объектами.)

Управление выравниванием

C ++ 11

`alignas` **СЛОВО** `alignas` можно использовать для принудительной `alignas` переменной, члена данных класса, объявления или определения класса или объявления или определения перечисления, чтобы иметь конкретное выравнивание, если оно поддерживается. Это происходит в двух формах:

- `alignas(x)` , где `x` является константным выражением, дает объекту выравнивание `x` , если поддерживается.
- `alignas(T)` , где `T` - тип, дает объекту выравнивание, равное требованию выравнивания `T` , то есть `alignof(T)` , если поддерживается.

Если к `alignas` и тому же объекту применяются несколько `alignas` , применяется самый строгий.

В этом примере буфер `buf` гарантированно будет соответствующим образом выровнен для хранения объекта `int` , даже если его тип элемента является `unsigned char` , который может иметь более слабое требование выравнивания.

```
alignas(int) unsigned char buf[sizeof(int)];
new (buf) int(42);
```

`alignas` не может использоваться, чтобы дать типу меньшее выравнивание, чем тип имел бы без этого объявления:

```
alignas(1) int i; //Il-formed, unless `int` on this platform is aligned to 1 byte.
alignas(char) int j; //Il-formed, unless `int` has the same or smaller alignment than `char`.
```

`alignas` , если задано целочисленное постоянное выражение, должно быть дано правильное выравнивание. Допустимые выравнивания всегда равны двум и должны быть больше нуля. Компиляторы должны поддерживать все допустимые выравнивания до выравнивания типа `std::max_align_t` . Они *могут* поддерживать более крупные выравнивания, чем это, но поддержка выделения памяти для таких объектов ограничена. Верхний предел выравнивания зависит от реализации.

C ++ 17 поддерживает прямую поддержку `operator new` для выделения памяти для

выверенных типов.

Прочитайте центровка онлайн: <https://riptutorial.com/ru/cplusplus/topic/9249/центровка>

глава 142: Цифровые разделители

Examples

Цифровой разделитель

Числовые литералы из более чем нескольких цифр трудно читать.

- Произнести слово «7237498123».
- Сравните 237498123 с 237499123 за равенство.
- Решите, больше ли 237499123 или 20249472.

C++14 define Simple Quotation Mark ' как разделитель цифр, в числах и пользовательских литералах. Это может облегчить для читателей людей синтаксический анализ больших чисел.

C++ 14

```
long long decn = 1'000'000'000ll;  
long long hexn = 0xFFFF'FFFFll;  
long long octn = 00'23'00ll;  
long long binn = 0b1010'0011ll;
```

Одиночная кавычка игнорируется при определении ее значения.

Пример:

- Литералы 1048576 , 1'048'576 , 0x100000 , 0x10'0000 и 0'004'000'000 все имеют одинаковое значение.
- Литералы 1.602'176'565e-19 и 1.602176565e-19 имеют одинаковое значение.

Позиция одиночных кавычек не имеет значения. Все следующие эквиваленты:

C++ 14

```
long long a1 = 12345678911;  
long long a2 = 123'456'78911;  
long long a3 = 12'34'56'78'911;  
long long a4 = 12345'678911;
```

Это также разрешено в user-defined литералах:

C++ 14

```
std::chrono::seconds tiempo = 1'674'456s + 5'300h;
```

Прочитайте Цифровые разделители онлайн: <https://riptutorial.com/ru/cplusplus/topic/10595/>

глава 143: Шаблон дизайна Singleton

замечания

Синглтон предназначен для обеспечения того, чтобы класс имел только один экземпляр и обеспечивал глобальную точку доступа к нему. Если вам нужен только один экземпляр *или* удобная глобальная точка доступа, но не оба, рассмотрите другие варианты, прежде чем обращаться к синглтону.

Глобальные переменные *могут* затруднить использование кода. Например, если одна из вызывающих функций недовольна данными, полученными ею от Singleton, вам нужно сначала отследить, что сначала дает синглтонные плохие данные.

Синглтоны также поощряют **сцепление**, термин, используемый для описания двух компонентов кода, которые объединены вместе, таким образом уменьшая каждый компонент собственной мерой самоограничения.

Синглтоны не совместимы с параллелизмом. Когда класс имеет глобальную точку доступа, каждый поток имеет доступ к нему, что может привести к взаимоблокировкам и условиям гонки.

Наконец, ленивая инициализация может вызвать проблемы с производительностью при инициализации в неподходящее время. Удаление ленивой инициализации также устраняет некоторые функции, которые делают особенно интересным Singleton, например полиморфизм (см. Подкласс).

Источники: [Шаблоны программирования игр Роберта Нистрома](#)

Examples

Ленивая инициализация

Этот пример был снят с раздела Q & A: <http://stackoverflow.com/a/1008289/3807729>

См. Эту статью для простой конструкции для ленивой оценки с гарантированным разрушением singleton:

[Может ли кто-нибудь предоставить мне образец Singleton в c ++?](#)

Классический ленивый оцененный и правильно уничтоженный синглтон.

```
class S
{
    public:
        static S& getInstance()
```

```

    {
        static S      instance; // Guaranteed to be destroyed.
                                // Instantiated on first use.

        return instance;
    }
private:
    S() {}; // Constructor? (the {} brackets) are needed here.

    // C++ 03
    // =====
    // Dont forget to declare these two. You want to make sure they
    // are unacceptable otherwise you may accidentally get copies of
    // your singleton appearing.
    S(S const&); // Don't Implement
    void operator=(S const&); // Don't implement

    // C++ 11
    // =====
    // We can use the better technique of deleting the methods
    // we don't want.
public:
    S(S const&) = delete;
    void operator=(S const&) = delete;

    // Note: Scott Meyers mentions in his Effective Modern
    // C++ book, that deleted functions should generally
    // be public as it results in better error messages
    // due to the compilers behavior to check accessibility
    // before deleted status
};

```

См. Эту статью о том, когда использовать синглтон: (не часто)

[Синглтон: как его использовать](#)

См. Две статьи о порядке инициализации и о том, как справиться:

[Порядок инициализации статических переменных](#)

[Поиск статических задач инициализации C ++](#)

См. Эту статью, описывающую сроки жизни:

[Каково время жизни статической переменной в C ++-функции?](#)

См. Эту статью, в которой обсуждаются некоторые потоковые последствия для синглетонов:

[Экземпляр Singleton, объявленный как статическая переменная метода GetInstance](#)

См. Эту статью, в которой объясняется, почему двойная проверка блокировки не будет работать на C ++:

[Каковы все общие неопределенные типы поведения, о которых должен знать программист на C ++?](#)

Подклассы

```
class API
```

```

{
public:
    static API& instance();

    virtual ~API() {}

    virtual const char* func1() = 0;
    virtual void func2() = 0;

protected:
    API() {}
    API(const API&) = delete;
    API& operator=(const API&) = delete;
};

class WindowsAPI : public API
{
public:
    virtual const char* func1() override { /* Windows code */ }
    virtual void func2() override { /* Windows code */ }
};

class LinuxAPI : public API
{
public:
    virtual const char* func1() override { /* Linux code */ }
    virtual void func2() override { /* Linux code */ }
};

API& API::instance() {
#ifdef PLATFORM == WIN32
    static WindowsAPI instance;
#elif PLATFORM == LINUX
    static LinuxAPI instance;
#endif
    return instance;
}

```

В этом примере простой коммутатор компилятора привязывает класс `API` к соответствующему подклассу. Таким образом, `API` может быть доступен без привязки к коду конкретной платформы.

Нитевидный Singeton

C ++ 11

Стандарты C ++ 11 гарантируют, что инициализация объектов области функций инициализируется синхронно. Это можно использовать для реализации потокобезопасного синглета с [ленивой инициализацией](#).

```

class Foo
{
public:
    static Foo& instance()
    {
        static Foo inst;

```

```

        return inst;
    }
private:
    Foo() {}
    Foo(const Foo&) = delete;
    Foo& operator =(const Foo&) = delete;
};

```

Статический деинициализационно-безопасный синглтон.

Бывают случаи с несколькими статическими объектами, где вы должны быть в состоянии гарантировать, что *синглтон* не будет уничтожен, пока все статические объекты, которые используют *синглтон*, больше не нуждаются в нем.

В этом случае `std::shared_ptr` может использоваться, чтобы поддерживать *singleton* для всех пользователей, даже когда статические деструкторы вызываются в конце программы:

```

class Singleton
{
public:
    Singleton(Singleton const&) = delete;
    Singleton& operator=(Singleton const&) = delete;

    static std::shared_ptr<Singleton> instance()
    {
        static std::shared_ptr<Singleton> s{new Singleton};
        return s;
    }

private:
    Singleton() {}
};

```

ПРИМЕЧАНИЕ. Этот пример появляется в ответе в разделе вопросов и ответов.

Прочитайте Шаблон дизайна Singleton онлайн: <https://riptutorial.com/ru/cplusplus/topic/2713/шаблон-дизайна-singleton>

глава 144: Шаблоны

Вступление

Классы, функции и (поскольку C ++ 14) могут быть шаблонами. Шаблон представляет собой фрагмент кода с некоторыми свободными параметрами, которые станут конкретным классом, функцией или переменной, когда будут указаны все параметры. Параметры могут быть типами, значениями или самими шаблонами. Известным шаблоном является `std::vector`, который становится конкретным типом контейнера, когда указан тип элемента, например `std::vector<int>`.

Синтаксис

- шаблон < шаблон-параметр-список > декларация
- `export template < template-parameter-list > объявление` / * до C ++ 11 * /
- шаблон <> Объявление
- объявление шаблона
- объявление шаблона `extern` / *, поскольку C ++ 11 * /
- `template < template-parameter-list > class ... ( opt ) identifier ( opt )`
- `template < template-parameter-list > идентификатор класса ( opt ) = id-expression`
- `template < template-parameter-list > typename ... ( opt ) identifier ( opt )` / *, поскольку C ++ 17 * /
- `template < template-parameter-list > typename identifier ( opt ) = id-expression` / *, поскольку C ++ 17 * /
- `postfix-expression .` Идентификатор шаблона
- `postfix-expression ->` идентификатор шаблона
- `template` вложенного имени-спецификатора `simple-template-id` ::

замечания

`template` слова - это **ключевое слово** с пятью различными значениями на языке C ++, в зависимости от контекста.

1. Когда за ним следует список параметров шаблона, заключенных в <>, он объявляет шаблон, такой как **шаблон класса**, **шаблон функции** или **частичную специализацию** существующего шаблона.

```
template <class T>
void increment(T& x) { ++x; }
```

2. Когда следует **пустой** <>, он объявляет **явную (полную) специализацию**.

```
template <class T>
void print(T x);

template <> // <-- keyword used in this sense here
void print(const char* s) {
    // output the content of the string
    printf("%s\n", s);
}
```

3. При последующей декларации без <> , она образует **явное инстанцирование** декларацию или определение.

```
template <class T>
std::set<T> make_singleton(T x) { return std::set<T>(x); }

template std::set<int> make_singleton(int x); // <-- keyword used in this sense here
```

4. В списке параметров шаблона он вводит **параметр шаблона шаблона** .

```
template <class T, template <class U> class Alloc>
//          ^^^^^^^ keyword used in this sense here
class List {
    struct Node {
        T value;
        Node* next;
    };
    Alloc<Node> allocator;
    Node* allocate_node() {
        return allocator.allocate(sizeof(T));
    }
    // ...
};
```

5. После оператора разрешения области видимости :: и операторов доступа к классу . и -> , он указывает, что следующее имя является шаблоном.

```
struct Allocator {
    template <class T>
    T* allocate();
};

template <class T, class Alloc>
class List {
    struct Node {
        T value;
        Node* next;
    }
    Alloc allocator;
    Node* allocate_node() {
        // return allocator.allocate<Node>(); // error: < and > are interpreted as
                                                // comparison operators
        return allocator.template allocate<Node>(); // ok; allocate is a template
        //          ^^^^^^^ keyword used in this sense here
    }
};
```


Перед C ++ 11 шаблон можно было объявить с **ключевым словом** `export` , превратив его в *экспортируемый* шаблон. Определение экспортируемого шаблона не обязательно должно присутствовать в каждой единицы перевода, в которой создается экземпляр шаблона. Например, должно было работать следующее:

foo.h :

```
#ifndef FOO_H
#define FOO_H
export template <class T> T identity(T x);
#endif
```

foo.cpp :

```
#include "foo.h"
template <class T> T identity(T x) { return x; }
```

main.cpp :

```
#include "foo.h"
int main() {
    const int x = identity(42); // x is 42
}
```

Из-за сложности выполнения ключевое слово `export` не поддерживалось большинством основных компиляторов. Он был удален в C ++ 11; В настоящее время запрещено использовать ключевое слово `export` вообще. Вместо этого обычно необходимо определять шаблоны в заголовках (в отличие от функций без шаблонов, которые обычно *не* определены в заголовках). См. [Почему шаблоны могут быть реализованы только в файле заголовка?](#)

Examples

Шаблоны функций

Шаблон может также применяться к функциям (а также к более традиционным структурам) с тем же эффектом.

```
// 'T' stands for the unknown type
// Both of our arguments will be of the same type.
template<typename T>
void printSum(T add1, T add2)
{
    std::cout << (add1 + add2) << std::endl;
}
```

Затем это можно использовать так же, как шаблоны структуры.

```
printSum<int>(4, 5);
printSum<float>(4.5f, 8.9f);
```

В обоих случаях аргумент шаблона используется для замены типов параметров; результат работает так же, как и обычная C++-функция (если параметры не соответствуют типу шаблона, то компилятор применяет стандартные преобразования).

Еще одно свойство шаблонных функций (в отличие от классов шаблонов) заключается в том, что компилятор может вывести параметры шаблона на основе параметров, переданных функции.

```
printSum(4, 5);           // Both parameters are int.
                           // This allows the compiler deduce that the type
                           // T is also int.

printSum(5.0, 4);         // In this case the parameters are two different types.
                           // The compiler is unable to deduce the type of T
                           // because there are contradictions. As a result
                           // this is a compile time error.
```

Эта функция позволяет нам упростить код, когда мы объединяем структуры и функции шаблонов. В стандартной библиотеке существует общий шаблон, который позволяет нам создавать `template structure X` с помощью вспомогательной функции `make_X()`.

```
// The make_X pattern looks like this.
// 1) A template structure with 1 or more template types.
template<typename T1, typename T2>
struct MyPair
{
    T1    first;
    T2    second;
};
// 2) A make function that has a parameter type for
//     each template parameter in the template structure.
template<typename T1, typename T2>
MyPair<T1, T2> make_MyPair(T1 t1, T2 t2)
{
    return MyPair<T1, T2>{t1, t2};
}
```

Как это помогает?

```
auto val1 = MyPair<int, float>{5, 8.7};           // Create object explicitly defining the types
auto val2 = make_MyPair(5, 8.7);                  // Create object using the types of the parameters.
                                                    // In this code both val1 and val2 are the same
                                                    // type.
```

Примечание. Это не предназначено для сокращения кода. Это сделано для того, чтобы сделать код более надежным. Он позволяет изменять типы, изменяя код в одном месте, а не в нескольких местах.

Переадресация аргументов

Шаблон может принимать как значения lvalue, так и rvalue с использованием *ссылки пересылки* :

```
template <typename T>
void f(T &&t);
```

В этом случае реальный тип `t` будет выведен в зависимости от контекста:

```
struct X { };

X x;
f(x); // calls f<X&>(x)
f(X()); // calls f<X>(x)
```

В первом случае тип `T` выводится как *ссылка на X* (`X&`), а тип `t` является *ссылкой на x*, а во втором случае тип `T` выводится как `X` и тип `t` качестве *ссылки на rvalue* к `X` (`X&&`).

Примечание. Стоит заметить, что в первом случае `decltype(t)` совпадает с `T`, но не во втором.

Чтобы отлично перевести `t` в другую функцию, будь то ссылка lvalue или rvalue, нужно использовать `std::forward` :

```
template <typename T>
void f(T &&t) {
    g(std::forward<T>(t));
}
```

Ссылки на пересылку могут использоваться с вариационными шаблонами:

```
template <typename... Args>
void f(Args&&... args) {
    g(std::forward<Args>(args)...);
}
```

Примечание. Ссылки на перенаправление могут использоваться только для параметров шаблона, например, в следующем коде, `v` является ссылкой на rvalue, а не ссылкой на пересылку:

```
#include <vector>

template <typename T>
void f(std::vector<T> &&v);
```

Шаблон базового класса

Основная идея шаблона класса заключается в том, что параметр шаблона заменяется типом во время компиляции. В результате один и тот же класс может быть повторно использован для нескольких типов. Пользователь указывает, какой тип будет использоваться при объявлении переменной класса. Три примера этого показаны в `main()` :

```
#include <iostream>
using std::cout;

template <typename T>           // A simple class to hold one number of any type
class Number {
public:
    void setNum(T n);           // Sets the class field to the given number
    T plus1() const;           // returns class field's "follower"
private:
    T num;                     // Class field
};

template <typename T>           // Set the class field to the given number
void Number<T>::setNum(T n) {
    num = n;
}

template <typename T>           // returns class field's "follower"
T Number<T>::plus1() const {
    return num + 1;
}

int main() {
    Number<int> anInt;          // Test with an integer (int replaces T in the class)
    anInt.setNum(1);
    cout << "My integer + 1 is " << anInt.plus1() << "\n";        // Prints 2

    Number<double> aDouble;     // Test with a double
    aDouble.setNum(3.1415926535897);
    cout << "My double + 1 is " << aDouble.plus1() << "\n";        // Prints 4.14159

    Number<float> aFloat;       // Test with a float
    aFloat.setNum(1.4);
    cout << "My float + 1 is " << aFloat.plus1() << "\n";          // Prints 2.4

    return 0; // Successful completion
}
```

Шаблонная специализация

Вы можете определить реализацию для конкретных экземпляров класса шаблона / метода.

Например, если у вас есть:

```
template <typename T>
T sqrt(T t) { /* Some generic implementation */ }
```

Затем вы можете написать:

```
template<>
```

```
int sqrt<int>(int i) { /* Highly optimized integer implementation */ }
```

Тогда пользователь, который пишет `sqrt(4.0)`, получит общую реализацию, тогда как `sqrt(4)` получит специализированную реализацию.

Специализация частичного шаблона

В отличие от полной специализированной специализации шаблона, можно ввести шаблон с некоторыми аргументами существующего шаблона. Специализация частичного шаблона доступна только для шаблона `class` / `structs`:

```
// Common case:
template<typename T, typename U>
struct S {
    T t_val;
    U u_val;
};

// Special case when the first template argument is fixed to int
template<typename V>
struct S<int, V> {
    double another_value;
    int foo(double arg) { // Do something }
};
```

Как показано выше, частичные специализированные шаблоны могут вводить совершенно разные наборы данных и членов функций.

Когда создается экземпляр частично специализированного шаблона, выбирается наиболее подходящая специализация. Например, давайте определим шаблон и две частичные специализации:

```
template<typename T, typename U, typename V>
struct S {
    static void foo() {
        std::cout << "General case\n";
    }
};

template<typename U, typename V>
struct S<int, U, V> {
    static void foo() {
        std::cout << "T = int\n";
    }
};

template<typename V>
struct S<int, double, V> {
    static void foo() {
        std::cout << "T = int, U = double\n";
    }
};
```

Теперь следующие вызовы:

```
S<std::string, int, double>::foo();
S<int, float, std::string>::foo();
S<int, double, std::string>::foo();
```

распечатает

```
General case
T = int
T = int, U = double
```

Шаблоны функций могут быть полностью специализированными:

```
template<typename T, typename U>
void foo(T t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}

// OK.
template<>
void foo<int, int>(int a1, int a2) {
    std::cout << "Two ints: " << a1 << " " << a2 << std::endl;
}

void invoke_foo() {
    foo(1, 2.1); // Prints "General case: 1 2.1"
    foo(1,2);   // Prints "Two ints: 1 2"
}

// Compilation error: partial function specialization is not allowed.
template<typename U>
void foo<std::string, U>(std::string t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}
```

Значение параметра шаблона по умолчанию

Как и в случае аргументов функции, параметры шаблона могут иметь значения по умолчанию. Все параметры шаблона со значением по умолчанию должны быть объявлены в конце списка параметров шаблона. Основная идея заключается в том, что параметры шаблона со значением по умолчанию могут быть опущены при создании шаблона.

Простой пример использования значения параметра шаблона по умолчанию:

```
template <class T, size_t N = 10>
struct my_array {
    T arr[N];
};

int main() {
    /* Default parameter is ignored, N = 5 */
    my_array<int, 5> a;
```

```

/* Print the length of a.arr: 5 */
std::cout << sizeof(a.arr) / sizeof(int) << std::endl;

/* Last parameter is omitted, N = 10 */
my_array<int> b;

/* Print the length of a.arr: 10 */
std::cout << sizeof(b.arr) / sizeof(int) << std::endl;
}

```

Шаблон псевдонимов

C++ 11

Основной пример:

```
template<typename T> using pointer = T*;
```

Это определение делает `pointer<T>` псевдоним `T*`. Например:

```
pointer<int> p = new int; // equivalent to: int* p = new int;
```

Шаблоны псевдонимов не могут быть специализированными. Однако эту функцию можно получить косвенно, если они ссылаются на вложенный тип в структуре:

```

template<typename T>
struct nonconst_pointer_helper { typedef T* type; };

template<typename T>
struct nonconst_pointer_helper<T const> { typedef T* type; };

template<typename T> using nonconst_pointer = nonconst_pointer_helper<T>::type;

```

Параметры шаблона шаблона

Иногда мы хотели бы перейти в шаблон типа шаблона без фиксации его значений. Для этого создаются параметры шаблона шаблона. Очень простые примеры шаблонов шаблонов:

```

template <class T>
struct Tag1 { };

template <class T>
struct Tag2 { };

template <template <class> class Tag>
struct IntTag {
    typedef Tag<int> type;
};

int main() {

```

```
IntTag<Tag1>::type t;
}
```

C ++ 11

```
#include <vector>
#include <iostream>

template <class T, template <class...> class C, class U>
C<T> cast_all(const C<U> &c) {
    C<T> result(c.begin(), c.end());
    return result;
}

int main() {
    std::vector<float> vf = {1.2, 2.6, 3.7};
    auto vi = cast_all<int>(vf);
    for(auto &&i: vi) {
        std::cout << i << std::endl;
    }
}
```

Объявление аргументов типа non-type с авто

До C ++ 17, когда вы пишете параметр непикового шаблона, вам нужно сначала указать его тип. Таким образом, общая картина стала писать что-то вроде:

```
template <class T, T N>
struct integral_constant {
    using type = T;
    static constexpr T value = N;
};

using five = integral_constant<int, 5>;
```

Но для сложных выражений, используя что-то вроде этого, приходится писать `decltype(expr), expr` при создании шаблонов. Решение состоит в том, чтобы упростить эту идиому и просто разрешить `auto`:

C ++ 17

```
template <auto N>
struct integral_constant {
    using type = decltype(N);
    static constexpr type value = N;
};

using five = integral_constant<5>;
```

Пустой пользовательский отладчик для unique_ptr

Хороший пример мотивации может прийти от попыток объединить пустую оптимизацию базы с пользовательской Deleter для `unique_ptr`. Различные C API-дескрипторы имеют

разные типы возврата, но нам все равно - мы просто хотим, чтобы что-то работало для любой функции:

```
template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) const {
        DeleteFn(ptr);
    }
};

template <T, auto DeleteFn>
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

И теперь вы можете просто использовать любой указатель на функцию, который может принимать аргумент типа `T` в качестве параметра шаблона без типа, независимо от типа возвращаемого значения, и получать из него нестандартные служебные данные `unique_ptr`:

```
unique_ptr_deleter<std::FILE, std::fclose> p;
```

Параметр шаблона непигового типа

Помимо типов в качестве параметра шаблона нам разрешено объявлять значения константных выражений, удовлетворяющие одному из следующих критериев:

- интегрального или перечисляемого типа,
- указатель на объект или указатель на функцию,
- lvalue ссылка на объект или lvalue ссылка на функцию,
- указатель на член,
- `std::nullptr_t`.

Как и все параметры шаблона, параметры шаблона непигового типа могут быть явно заданы, дефолтны или производятся неявно через вычитание аргумента шаблона.

Пример использования параметров шаблона непигового типа:

```
#include <iostream>

template<typename T, std::size_t size>
std::size_t size_of(T (&anArray)[size]) // Pass array by reference. Requires.
{                                         // an exact size. We allow all sizes
    return size;                         // by using a template "size".
}

int main()
{
    char anArrayOfChar[15];
    std::cout << "anArrayOfChar: " << size_of(anArrayOfChar) << "\n";

    int anArrayOfData[] = {1,2,3,4,5,6,7,8,9};
    std::cout << "anArrayOfData: " << size_of(anArrayOfData) << "\n";
}
```

```
}
```

Пример явного указания параметров шаблона типа и не-типа:

```
#include <array>
int main ()
{
    std::array<int, 5> foo; // int is a type parameter, 5 is non-type
}
```

Нестандартные параметры шаблона являются одним из способов достижения повторения шаблона и позволяют выполнять [метапрограммирование](#) .

Структуры данных шаблонов Variadic

C ++ 14

Часто бывает полезно определить классы или структуры, которые имеют переменное число и тип элементов данных, которые определены во время компиляции. Канонический пример - `std::tuple` , но иногда необходимо определить собственные пользовательские структуры. Вот пример, который определяет структуру с использованием рецептуры (а не наследования, как и для `std::tuple` . Начните с общего (пустого) определения, которое также служит базовым случаем для завершения рекурсии в поздней специализации:

```
template<typename ... T>
struct DataStructure {};
```

Это уже позволяет нам определить пустую структуру `DataStructure<> data` , хотя это еще не очень полезно.

Далее идет рекурсивная специализация случая:

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;
};
```

Теперь нам достаточно создавать произвольные структуры данных, такие как `DataStructure<int, float, std::string> data(1, 2.1, "hello")` .

Так, что происходит? Во-первых, обратите внимание, что это специализация, требование которой состоит в том, что существует хотя бы один параметр вариационного шаблона (а

именно `T` выше), но не заботятся о конкретном составе пакета `Rest`. Знание того, что `T` существует, позволяет `first` определить его член данных. Остальные данные рекурсивно упаковываются как `DataStructure<Rest ...> rest`. Конструктор инициализирует оба этих элемента, включая вызов рекурсивного конструктора для `rest` членов.

Чтобы понять это лучше, мы можем работать с примером: предположим, что у вас есть данные `DataStructure<int, float> data`. Объявление сначала совпадает со специализацией, `DataStructure<float> rest` структуру с `DataStructure<float> rest int first` И `DataStructure<float> rest . rest` определение снова совпадает с этой специализацией, создавая `float first` свои собственные `float first` И `DataStructure<> rest`. Наконец, последний `rest` совпадает с деформированием базового блока, создавая пустую структуру.

Вы можете визуализировать это следующим образом:

```
DataStructure<int, float>
-> int first
-> DataStructure<float> rest
    -> float first
    -> DataStructure<> rest
        -> (empty)
```

Теперь у нас есть структура данных, но она не очень полезна, но мы не можем легко получить доступ к отдельным элементам данных (например, чтобы получить доступ к последнему члену данных `DataStructure<int, float, std::string> data` нам пришлось бы использовать `data.rest.rest.first`, что не совсем удобно). Таким образом, мы добавляем к нему метод `get` (нужен только в специализации, так как базовая структура не имеет данных для `get`):

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    ...
    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T, Rest...>>::get(*this);
    }
    ...
};
```

Как вы можете видеть это `get` функцию члена сама шаблонным - на этот раз по индексу элемента, который необходим (и поэтому использование может быть что - то вроде `data.get<1>()`, аналогичен `std::tuple`). Фактическая работа выполняется статической функцией в вспомогательном классе `GetHelper`. Причина, почему мы не можем определить необходимые функции непосредственно в `DataStructure` «S `get` потому, что (как мы скоро увидим), мы должны были бы специализироваться на `idx` - но это не возможно специализироваться функциями - члена шаблона без Специализируя содержащий класс шаблон. Обратите также внимание на то, что использование C++14-стиля `auto` делает

нашу жизнь значительно проще, поскольку в противном случае нам понадобится довольно сложное выражение для типа возврата.

Итак, на вспомогательный класс. На этот раз нам понадобится пустая декларация и две специализации. Сначала заявление:

```
template<size_t idx, typename T>
struct GetHelper;
```

Теперь базовый регистр (когда `idx==0`). В этом случае мы просто возвращаем `first` члена:

```
template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};
```

В рекурсивном случае мы уменьшаем `idx` и вызываем `GetHelper` для `rest` членов:

```
template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ... >>::get(data.rest);
    }
};
```

Для работы с примером предположим, что у нас есть данные `DataStructure<int, float> data` и нам нужен `data.get<1>()` . Это вызывает `GetHelper<1, DataStructure<int, float>>::get(data)` (вторая специализация), которая, в свою очередь, вызывает `GetHelper<0, DataStructure<float>>::get(data.rest)` , который, наконец, возвращает (по 1-й специализации, так как теперь `idx` равен 0) `data.rest.first` .

Итак, это все! Вот весь код функционирования, с некоторым примером используемого в `main` функции:

```
#include <iostream>

template<size_t idx, typename T>
struct GetHelper;

template<typename ... T>
struct DataStructure
{
};

template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
```

```

{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;

    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T, Rest...>>::get(*this);
    }
};

template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};

template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ... >>::get(data.rest);
    }
};

int main()
{
    DataStructure<int, float, std::string> data(1, 2.1, "Hello");

    std::cout << data.get<0>() << std::endl;
    std::cout << data.get<1>() << std::endl;
    std::cout << data.get<2>() << std::endl;

    return 0;
}

```

Явная реализация

Явное определение экземпляра создает и объявляет конкретный класс, функцию или переменную из шаблона, но пока не использует его. На явное создание можно ссылаться на другие единицы перевода. Это можно использовать, чтобы избежать определения шаблона в файле заголовка, если он будет создан только с помощью конечного набора аргументов. Например:

```

// print_string.h
template <class T>
void print_string(const T* str);

```

```
// print_string.cpp
#include "print_string.h"
template void print_string(const char*);
template void print_string(const wchar_t*);
```

Поскольку `print_string<char>` и `print_string<wchar_t>` явно создаются в `print_string.cpp`, компоновщик сможет их найти, даже если шаблон `print_string` не определен в заголовке. Если эти объявления с явным манифестом не присутствуют, вероятно, возникнет ошибка компоновщика. См. [Почему шаблоны могут быть реализованы только в файле заголовка?](#)

C++ 11

Если явному описанию инстанцирования предшествует **ключевое слово** `extern`, оно вместо этого становится явным *заявлением о создании*. Наличие явной декларации о создании для данной специализации предотвращает неявное создание данной специализации в пределах текущей единицы перевода. Вместо этого ссылка на эту специализацию, которая в противном случае вызвала бы неявное создание, может ссылаться на явное определение инстанцирования в том же или другом TU.

foo.h

```
#ifndef FOO_H
#define FOO_H
template <class T> void foo(T x) {
    // complicated implementation
}
#endif
```

foo.cpp

```
#include "foo.h"
// explicit instantiation definitions for common cases
template void foo(int);
template void foo(double);
```

main.cpp

```
#include "foo.h"
// we already know foo.cpp has explicit instantiation definitions for these
extern template void foo(double);
int main() {
    foo(42);    // instantiates foo<int> here;
               // wasteful since foo.cpp provides an explicit instantiation already!
    foo(3.14); // does not instantiate foo<double> here;
               // uses instantiation of foo<double> in foo.cpp instead
}
```

Прочитайте Шаблоны онлайн: <https://riptutorial.com/ru/cplusplus/topic/460/шаблоны>

глава 145: Шаблоны выписки

Examples

Базовые шаблоны выражений на элементарных алгебраических выражениях

Введение и мотивация

Шаблоны экспрессии (обозначаемые как **ЕТ** в следующем) - это мощная технология метапрограммирования шаблонов, используемая для ускорения вычислений иногда довольно дорогостоящих выражений. Он широко используется в разных областях, например, в реализации библиотек линейной алгебры.

В этом примере рассмотрим контекст линейных алгебраических вычислений. Более конкретно, вычисления, включающие только **элементарные операции**. Такие вычисления являются наиболее распространенными приложениями **ЕТ**, и они служат хорошим представлением о том, как **ЕТ** работают внутри страны.

Давайте посмотрим на мотивирующий пример. Рассмотрим вычисление выражения:

```
Vector vec_1, vec_2, vec_3;

// Initializing vec_1, vec_2 and vec_3.

Vector result = vec_1 + vec_2*vec_3;
```

Здесь для простоты я предполагаю, что класс `Vector` и операция `+` (вектор плюс: элемент-плюс операция) и операция `*` (здесь означает векторное скалярное произведение: также элементная операция), как правильно реализованы, так и как они должны быть, математически.

В обычной реализации без использования **ЕТ** (или других аналогичных методов) для получения конечного `result` имеет место **не менее пяти** конструкций векторов `Vector`:

1. Три экземпляра, соответствующие `vec_1`, `vec_2` и `vec_3`.
2. Временной `Vector` экземпляр `_tmp`, представляющий результат `_tmp = vec_2*vec_3;`,
3. Наконец, при правильном использовании **оптимизации возвращаемого значения** построение конечного `result` в `result = vec_1 + _tmp;`,

Реализация с использованием **ЕТ** может **исключить создание временного** `Vector _tmp` в 2, оставив при этом только **четыре** конструкции экземпляров `Vector`. Более интересно, рассмотрим следующее более сложное выражение:

```
Vector result = vec_1 + (vec_2*vec3 + vec_1)*(vec_2 + vec_3*vec_1);
```

Также будет четыре конструкции экземпляров `Vector`: `vec_1`, `vec_2`, `vec_3` и `result`. Другими словами, в этом примере, где задействуются только элементарные операции, гарантируется, что из промежуточных вычислений не создаются временные объекты.

Как работают ET

В принципе, **ЕТ** для любых алгебраических вычислений состоят из двух строительных блоков:

1. **Чистые алгебраические выражения (РАЕ)**: они являются прокси-абстракциями алгебраических выражений. Чистая алгебраика не выполняет фактических вычислений, это просто абстракции / моделирование рабочего потока вычислений. РАЕ может быть моделью **либо ввода, либо вывода любых алгебраических выражений**. Случаи РАЕ часто считаются дешевыми для копирования.
2. **Ленивые оценки**: реализация реальных вычислений. В следующем примере мы увидим, что для выражений, содержащих только элементарные операции, ленивые оценки могут реализовывать фактические вычисления внутри операции индексированного доступа в конечном результате, создавая таким образом схему оценки по требованию: вычисление не выполняется только в том случае, если конечный результат был получен / запрошен.

Итак, конкретно, как мы внедряем **ЕТ** в этом примере? Пойдем теперь через него.

Рассмотрим всегда следующий фрагмент кода:

```
Vector vec_1, vec_2, vec_3;  
  
// Initializing vec_1, vec_2 and vec_3.  
  
Vector result = vec_1 + vec_2*vec_3;
```

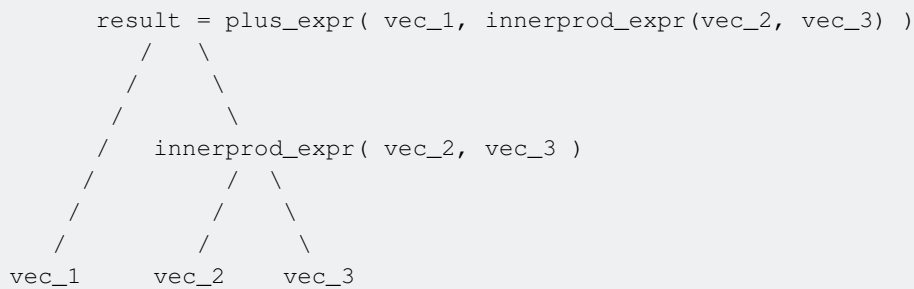
Выражение для вычисления результата может быть далее разложено на два подвыражения:

1. Векторное выражение (обозначенное как **plus_expr**)
2. Векторное выражение внутреннего произведения (обозначается как **innerprod_expr**).

Что делают **ГЭТ**:

- Вместо того, чтобы сразу вычислить каждое **подвыражение**, **ЕТs** сначала моделируют все выражение, используя графическую структуру. Каждый узел в графе представляет собой **РАЕ**. Конечное соединение узлов представляет собой

фактический поток вычислений. Итак, для приведенного выше выражения мы получаем следующий график:



- Окончательное вычисление реализуется путем **просмотра иерархии графов** : поскольку здесь мы имеем дело **только с элементарными** операциями, вычисление каждого проиндексированного значения в `result` **может быть выполнено независимо** : окончательная оценка `result` может быть лениво перенесена на **элемент- мудрая оценка** каждого элемента `result` . Другими словами, поскольку вычисление элемента `result elem_res` может быть выражено с использованием соответствующих элементов в `vec_1 ( elem_1 )`, `vec_2 ( elem_2 )` и `vec_3 ( elem_3 )` как:

```
elem_res = elem_1 + elem_2*elem_3;
```

поэтому нет необходимости создавать временный `Vector` для хранения результата промежуточного внутреннего продукта: **весь расчет для одного элемента может быть выполнен в целом и закодирован внутри операции с индексированным доступом** .

Вот пример кода в действии.

Файл `vec.hh`: wrapper для `std :: vector`, используемый для отображения журнала при вызове конструкции.

```
#ifndef EXPR_VEC
# define EXPR_VEC

# include <vector>
# include <cassert>
# include <utility>
# include <iostream>
# include <algorithm>
# include <functional>

///
/// This is a wrapper for std::vector. It's only purpose is to print out a log when a
/// vector constructions in called.
/// It wraps the indexed access operator [] and the size() method, which are
/// important for later ETs implementation.
```

```

///

// std::vector wrapper.
template<typename ScalarType> class Vector
{
public:
    explicit Vector() { std::cout << "ctor called.\n"; };
    explicit Vector(int size): _vec(size) { std::cout << "ctor called.\n"; };
    explicit Vector(const std::vector<ScalarType> &vec): _vec(vec)
    { std::cout << "ctor called.\n"; };

    Vector(const Vector<ScalarType> & vec): _vec{vec{}}
    { std::cout << "copy ctor called.\n"; };
    Vector(Vector<ScalarType> && vec): _vec(std::move(vec{}))
    { std::cout << "move ctor called.\n"; };

    Vector<ScalarType> & operator=(const Vector<ScalarType> &) = default;
    Vector<ScalarType> & operator=(Vector<ScalarType> &&) = default;

    decltype(auto) operator[](int indx) { return _vec[indx]; }
    decltype(auto) operator[](int indx) const { return _vec[indx]; }

    decltype(auto) operator()() & { return (_vec); };
    decltype(auto) operator()() const & { return (_vec); };
    Vector<ScalarType> && operator()() && { return std::move(*this); }

    int size() const { return _vec.size(); }

private:
    std::vector<ScalarType> _vec;
};

///
/// These are conventional overloads of operator + (the vector plus operation)
/// and operator * (the vector inner product operation) without using the expression
/// templates. They are later used for bench-marking purpose.
///

// + (vector plus) operator.
template<typename ScalarType>
auto operator+(const Vector<ScalarType> &lhs, const Vector<ScalarType> &rhs)
{
    assert(lhs().size() == rhs().size() &&
           "error: ops plus -> lhs and rhs size mismatch.");

    std::vector<ScalarType> _vec;
    _vec.resize(lhs().size());
    std::transform(std::cbegin(lhs()), std::cend(lhs()),
                   std::cbegin(rhs()), std::begin(_vec),
                   std::plus<>());
    return Vector<ScalarType>(std::move(_vec));
}

// * (vector inner product) operator.
template<typename ScalarType>
auto operator*(const Vector<ScalarType> &lhs, const Vector<ScalarType> &rhs)
{
    assert(lhs().size() == rhs().size() &&
           "error: ops multiplies -> lhs and rhs size mismatch.");

    std::vector<ScalarType> _vec;

```

```

_vec.resize(lhs().size());
std::transform(std::cbegin(lhs()), std::cend(lhs()),
               std::cbegin(rhs()), std::begin(_vec),
               std::multiplies<>());
return Vector<ScalarType>(std::move(_vec));
}

#endif //!EXPR_VEC

```

Файл expr.hh: реализация шаблонов выражений для элементарных операций (векторный плюс и векторный скалярный продукт)

Давайте разложим его на разделы.

1. Раздел 1 реализует базовый класс для всех выражений. В нем используется [Curiously Recurring Template Pattern \(CRTP \)](#).
2. Раздел 2 реализует первый **РАЕ** : **терминал** , который является просто оболочкой (константной ссылкой) структуры входных данных, содержащей реальное входное значение для вычисления.
3. Раздел 3 реализует второй **РАЕ** : **binary_operation** , который является шаблоном класса, который впоследствии используется для vector_plus и vector_innerprod. Он параметризуется **типом операции** , **левым РАЕ** и **правым РАЕ** . Фактическое вычисление кодируется в операторе индексированного доступа.
4. Раздел 4 определяет операции vector_plus и vector_innerprod как **элементную операцию** . Он также перегружает оператор + и * для **РАЕ** s: так что эти две операции также возвращают **РАЕ** .

```

#ifndef EXPR_EXPR
# define EXPR_EXPR

// Fwd declaration.
template<typename> class Vector;

namespace expr
{

// -----
//
// Section 1.
//
// The first section is a base class template for all kinds of expression. It
// employs the Curiously Recurring Template Pattern, which enables its instantiation
// to any kind of expression structure inheriting from it.
//
// -----

```

```

/// Base class for all expressions.
template<typename Expr> class expr_base
{
public:
    const Expr& self() const { return static_cast<const Expr&>(*this); }
    Expr& self() { return static_cast<Expr&>(*this); }

protected:
    explicit expr_base() {};
    int size() const { return self().size_impl(); }
    auto operator[](int indx) const { return self().at_impl(indx); }
    auto operator()() const { return self()(); };
};

/// -----
///
/// The following section 2 & 3 are abstractions of pure algebraic expressions (PAE).
/// Any PAE can be converted to a real object instance using operator(): it is in
/// this conversion process, where the real computations are done.

///
/// Section 2. Terminal
///
/// A terminal is an abstraction wrapping a const reference to the Vector data
/// structure. It inherits from expr_base, therefore providing a unified interface
/// wrapping a Vector into a PAE.
///
/// It provides the size() method, indexed access through at_impl() and a conversion
/// to referenced object through () operator.
///
/// It might no be necessary for user defined data structures to have a terminal
/// wrapper, since user defined structure can inherit expr_base, therefore eliminates
/// the need to provide such terminal wrapper.
///
/// -----

/// Generic wrapper for underlying data structure.
template<typename DataType> class terminal: expr_base<terminal<DataType>>
{
public:
    using base_type = expr_base<terminal<DataType>>;
    using base_type::size;
    using base_type::operator[];
    friend base_type;

    explicit terminal(const DataType &val): _val(val) {}
    int size_impl() const { return _val.size(); };
    auto at_impl(int indx) const { return _val[indx]; };
    decltype(auto) operator()() const { return (_val); }

private:
    const DataType &_val;
};

/// -----
///
/// Section 3. Binary operation expression.

```

```

///
/// This is a PAE abstraction of any binary expression. Similarly it inherits from
/// expr_base.
///
/// It provides the size() method, indexed access through at_impl() and a conversion
/// to referenced object through () operator. Each call to the at_impl() method is
/// a element wise computation.
///
/// -----

/// Generic wrapper for binary operations (that are element-wise).
template<typename Ops, typename lExpr, typename rExpr>
class binary_ops: public expr_base<binary_ops<Ops,lExpr,rExpr>>
{
public:
    using base_type = expr_base<binary_ops<Ops,lExpr,rExpr>>;
    using base_type::size;
    using base_type::operator[];
    friend base_type;

    explicit binary_ops(const Ops &ops, const lExpr &lxpr, const rExpr &rxpr)
        : _ops(ops), _lxpr(lxpr), _rxpr(rxpr) {};
    int size_impl() const { return _lxpr.size(); };

    /// This does the element-wise computation for index indx.
    auto at_impl(int indx) const { return _ops(_lxpr[indx], _rxpr[indx]); };

    /// Conversion from arbitrary expr to concrete data type. It evaluates
    /// element-wise computations for all indices.
    template<typename DataType> operator DataType()
    {
        DataType _vec(size());
        for(int _ind = 0; _ind < _vec.size(); ++_ind)
            _vec[_ind] = (*this)[_ind];
        return _vec;
    }

private: /// Ops and expr are assumed cheap to copy.
    Ops _ops;
    lExpr _lxpr;
    rExpr _rxpr;
};

/// -----
/// Section 4.
///
/// The following two structs defines algebraic operations on PAEs: here only vector
/// plus and vector inner product are implemented.
///
/// First, some element-wise operations are defined : in other words, vec_plus and
/// vec_prod acts on elements in Vectors, but not whole Vectors.
///
/// Then, operator + & * are overloaded on PAEs, such that: + & * operations on PAEs
/// also return PAEs.
///
/// -----

/// Element-wise plus operation.

```

```

struct vec_plus_t
{
    constexpr explicit vec_plus_t() = default;
    template<typename LType, typename RType>
    auto operator()(const LType &lhs, const RType &rhs) const
    { return lhs+rhs; }
};

/// Element-wise inner product operation.
struct vec_prod_t
{
    constexpr explicit vec_prod_t() = default;
    template<typename LType, typename RType>
    auto operator()(const LType &lhs, const RType &rhs) const
    { return lhs*rhs; }
};

/// Constant plus and inner product operator objects.
constexpr vec_plus_t vec_plus{};
constexpr vec_prod_t vec_prod{};

/// Plus operator overload on expressions: return binary expression.
template<typename lExpr, typename rExpr>
auto operator+(const lExpr &lhs, const rExpr &rhs)
{ return binary_ops<vec_plus_t, lExpr, rExpr>(vec_plus, lhs, rhs); }

/// Inner prod operator overload on expressions: return binary expression.
template<typename lExpr, typename rExpr>
auto operator*(const lExpr &lhs, const rExpr &rhs)
{ return binary_ops<vec_prod_t, lExpr, rExpr>(vec_prod, lhs, rhs); }

} //!expr

#endif //!EXPR_EXPR

```

Файл main.cc: файл test src

```

# include <chrono>
# include <iomanip>
# include <iostream>
# include "vec.hh"
# include "expr.hh"
# include "boost/core/demangle.hpp"

int main()
{
    using dtype = float;
    constexpr int size = 5e7;

    std::vector<dtype> _vec1(size);
    std::vector<dtype> _vec2(size);
    std::vector<dtype> _vec3(size);

    // ... Initialize vectors' contents.

```

```

Vector<dtype> vec1(std::move(_vec1));
Vector<dtype> vec2(std::move(_vec2));
Vector<dtype> vec3(std::move(_vec3));

unsigned long start_ms_no_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
        (std::chrono::system_clock::now().time_since_epoch()).count();
std::cout << "\nNo-ETs evaluation starts.\n";

Vector<dtype> result_no_ets = vec1 + (vec2*vec3);

unsigned long stop_ms_no_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
        (std::chrono::system_clock::now().time_since_epoch()).count();
std::cout << std::setprecision(6) << std::fixed
    << "No-ETs. Time elapses: " << (stop_ms_no_ets-start_ms_no_ets)/1000.0
    << " s.\n" << std::endl;

unsigned long start_ms_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
        (std::chrono::system_clock::now().time_since_epoch()).count();
std::cout << "Evaluation using ETs starts.\n";

expr::terminal<Vector<dtype>> vec4(vec1);
expr::terminal<Vector<dtype>> vec5(vec2);
expr::terminal<Vector<dtype>> vec6(vec3);

Vector<dtype> result_ets = (vec4 + vec5*vec6);

unsigned long stop_ms_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
        (std::chrono::system_clock::now().time_since_epoch()).count();
std::cout << std::setprecision(6) << std::fixed
    << "With ETs. Time elapses: " << (stop_ms_ets-start_ms_ets)/1000.0
    << " s.\n" << std::endl;

auto ets_ret_type = (vec4 + vec5*vec6);
std::cout << "\nETs result's type:\n";
std::cout << boost::core::demangle( typeid(decltype(ets_ret_type)).name() ) << '\n';

return 0;
}

```

Вот один из возможных результатов при компиляции с `-O3 -std=c++14` с использованием GCC 5.3:

```

ctor called.
ctor called.
ctor called.

No-ETs evaluation starts.
ctor called.
ctor called.
No-ETs. Time elapses: 0.571000 s.

Evaluation using ETs starts.
ctor called.
With ETs. Time elapses: 0.164000 s.

```

```
ETs result's type:
expr::binary_ops<expr::vec_plus_t, expr::terminal<Vector<float> >,
expr::binary_ops<expr::vec_prod_t, expr::terminal<Vector<float> >,
expr::terminal<Vector<float> > > >
```

Наблюдения:

- Использование **ETs** в этом случае приводит к значительному повышению производительности (> 3x).
- Создание временного векторного объекта устраняется. Как и в случае **ГЭ** `тер` вызывается только один раз.
- `Boost :: demangle` использовался для визуализации типа возврата ET перед преобразованием: он четко построил точно такой же график выражений, что и выше.

Оттяжки и оговорки

- Очевидным недостатком **ГЭ** является кривая обучения, сложность реализации и сложность обслуживания кода. В приведенном выше примере, где рассматриваются только элементарные операции, реализация содержит уже огромное количество шаблонов, не говоря уже о реальном мире, где более сложные алгебраические выражения встречаются в каждом вычислении, а элементная независимость больше не выполняется (например, матричное умножение), трудность будет экспоненциальной.
- Еще одно предостережение от использования **ЕТ** - это то, что они хорошо `auto` ключевым словом `auto`. Как упоминалось выше, **PAEs** по существу являются прокси-серверами: и прокси-серверы в основном не хорошо работают с `auto`. Рассмотрим следующий пример:

```
auto result = ...;                                // Some expensive expression:
                                                    // auto returns the expr graph,
                                                    // NOT the computed value.

for(auto i = 0; i < 100; ++i)
    ScalarType value = result* ... // Some other expensive computations using result.
```

Здесь **в каждой итерации цикла for результат будет переоценен**, так как граф выражения вместо вычисленного значения передается в цикл `for`.

Существующие библиотеки, внедряющие ET

- **boost :: proto** - это мощная библиотека, позволяющая вам определять свои собственные правила и грамматики для собственных выражений и выполнять их с помощью **ЕТ**.
- **Eigen** - библиотека для линейной алгебры, которая эффективно использует

различные алгебраические вычисления с использованием **ЕТ** .

Основной пример, иллюстрирующий шаблоны выражений

Шаблон выражения - это метод оптимизации времени компиляции, используемый в основном в научных вычислениях. Основная цель - избегать ненужных временных рядов и оптимизировать вычисления циклов с использованием одного прохода (как правило, при выполнении операций с числовыми агрегатами). Шаблоны экспрессии были первоначально разработаны для того, чтобы обойти неэффективность наивной перегрузки оператора при реализации численных типов `Array` или `Matrix` . Эквивалентная терминология для шаблонов выражений была введена Бьярне Страуступом, который называет их «сплавленными операциями» в последней версии своей книги «Язык программирования C ++».

Прежде чем погрузиться в шаблоны выражений, вы должны понять, почему они вам нужны в первую очередь. Чтобы проиллюстрировать это, рассмотрим очень простой класс `Matrix`, приведенный ниже:

```
template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
public:
    using value_type = T;

    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

private:
    std::vector<T> values;
};

template <typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW>
operator+(const Matrix<T, COL, ROW>& lhs, const Matrix<T, COL, ROW>& rhs)
{
    Matrix<T, COL, ROW> result;

    for (size_t y = 0; y != lhs.rows(); ++y) {
        for (size_t x = 0; x != lhs.cols(); ++x) {
            result(x, y) = lhs(x, y) + rhs(x, y);
        }
    }
    return result;
}
```

Учитывая предыдущее определение класса, вы можете теперь написать выражения `Matrix`, такие как:

```
const std::size_t cols = 2000;
const std::size_t rows = 1000;
```

```

Matrix<double, cols, rows> a, b, c;

// initialize a, b & c
for (std::size_t y = 0; y != rows; ++y) {
    for (std::size_t x = 0; x != cols; ++x) {
        a(x, y) = 1.0;
        b(x, y) = 2.0;
        c(x, y) = 3.0;
    }
}

Matrix<double, cols, rows> d = a + b + c; // d(x, y) = 6

```

Как показано выше, возможность перегрузить `operator+()` дает вам обозначение, которое имитирует естественную математическую нотацию для матриц.

К сожалению, предыдущая реализация также очень неэффективна по сравнению с эквивалентной версией «ручной работы».

Чтобы понять, почему, вы должны учитывать, что происходит, когда вы пишете выражение, такое как `Matrix d = a + b + c`. Это фактически расширяется до `((a + b) + c)` или `operator+(operator+(a, b), c)`. Другими словами, цикл внутри `operator+()` выполняется дважды, тогда как его можно было бы легко выполнить за один проход. Это также приводит к созданию 2 временных переменных, что еще больше ухудшает производительность. По сути, добавив гибкость в использование нотации, близкой к ее математическому аналогу, вы также сделали класс `Matrix` крайне неэффективным.

Например, без перегрузки оператора вы могли бы реализовать гораздо более эффективное суммирование `Matrix` с использованием одного прохода:

```

template<typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW> add3(const Matrix<T, COL, ROW>& a,
                        const Matrix<T, COL, ROW>& b,
                        const Matrix<T, COL, ROW>& c)
{
    Matrix<T, COL, ROW> result;
    for (size_t y = 0; y != ROW; ++y) {
        for (size_t x = 0; x != COL; ++x) {
            result(x, y) = a(x, y) + b(x, y) + c(x, y);
        }
    }
    return result;
}

```

Однако предыдущий пример имеет свои недостатки, поскольку он создает гораздо более запутанный интерфейс для класса `Matrix` (вам нужно будет рассмотреть такие методы, как `Matrix::add2()`, `Matrix::AddMultiply()` и т. Д.).

Вместо этого сделаем шаг назад и посмотрим, как мы можем адаптировать перегрузку оператора для более эффективного выполнения

Проблема связана с тем, что выражение `Matrix d = a + b + c` оценивается слишком «нетерпеливо», прежде чем у вас появилась возможность построить все дерево выражений. Другими словами, то, что вы действительно хотите достичь, - это оценить `a + b + c` за один проход и только один раз, когда вам действительно нужно назначить полученное выражение `d`.

Это основная идея шаблонов выражений: вместо того, чтобы `operator+()` сразу оценил результат добавления двух экземпляров `Matrix`, он вернет «шаблон выражения» для будущей оценки после того, как будет создано все дерево выражений.

Например, вот возможная реализация шаблона выражения, соответствующего суммированию двух типов:

```
template <typename LHS, typename RHS>
class MatrixSum
{
public:
    using value_type = typename LHS::value_type;

    MatrixSum(const LHS& lhs, const RHS& rhs) : rhs(rhs), lhs(lhs) {}

    value_type operator() (int x, int y) const {
        return lhs(x, y) + rhs(x, y);
    }
private:
    const LHS& lhs;
    const RHS& rhs;
};
```

И вот обновленная версия `operator+()`

```
template <typename LHS, typename RHS>
MatrixSum<LHS, RHS> operator+(const LHS& lhs, const RHS& rhs) {
    return MatrixSum<LHS, RHS>(lhs, rhs);
}
```

Как вы можете видеть, `operator+()` больше не возвращает «нетерпеливую оценку» результата добавления 2 экземпляров `Matrix` (это будет другой экземпляр `Matrix`), а вместо этого шаблон выражения, представляющий операцию добавления. Самый важный момент, который следует иметь в виду, заключается в том, что выражение еще не оценено. Он просто содержит ссылки на его операнды.

Фактически, ничто не мешает вам создать экземпляр `MatrixSum<>` выражения `MatrixSum<>` следующим образом:

```
MatrixSum<Matrix<double>, Matrix<double> > SumAB(a, b);
```

Вы можете, однако, на более позднем этапе, когда вам действительно нужен результат суммирования, оцените выражение `d = a + b` следующим образом:

```

for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumAB(x, y);
    }
}

```

Как вы можете видеть, еще одно преимущество использования шаблона выражения заключается в том, что вам в основном удалось оценить сумму a и b и назначить его d за один проход.

Кроме того, ничто не мешает вам комбинировать несколько шаблонов выражений. Например, $a + b + c$ приведет к следующему шаблону выражения:

```

MatrixSum<MatrixSum<Matrix<double>, Matrix<double> >, Matrix<double> > SumABC(SumAB, c);

```

И здесь вы можете оценить окончательный результат с помощью одного прохода:

```

for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumABC(x, y);
    }
}

```

Наконец, последний фрагмент головоломки состоит в том, чтобы фактически подключить ваш шаблон выражения к классу `Matrix`. Это достигается, главным образом, за счет реализации `Matrix::operator=()`, которая принимает шаблон выражения в качестве аргумента и оценивает его за один проход, как вы делали «вручную» до:

```

template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
public:
    using value_type = T;

    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

    template <typename E>
    Matrix<T, COL, ROW>& operator=(const E& expression) {
        for (std::size_t y = 0; y != rows(); ++y) {
            for (std::size_t x = 0; x != cols(); ++x) {
                values[y * COL + x] = expression(x, y);
            }
        }
        return *this;
    }

private:
    std::vector<T> values;
};

```

Прочитайте Шаблоны выписки онлайн: <https://riptutorial.com/ru/cplusplus/topic/3404/шаблоны-выписки>

глава 146: Этот указатель

замечания

`this` указатель является ключевым словом C ++ Therefore нет библиотеки , необходимой для реализации этого. И не забывайте, что `this` указатель! Таким образом, вы не можете:

```
this.someMember();
```

Когда вы обращаетесь к элементам-членам или переменным-членам из указателей, используя символ стрелки `->` :

```
this->someMember();
```

Другие полезные ссылки на лучшее понимание `this` указателя:

Что такое «этот» указатель?

<http://www.geeksforgeeks.org/this-pointer-in-c/>

https://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm

Examples

этот указатель

Все нестатические функции-члены имеют скрытый параметр, указатель на экземпляр класса, названный `this` ; этот параметр тихо вставлен в начале списка параметров и полностью обрабатывается компилятором. Когда член класса доступен внутри функции члена, она молча доступ через `this` ; это позволяет компилятору использовать одну нестационарную функцию-член для всех экземпляров и позволяет функции-члене полиморфно вызывать другие функции-члены.

```
struct ThisPointer {
    int i;

    ThisPointer(int ii);

    virtual void func();

    int  get_i() const;
    void set_i(int ii);
};
ThisPointer::ThisPointer(int ii) : i(ii) {}
// Compiler rewrites as:
ThisPointer::ThisPointer(int ii) : this->i(ii) {}
// Constructor is responsible for turning allocated memory into 'this'.
```

```

// As the constructor is responsible for creating the object, 'this' will not be "fully"
// valid until the instance is fully constructed.

/* virtual */ void ThisPointer::func() {
    if (some_external_condition) {
        set_i(182);
    } else {
        i = 218;
    }
}

// Compiler rewrites as:
/* virtual */ void ThisPointer::func(ThisPointer* this) {
    if (some_external_condition) {
        this->set_i(182);
    } else {
        this->i = 218;
    }
}

int ThisPointer::get_i() const { return i; }
// Compiler rewrites as:
int ThisPointer::get_i(const ThisPointer* this) { return this->i; }

void ThisPointer::set_i(int ii) { i = ii; }
// Compiler rewrites as:
void ThisPointer::set_i(ThisPointer* this, int ii) { this->i = ii; }

```

В конструкторе `this` можно безопасно использовать (неявно или явно) для доступа к любому уже инициализированному полю или к любому полю родительского класса; наоборот, (неявно или явно) доступ к любым полям, которые еще не были инициализированы или какие-либо поля в производном классе, является небезопасным (из-за того, что производный класс еще не сконструирован, и, следовательно, его поля не инициализируются и не существуют). Также небезопасно вызывать функции виртуального члена через `this` в конструкторе, так как любые производные функции класса не рассматриваются (из-за того, что производный класс еще не сконструирован, и, следовательно, его конструктор еще не обновил `vtable`).

Также обратите внимание, что, хотя в конструкторе тип объекта является типом, который строится конструктором. Это справедливо, даже если объект объявлен как производный тип. Например, в приведенном ниже примере `ctd_good` и `ctd_bad` являются типами `CtorThisBase` внутри `CtorThisBase()` и набирают `CtorThis` внутри `CtorThis()`, хотя их канонический тип - `CtorThisDerived`. Поскольку более производные классы построены вокруг базового класса, экземпляр постепенно переходит через иерархию классов до тех пор, пока он не будет полностью сконструированным экземпляром его предполагаемого типа.

```

class CtorThisBase {
    short s;

public:
    CtorThisBase() : s(516) {}
};

```

```

class CtorThis : public CtorThisBase {
    int i, j, k;

public:
    // Good constructor.
    CtorThis() : i(s + 42), j(this->i), k(j) {}

    // Bad constructor.
    CtorThis(int ii) : i(ii), j(this->k), k(b ? 51 : -51) {
        virt_func();
    }

    virtual void virt_func() { i += 2; }
};

class CtorThisDerived : public CtorThis {
    bool b;

public:
    CtorThisDerived() : b(true) {}
    CtorThisDerived(int ii) : CtorThis(ii), b(false) {}

    void virt_func() override { k += (2 * i); }
};

// ...

CtorThisDerived ctd_good;
CtorThisDerived ctd_bad(3);

```

С этими классами и функциями-членами:

- В хорошем конструкторе для `ctd_good` :
 - `CtorThisBase` полностью сконструирован к моменту `CtorThis` конструктора `CtorThis` . Следовательно, `s` находится в допустимом состоянии при инициализации `i` , и поэтому к нему можно получить доступ.
 - `i` инициализируется до достижения `j(this->i)` . Следовательно, `i` находится в допустимом состоянии при инициализации `j` , и поэтому к нему можно получить доступ.
 - `j` инициализируется до достижения `k(j)` . Следовательно, `j` находится в допустимом состоянии при инициализации `k` и, таким образом, можно получить доступ.
- В плохом конструкторе для `ctd_bad` :
 - `k` инициализируется после достижения `j(this->k)` . Следовательно, `k` находится в недопустимом состоянии при инициализации `j` , и доступ к нему вызывает неопределенное поведение.
 - `CtorThisDerived` не `CtorThisDerived` до тех пор, пока `CtorThis` будет построен. Следовательно, `b` находится в недопустимом состоянии при инициализации `k` , и доступ к нему вызывает неопределенное поведение.
 - Объект `ctd_bad` еще `CtorThis` , пока он не покинет `CtorThis()` , и не будет обновляться, чтобы использовать `CtorThisDerived` виртуальные таблицы «s до

CtorThisDerived() . Следовательно, virt_func() вызовет CtorThis::virt_func() , независимо от того, предназначено ли это для вызова или CtorThisDerived::virt_func() .

Использование этого указателя для доступа к данным участника

В этом контексте использование `this` указателя не является абсолютно необходимым, но это сделает ваш код более понятным для читателя, указав, что данная функция или переменная является членом класса. Пример в этой ситуации:

```
// Example for this pointer
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Class
{
public:
    Class();
    ~Class();
    int getPrivateNumber () const;
private:
    int private_number = 42;
};

Class::Class(){}
Class::~Class(){}

int Class::getPrivateNumber() const
{
    return this->private_number;
}

int main()
{
    Class class_example;
    cout << class_example.getPrivateNumber() << endl;
}
```

Смотрите в действии [здесь](#) .

Использование этого указателя для дифференциации данных и параметров элемента

Это фактическая полезная стратегия для дифференциации данных членов из параметров ... Давайте рассмотрим этот пример:

```
// Dog Class Example
#include <iostream>
#include <string>
```

```

using std::cout;
using std::endl;

/*
 * @class Dog
 *   @member name
 *     Dog's name
 *   @function bark
 *     Dog Barks!
 *   @function getName
 *     To Get Private
 *     Name Variable
 */
class Dog
{
public:
    Dog(std::string name);
    ~Dog();
    void bark() const;
    std::string getName() const;
private:
    std::string name;
};

Dog::Dog(std::string name)
{
    /*
     * this->name is the
     * name variable from
     * the class dog . and
     * name is from the
     * parameter of the function
     */
    this->name = name;
}

Dog::~Dog() {}

void Dog::bark() const
{
    cout << "BARK" << endl;
}

std::string Dog::getName() const
{
    return this->name;
}

int main()
{
    Dog dog("Max");
    cout << dog.getName() << endl;
    dog.bark();
}

```

Вы можете видеть здесь, в конструкторе мы выполняем следующее:

```
this->name = name;
```

Здесь вы можете увидеть, что имя параметра присваивается имени частной переменной из класса Dog (`this-> name`).

Чтобы увидеть вывод вышеуказанного кода: <http://cpp.sh/75r7>

этот указатель CV-Qualifiers

`this` также может быть cv-квалифицированным, как и любой другой указатель. Однако из-за того, что `this` параметр не указан в списке параметров, для этого требуется специальный синтаксис; cv-квалификаторы перечисляются после списка параметров, но перед телом функции.

```
struct ThisCVQ {
    void no_qualifier()           {} // "this" is: ThisCVQ*
    void c_qualifier() const      {} // "this" is: const ThisCVQ*
    void v_qualifier() volatile   {} // "this" is: volatile ThisCVQ*
    void cv_qualifier() const volatile {} // "this" is: const volatile ThisCVQ*
};
```

Поскольку `this` параметр, функция может быть перегружена на основе `this` cv-квалификатора (ов) .

```
struct CVOverload {
    int func()           { return 3; }
    int func() const      { return 33; }
    int func() volatile   { return 333; }
    int func() const volatile { return 3333; }
};
```

Когда `this const` (в том числе `const volatile`), функция не может записывать в него переменные-члены, неявно или явно. Единственным исключением из этого является `mutable переменные-члены`, которые могут быть записаны независимо от константы. Из-за этого `const` используется, чтобы указать, что функция-член не изменяет логическое состояние объекта (способ, которым объект появляется во внешнем мире), даже если он изменяет физическое состояние (способ, которым объект выглядит под капотом).

Логическое состояние - это способ, которым объект кажется внешним наблюдателям. Он напрямую не привязан к физическому состоянию и даже не может быть сохранен как физическое состояние. Пока внешние наблюдатели не могут видеть никаких изменений, логическое состояние является постоянным, даже если вы переворачиваете каждый бит в объекте.

Физическое состояние, также известное как побитовое состояние, заключается в том, как объект хранится в памяти. Это объектно-мелочные, необработанные 1s и 0s, которые составляют его данные. Объект является только физически постоянным, если его представление в памяти никогда не изменяется.

Следует отметить, что C++ основы `const` не на логическом состоянии, а не физическое состояние.

```
class DoSomethingComplexAndOrExpensive {
    mutable ResultType cached_result;
    mutable bool state_changed;

    ResultType calculate_result();
    void modify_somewhat(const Param& p);

    // ...

public:
    DoSomethingComplexAndOrExpensive(Param p) : state_changed(true) {
        modify_somewhat(p);
    }

    void change_state(Param p) {
        modify_somewhat(p);
        state_changed = true;
    }

    // Return some complex and/or expensive-to-calculate result.
    // As this has no reason to modify logical state, it is marked as "const".
    ResultType get_result() const;
};

ResultType DoSomethingComplexAndOrExpensive::get_result() const {
    // cached_result and state_changed can be modified, even with a const "this" pointer.
    // Even though the function doesn't modify logical state, it does modify physical state
    // by caching the result, so it doesn't need to be recalculated every time the function
    // is called. This is indicated by cached_result and state_changed being mutable.

    if (state_changed) {
        cached_result = calculate_result();
        state_changed = false;
    }

    return cached_result;
}
```

Обратите внимание, что, хотя вы технически можете использовать `const_cast` для `this` чтобы сделать его не-св-квалифицированным, вы действительно, **ДЕЙСТВИТЕЛЬНО** не должны, и вместо этого должны использовать `mutable`. `const_cast` может вызывать неопределенное поведение при использовании объекта, который фактически является `const`, в то время как `mutable` предназначен для безопасного использования. Однако, возможно, вы можете столкнуться с этим в чрезвычайно старом коде.

Исключением из этого правила является определение не-св-квалифицированных аксессоров в терминах `const` устройств; поскольку объект гарантированно не будет `const` если вызывается не-св-квалифицированная версия, нет никакого риска UB.

```
class CVAccessor {
    int arr[5];

public:
```

```

const int& get_arr_element(size_t i) const { return arr[i]; }

int& get_arr_element(size_t i) {
    return const_cast<int&>(const_cast<const CVAccessor*>(this)->get_arr_element(i));
}
};

```

Это предотвращает ненужное дублирование кода.

Как и в случае с обычными указателями, если `this` является `volatile` (в том числе `const volatile`), он загружается из памяти каждый раз, когда к нему обращаются, вместо кэширования. Это оказывает такое же влияние на оптимизацию, как и объявление любого другого указателя `volatile`, поэтому следует проявлять осторожность.

Обратите внимание, что если экземпляр имеет cv-квалификацию, единственными функциями-членами, которым разрешен доступ, являются функции-члены, у которых `this` указатель, по меньшей мере, cv-квалифицирован как сам экземпляр:

- Не-cv-экземпляры могут обращаться к любым функциям-членам.
- `const` могут обращаться к `const` и `const volatile`.
- `volatile` экземпляры могут обращаться к `volatile` и `const volatile` функциям.
- `const volatile` могут обращаться к `const volatile` функциям.

Это один из ключевых принципов [корректности const](#).

```

struct CVAccess {
    void    func()                {}
    void    func_c() const        {}
    void    func_v() volatile     {}
    void    func_cv() const volatile {}
};

CVAccess cva;
cva.func();    // Good.
cva.func_c();  // Good.
cva.func_v();  // Good.
cva.func_cv(); // Good.

const CVAccess c_cva;
c_cva.func();  // Error.
c_cva.func_c(); // Good.
c_cva.func_v(); // Error.
c_cva.func_cv(); // Good.

volatile CVAccess v_cva;
v_cva.func();  // Error.
v_cva.func_c(); // Error.
v_cva.func_v(); // Good.
v_cva.func_cv(); // Good.

const volatile CVAccess cv_cva;
cv_cva.func();  // Error.

```

```
cv_cva.func_c(); // Error.
cv_cva.func_v(); // Error.
cv_cva.func_cv(); // Good.
```

этот указатель Ref-Qualifiers

C ++ 11

Подобно `this` cv-квалификаторам, мы также можем применить *ref-qualifiers* к `*this`. Ref-qualifiers используются для выбора между семантикой нормального и rvalue-ссылок, позволяющей компилятору использовать либо семантику копирования или перемещения, в зависимости от того, что является более подходящим, и применяется к `*this` вместо `this`.

Обратите внимание, что, несмотря на ref-qualifiers с использованием ссылочного синтаксиса, `this` само по-прежнему является указателем. Также обратите внимание, что ref-qualifiers фактически не изменяют тип `*this`; это просто легче описать и понять их последствия, глядя на них, как будто они это сделали.

```
struct RefQualifiers {
    std::string s;

    RefQualifiers(const std::string& ss = "The nameless one.") : s(ss) {}

    // Normal version.
    void func() & { std::cout << "Accessed on normal instance " << s << std::endl; }
    // Rvalue version.
    void func() && { std::cout << "Accessed on temporary instance " << s << std::endl; }

    const std::string& still_a_pointer() & { return this->s; }
    const std::string& still_a_pointer() && { this->s = "Bob"; return this->s; }
};

// ...

RefQualifiers rf("Fred");
rf.func(); // Output: Accessed on normal instance Fred
RefQualifiers{}.func(); // Output: Accessed on temporary instance The nameless one
```

Функция-член не может иметь перегрузок как с реф-квалификаторами, так и без них; программист должен выбирать один или другой. К счастью, CV-классификаторы могут быть использованы в сочетании с реф-классификаторов, что позволяет [const корректности](#) правила, которым необходимо следовать.

```
struct RefCV {
    void func() & {}
    void func() && {}
    void func() const& {}
    void func() const&& {}
    void func() volatile& {}
    void func() volatile&& {}
    void func() const volatile& {}
    void func() const volatile&& {}
};
```

Прочитайте Этот указатель онлайн: <https://riptutorial.com/ru/cplusplus/topic/7146/этот-указатель>

глава 147: Явные преобразования типов

Вступление

Выражение может быть *преобразовано в явном виде* или *преобразованный* к типу `T` с помощью `dynamic_cast<T>`, `static_cast<T>`, `reinterpret_cast<T>` или `const_cast<T>`, в зависимости от того, какого типа броска предназначен.

C++ также поддерживает фрейм-нотацию в стиле функции, `T(expr)` и буквенную нотацию C-стиля, `(T)expr`.

Синтаксис

- *простой тип-спецификатор* ()
- *простой тип-спецификатор* (список выражений)
- *simple-type-specifier braced-init-list*
- *typename-specifier* ()
- *typename-specifier* (список выражений)
- *typename-specifier braced-init-list*
- `dynamic_cast < type-id > ( выражение )`
- `static_cast < type-id > ( выражение )`
- `reinterpret_cast < type-id > ( выражение )`
- `const_cast < type-id > ( выражение )`
- (тип-id) *литье-выражение*

замечания

Все шесть нотных обозначений имеют одну общую черту:

- Приведение к ссылочному типу `lvalue`, как и в `dynamic_cast<Derived&>(base)`, дает значение `l`. Поэтому, когда вы хотите что-то делать с одним и тем же объектом, но относитесь к нему как к другому типу, вы должны указывать тип ссылки `lvalue`.
- Приведение к ссылочному типу `rvalue`, как и `static_cast<string&&>(s)`, дает значение `r`.
- Приведение к не ссылочному типу, как в `(int)x`, дает значение `rvalue`, которое можно рассматривать как копию значения, которое выполняется, но с другим типом от оригинала.

Ключевое слово `reinterpret_cast` отвечает за выполнение двух разных «небезопасных» преобразований:

- Преобразования типа «**punning**», которые могут использоваться для доступа к памяти одного типа, как если бы он был другого типа.

- Конверсии [между целыми типами и типами указателей](#) в любом направлении.

Ключевое слово `static_cast` может выполнять множество различных преобразований:

- [Основание для производных](#) преобразований
- Любое преобразование, которое может быть выполнено путем прямой инициализации, включая как неявные преобразования, так и преобразования, которые вызывают явный конструктор или функцию преобразования. См. [Здесь](#) и [здесь](#) для более подробной информации.
- `void`, которое отбрасывает значение выражения.

```
// on some compilers, suppresses warning about x being unused
static_cast<void>(x);
```

- Между арифметическими и перечисляемыми типами и между различными типами перечислений. См. [Переходы перечисления](#)
- От указателя к члену производного класса, чтобы указатель на член базового класса. Указанные типы должны совпадать. См. [Производное преобразование базы для указателей на участников](#)
- `void*` [to](#) `T*`.

C++ 11

- От `lvalue` до значения `x`, как в `std::move`. См. [Перемещение семантики](#).

Examples

Основание для производной конверсии

Указатель на базовый класс может быть преобразован в указатель на производный класс с использованием `static_cast`. `static_cast` не выполняет проверку во время выполнения и может привести к неопределенному поведению, когда указатель фактически не указывает на нужный тип.

```
struct Base {};
struct Derived : Base {};
Derived d;
Base* p1 = &d;
Derived* p2 = p1; // error; cast required
Derived* p3 = static_cast<Derived*>(p1); // OK; p2 now points to Derived object
Base b;
Base* p4 = &b;
Derived* p5 = static_cast<Derived*>(p4); // undefined behaviour since p4 does not
// point to a Derived object
```

Аналогично, ссылка на базовый класс может быть преобразована в ссылку на производный класс с использованием `static_cast`.

```
struct Base {};  
struct Derived : Base {};  
Derived d;  
Base& r1 = d;  
Derived& r2 = r1; // error; cast required  
Derived& r3 = static_cast<Derived&>(r1); // OK; r3 now refers to Derived object
```

Если тип источника является полиморфным, `dynamic_cast` может использоваться для преобразования базы в производную. Он выполняет проверку времени выполнения, а отказ восстанавливается, а не создает неопределенное поведение. В случае с указателем при ошибке возвращается нулевой указатель. В ссылочном случае исключение вызывается при ошибке типа `std::bad_cast` (или класса, полученного из `std::bad_cast`).

```
struct Base { virtual ~Base(); }; // Base is polymorphic  
struct Derived : Base {};  
Base* b1 = new Derived;  
Derived* d1 = dynamic_cast<Derived*>(b1); // OK; d1 points to Derived object  
Base* b2 = new Base;  
Derived* d2 = dynamic_cast<Derived*>(b2); // d2 is a null pointer
```

Изгнание

Указатель на объект `const` может быть преобразован в указатель на объект, не `const` с использованием `const_cast` **слова** `const_cast`. Здесь мы используем `const_cast` для вызова функции, которая не является `const`. Он принимает только аргумент `non-const char*` хотя он никогда не пишет через указатель:

```
void bad_strlen(char*);  
const char* s = "hello, world!";  
bad_strlen(s); // compile error  
bad_strlen(const_cast<char*>(s)); // OK, but it's better to make bad_strlen accept const char*
```

`const_cast` для ссылочного типа может быть использован для преобразования значения `const` в значение, не соответствующее `const`.

`const_cast` опасен, поскольку он не позволяет системе типа C++ не пытаться изменить объект `const`. Это приводит к неопределенному поведению.

```
const int x = 123;  
int& mutable_x = const_cast<int&>(x);  
mutable_x = 456; // may compile, but produces *undefined behavior*
```

Преобразование типа `running`

Указатель (соответственно ссылка) на тип объекта может быть преобразован в указатель

(соотв. Ссылку) на любой другой тип объекта, используя `reinterpret_cast` . Это не вызывает никаких конструкторов или функций преобразования.

```
int x = 42;
char* p = static_cast<char*>(&x);      // error: static_cast cannot perform this conversion
char* p = reinterpret_cast<char*>(&x); // OK
*p = 'z';                             // maybe this modifies x (see below)
```

C ++ 11

Результат `reinterpret_cast` представляет тот же адрес, что и операнд, при условии, что адрес соответствующим образом выровнен для целевого типа. В противном случае результат не указан.

```
int x = 42;
char& r = reinterpret_cast<char&>(x);
const void* px = &x;
const void* pr = &r;
assert(px == pr); // should never fire
```

C ++ 11

Результат `reinterpret_cast` не указан, за исключением того, что указатель (соответствующая ссылка) выдержит круговую поездку от типа источника до типа назначения и обратно, если требование выравнивания типа назначения не является более строгим, чем требование типа источника.

```
int x = 123;
unsigned int& r1 = reinterpret_cast<unsigned int&>(x);
int& r2 = reinterpret_cast<int&>(r1);
r2 = 456; // sets x to 456
```

В большинстве реализаций `reinterpret_cast` не изменяет адрес, но это требование не было стандартизировано до C ++ 11.

`reinterpret_cast` также может использоваться для преобразования из одного типа-указателя-данных в другой или одного типа-указателя-члена в другой.

Использование `reinterpret_cast` считается опасным, поскольку чтение или запись через указатель или ссылку, полученные с использованием `reinterpret_cast` могут вызывать неопределенное поведение, когда исходный и целевой типы не связаны.

Преобразование между указателем и целым числом

Указатель объекта (включая `void*`) или указатель функции может быть преобразован в целочисленный тип с использованием `reinterpret_cast` . Это будет скомпилировано только в том случае, если тип назначения достаточно длинный. Результат определяется реализацией и обычно дает числовой адрес байта в памяти, на который указывает

указатель.

Как правило, `long` или `unsigned long` достаточно длинный, чтобы удерживать любое значение указателя, но это не гарантируется стандартом.

C++ 11

Если существуют типы `std::intptr_t` и `std::uintptr_t`, они гарантированно будут достаточно длинными, чтобы удерживать `void*` (и, следовательно, любой указатель на тип объекта). Однако они не гарантируют, что они будут достаточно длинными, чтобы удерживать указатель на функцию.

Аналогично, `reinterpret_cast` может использоваться для преобразования целочисленного типа в тип указателя. Опять же результат определяется реализацией, но значение указателя гарантируется неизменным путем кругового перехода через целочисленный тип. Стандарт не гарантирует, что значение нуля преобразуется в нулевой указатель.

```
void register_callback(void (*fp)(void*), void* arg); // probably a C API
void my_callback(void* x) {
    std::cout << "the value is: " << reinterpret_cast<long>(x); // will probably compile
}
long x;
std::cin >> x;
register_callback(my_callback,
                 reinterpret_cast<void*>(x)); // hopefully this doesn't lose information...
```

Преобразование с помощью явного конструктора или явной функции преобразования

Преобразование, которое включает вызов **явного** конструктора или функции преобразования, не может выполняться неявно. Мы можем запросить, чтобы преобразование выполнялось явно с использованием `static_cast`. Значение такое же, как и для прямой инициализации, за исключением того, что результат является временным.

```
class C {
    std::unique_ptr<int> p;
public:
    explicit C(int* p) : p(p) {}
};

void f(C c);
void g(int* p) {
    f(p); // error: C::C(int*) is explicit
    f(static_cast<C>(p)); // ok
    f(C(p)); // equivalent to previous line
    C c(p); f(c); // error: C is not copyable
}
```

Неявное преобразование

`static_cast` может выполнять любое неявное преобразование. Иногда использование

`static_cast` может быть полезным, например, в следующих примерах:

- При передаче аргументов в эллипсис «ожидаемый» тип аргумента не статически известен, поэтому не происходит никакого неявного преобразования.

```
const double x = 3.14;
printf("%d\n", static_cast<int>(x)); // prints 3
// printf("%d\n", x); // undefined behaviour; printf is expecting an int here
// alternative:
// const int y = x; printf("%d\n", y);
```

Без явного преобразования типов `double` объект будет передан эллипсису, и произойдет неопределенное поведение.

- Оператор присваивания производного класса может вызвать оператор присваивания базового класса следующим образом:

```
struct Base { /* ... */ };
struct Derived : Base {
    Derived& operator=(const Derived& other) {
        static_cast<Base&>(*this) = other;
        // alternative:
        // Base& this_base_ref = *this; this_base_ref = other;
    }
};
```

Конверсии Enum

`static_cast` может преобразовывать из целого числа или типа с плавающей точкой в тип перечисления (независимо от того, областны ли они или не определены) и *наоборот*. Он также может преобразовывать типы перечислений.

- Преобразование из неперечисленного типа перечисления в арифметический тип является неявным преобразованием; можно, но не обязательно, использовать `static_cast`.

C ++ 11

- Когда тип перечислимого типа конвертируется в арифметический тип:
 - Если значение перечисления может быть представлено точно в целевом типе, результатом будет это значение.
 - В противном случае, если тип назначения является целым типом, результат не указан.
 - В противном случае, если тип назначения является типом с плавающей точкой, результат будет таким же, как и для преобразования в базовый тип, а затем в тип с плавающей точкой.

Пример:

```
enum class Format {
    TEXT = 0,
    PDF = 1000,
    OTHER = 2000,
};
Format f = Format::PDF;
int a = f; // error
int b = static_cast<int>(f); // ok; b is 1000
char c = static_cast<char>(f); // unspecified, if 1000 doesn't fit into char
double d = static_cast<double>(f); // d is 1000.0... probably
```

- Когда целочисленный или перечисляемый тип преобразуется в тип перечисления:
 - Если исходное значение находится в пределах диапазона адресата, результатом является это значение. Обратите внимание, что это значение может быть неравным для всех счетчиков.
 - В противном случае результат не указан ($\leq$ C ++ 14) или undefined ($>$ = C ++ 17).

Пример:

```
enum Scale {
    SINGLE = 1,
    DOUBLE = 2,
    QUAD = 4
};
Scale s1 = 1; // error
Scale s2 = static_cast<Scale>(2); // s2 is DOUBLE
Scale s3 = static_cast<Scale>(3); // s3 has value 3, and is not equal to any enumerator
Scale s9 = static_cast<Scale>(9); // unspecified value in C++14; UB in C++17
```

C ++ 11

- Когда тип с плавающей точкой преобразуется в тип перечисления, результат будет таким же, как преобразование в базовый тип перечисления, а затем в тип перечисления.

```
enum Direction {
    UP = 0,
    LEFT = 1,
    DOWN = 2,
    RIGHT = 3,
};
Direction d = static_cast<Direction>(3.14); // d is RIGHT
```

Производится преобразование базы для указателей на членов

Указатель на член производного класса может быть преобразован в указатель на член базового класса с использованием `static_cast`. Указанные типы должны совпадать.

Если операнд является нулевым указателем на значение члена, результат также является нулевым указателем на значение члена.

В противном случае преобразование будет действительным только в том случае, если элемент, на который указывает операнд, фактически существует в целевом классе, или если целевой класс является базовым или производным классом класса, содержащего член, на который указывает операнд. `static_cast` не проверяет достоверность. Если преобразование недействительно, поведение не определено.

```
struct A {};  
struct B { int x; };  
struct C : A, B { int y; double z; };  
int B::*p1 = &B::x;  
int C::*p2 = p1; // ok; implicit conversion  
int B::*p3 = p2; // error  
int B::*p4 = static_cast<int B::*>(p2); // ok; p4 is equal to p1  
int A::*p5 = static_cast<int A::*>(p2); // undefined; p2 points to x, which is a member  
// of the unrelated class B  
  
double C::*p6 = &C::z;  
double A::*p7 = static_cast<double A::*>(p6); // ok, even though A doesn't contain z  
int A::*p8 = static_cast<int A::*>(p6); // error: types don't match
```

void * to T *

В C++ `void*` не может быть неявно преобразован в `T*` где `T` - тип объекта. Вместо этого `static_cast` следует использовать для явного преобразования. Если операнд фактически указывает на объект `T`, результат указывает на этот объект. В противном случае результат не указан.

C++ 11

Даже если операнд не указывает на объект `T`, если операнд указывает на байт, адрес которого правильно выровнен для типа `T`, результат преобразования указывает на тот же байт.

```
// allocating an array of 100 ints, the hard way  
int* a = malloc(100*sizeof(*a)); // error; malloc returns void*  
int* a = static_cast<int*>(malloc(100*sizeof(*a))); // ok  
// int* a = new int[100]; // no cast needed  
// std::vector<int> a(100); // better  
  
const char c = '!';  
const void* p1 = &c;  
const char* p2 = p1; // error  
const char* p3 = static_cast<const char*>(p1); // ok; p3 points to c  
const int* p4 = static_cast<const int*>(p1); // unspecified in C++03;  
// possibly unspecified in C++11 if  
// alignof(int) > alignof(char)  
char* p5 = static_cast<char*>(p1); // error: casting away constness
```

Литье в стиле C

Кастинг C-Style можно рассматривать как «Лучшее усилие» и назван так, что он является единственным литым, который может быть использован в C. Синтаксис этого

```
(NewType)variable - (NewType)variable .
```

Всякий раз, когда это приведение используется, он использует одну из следующих с ++-трансляций (по порядку):

- `const_cast<NewType>(variable)`
- `static_cast<NewType>(variable)`
- `const_cast<NewType>(static_cast<const NewType>(variable))`
- `reinterpret_cast<const NewType>(variable)`
- `const_cast<NewType>(reinterpret_cast<const NewType>(variable))`

Функциональное литье очень похоже, но в качестве нескольких ограничений в результате его синтаксиса: `NewType(expression)` . В результате могут быть применены только типы без пробелов.

Лучше использовать новую с ++-трансляцию, потому что она более читаема и может быть легко обнаружена в любом месте исходного кода на C ++, и ошибки будут обнаружены во время компиляции, а не во время выполнения.

Поскольку это приведение может привести к непреднамеренному `reinterpret_cast` , его часто считают опасным.

Прочитайте [Явные преобразования типов онлайн](https://riptutorial.com/ru/cplusplus/topic/3090/явные-преобразования-типов):

<https://riptutorial.com/ru/cplusplus/topic/3090/явные-преобразования-типов>

кредиты

S. No	Главы	Contributors
1	Начало работы с C ++	Adhokshaj Mishra, ankit dassor, aquirdturtle, ArchbishopOfBanterbury, Bakhtiar Hasan, Bart van Nierop, Ben H, Bo Persson, Brandon, Brian, BullshitPingu, cb4, celtschk, Cheers and hth. - Alf, chrisb2244, Cody Gray, Community, cpatricio, Curious, Daemon, Daksh Gupta, Danh, darkpsychic, David Bippes, David G., DeepCoder, Dim_ov, dlemstra, Donald Duck, Dr t, Dylan Little, Edward, emlai, Erick Q., ethanwu10, Fantastic Mr Fox, Florian, GIRISH kuniyal, greatwolf, honk, Humam Helfawi, Hurkyl, Ilyas Mimouni, Isak Combrinck, itzmukeshy7, Jason Watkins, JedaiCoder, Jerry Coffin, Jim Clark, Johan Lundberg, Jon Harper, jotik, Justin, Justin Time, JVApen, K48, Ken Y-N, Keshav Sharma, kiner_shah, krOoze, Leandros, maccard, Malcolm, Malick, Manan Sharma, manetsus, manlio, Marco A., Mark Gardner, MasterHD, Matt, Matt Lord, mnoronha, Muhammad Aladdin, Mustaghees, muXXmit2X, mynameisausten, Nathan Osman, Neil A., Nemanja Boric, neuro, Nicol Bolas, nouλλd λzeλO, Optimus Prime, Pavel Strakhov, Peter, Praetorian, Qchmq5, Quirk, RamenChef, Rushikesh Deshpande, SajithP, Sam Cristall, Serikov, Shoe, SirGuy, Soapy, Soul_man, theo2003, λolæz æuλ qoq, Tom K, TriskalJM, Trizzle, UncleZeiv, VermillionAzure, Walter, Wen Qin, Wexiwa, πάντα ρῆι, パスカル
2	С несовместимости	パスカル
3	Const Correctness	amanuel2, Justin Time
4	constexpr	Ajay, Brian, diegodfrf, mtb, Null
5	decltype	Ajay
6	Loops	ankit dassor, anotherGatsby, Barry, ChemiCalChems, Chris, ChrisN, Christian Rau, ColleenV, Debanjan Dhar, DrZoo, Edward, emlai, holmicz, honk, Johannes Schaub - litb, Justin Time, L.V.Rao, manlio, Nicholas, Nicol Bolas, Null, Ped7g, pmelanson, Pyves, Rakete1111, Sergey, sp2danny, user1336087, VladimirS, Yakk
7	Perfect Forwarding	In silico, Johannes Schaub - litb, Nicol Bolas, Roland

8	RAII: Приобретение ресурсов является инициализацией	Barry , defube , Jarod42 , JVApen , Loki Astari , Niall , Nicol Bolas , RamenChef , Sumurai8 , Tannin
9	RTTI: информация о времени выполнения	Brian , deepmax , Pankaj Kumar Boora , Roland , Savas Mikail KAPLAN
10	SFINAE (ошибка замены не является ошибкой)	Barry , Fox , Jarod42 , Jason R , Jonathan Lee , Luc Danton , sp2danny , SU3 , w1th0utnam3 , Xosdy , Yakk
11	static_assert	Jarod42 , JVApen , lorro , Marco A. , Richard Dally , T.C.
12	std :: function: для обертывания любого элемента, вызываемого	elimad , Evgeniy , Nicol Bolas , Tarod
13	std :: set и std :: multiset	G-Man , JVApen , Mikitori
14	авто	Artalus , Barry , celtschk , Daniele Pallastrelli , Edward , Igor Oks , Jarod42 , Johan Lundberg , manlio , Yakk
15	Арифметика с плавающей точкой	Xirema
16	Арифметическое метапрограммирование	Meena Alfons
17	Атомные типы	JVApen , Stephen
18	Атрибуты	ibrahim5253 , JVApen , Kerrek SB , MathSquared , SingerOfTheFall
19	Базовый ввод / вывод в C ++	Daemon , Nicol Bolas , Владимир Стрелец
20	Без названия	jotik , Roland , ThyReaper
21	Бит-манипуляция	A. Sarid , Barry , Cody Gray , CroCo , FedeWar , Jarod42 , JVApen , manlio , tambre , Tarod , Trevor Hickey , Алексей Неудачин
22	Более неопределенное поведение в C ++	didiz
23	Виртуальные функции участника	0x5f3759df , Daksh Gupta , Johan Lundberg , Justin Time , Motti , Sergey , T.C.

24	Возвращение нескольких значений из функции	aaron斯诺well , Bakhtiar Hasan , Barry , bitek , celtschk , Christopher Oezbek , Community , DeepCoder , Dr t , Ela782 , Fantastic Mr Fox , Galik , honk , J_T , Jarod42 , Johan Lundberg , Johannes Schaub - litb , John Slegers , Jon Chesterfield , Kevin Katzke , Let_Me_Be , Loki Astari , M. Sadeq H. E. , manetsus , Menasheh , Michael Gaskill , mnoronha , Niall , Nicol Bolas , Null , Peter , Rakete1111 , Richard Forrest , Ryan Hilbert , Stephen , T.C. , templatetypedef , tenpercent , user3384414 , Yakk , Ze Rubeus , パスカ
25	Встроенные переменные	Brian
26	Встроенные функции	amanuel2 , Aravind .KEN , Bim , Brian , legends2k
27	Вывод типа	Andrea Chua , Jim Clark
28	Вызываемые объекты	JVApen , turon
29	вычет типа	Barry , Brian , Emmanuel Mathi-Amorim
30	Генерация случайных чисел	Ha. , manlio , merlinND , Sumurai8
31	Дата и время использования заголовков	Edward , marcinj , Naor Hadar , RamenChef
32	Значение и справочная семантика	JVApen , Nicol Bolas
33	Идиома Pimpl	Danh , Daniele Pallastrelli , emlai , Jordan Chapman , JVApen , manlio , Stephen Cross , Yakk
34	Интернационализация в C ++	John Bargman
35	Исключения	Alexey Guseynov , Brian , callyalater , Dr t , Jahid , Jarod42 , Johan Lundberg , jotik , Martin Ba , Nemanja Boric , Null , Peter , Rakete1111 , Ronen Ness
36	Использование std :: unordered_map	tulak.hord
37	Использование декларации	Brian

38	итераторы	Barry, chrisb2244, cute_ptr, Daniel Jour, Edgar Rokyan, EvgeniyZh, fbrereto, Gal Dreiman, Gaurav Kumar Garg, GIRISH kuniyal, honk, Hurkyl, JPNotADragon, JVApn, Mike H-R, Null, Oz., Sergey, Serikov, tilz0R, Yakk
39	итерация	Brian, Daniel Käfer, Emmanuel Mathi-Amorim, marquesm91, RamenChef
40	Категории ценностей	Barry, ChemiCalChems, Curious, fefe, Johannes Schaub - litb, mnoronha, Nicol Bolas, Praetorian, SirGuy
41	Классы / Структуры	Alexey Voytenko, anderas, aquirdturtle, Brian, callyalater, chrisb2244, Colin Basnett, Dan Hulme, darkpsychic, Dagma, Fantastic Mr Fox, Firas Moalla, Jarod42, Jerry Coffin, jotik, Justin Time, Kerrek SB, Nicol Bolas, Null, OliPro007, PcAF, Ph03n1x, pingul, Rakete1111, Sándor Mátyás Márton, Sergey, silvergasp, Skywrath, Yakk
42	ключевое слово const	Barry, Jarod42, Jatin, Justin, Podgorskiy, tenpercent, ThyReaper
43	Ключевое слово Friend	Perette Barella, Sergey
44	ключевое слово mutable	Barry, Community, Dean Seo, start2learn, T.C., tenpercent
45	Ключевые слова	ADITYA, ammanuel2, Brian, Danh, John London, Justin Time, JVApn, Kerrek SB, Loki Astari, manlio, Marco A., Nicol Bolas, OliPro007, Rakete1111, RamenChef, Roland, start2learn
46	Ключевые слова с переменной переменной	Brian, RamenChef, start2learn
47	Ковариантность типа возврата	Cheers and hth. - Alf, sorosh_sabz
48	Компиляция и строительство	4444, Adhokshaj Mishra, Ami Tavory, ArchbishopOfBanterbury, Barry, Ben Steffan, celtschk, Curious, Donald Duck, Dr t, elvis.dukaj, Fantastic Mr Fox, Florian, greatwolf, Griffin, Isak Combrinck, Jahid, Jarod42, Jason Watkins, Johan Lundberg, jotik, Justin, Justin Time, JVApn, madduci, Malick, manetsus, manlio, Matt, Michael Gaskill, Morten Kristensen, MSD, muXXmit2X, n.m., Nathan Osman, Nemanja Boric, Peter, Quirk, Richard Dally, Sergey, Tharindu Kumara, Toby, Trygve Laugstøl, VermillionAzure

49	Константные функции члена класса	Vijayabhaskarreddy CH , Yakk
50	Контейнеры C ++	John DiFinì
51	Копирование Elision	Nicol Bolas , TartanLlama
52	Копирование и назначение	amanuel2 , Roland
53	литералы	Brian , Nikola Vasilev , RamenChef
54	Любопытно повторяющийся шаблон шаблона (CRTP)	Barry , Brian , Gabriel , honk , Nicol Bolas , Ryan Haining
55	Лямбда	Adi Lester , Aganju , Ajay , alain , anderas , Andrea Corbelli , Barry , bcmppinc , Brian , Christopher Oezbek , Community , cpplearner , derekerdmann , Edd , Falias , Firas Moalla , honk , Jean-Baptiste Yunès , Johan Lundberg , Johannes Schaub - litb , John Slegers , JVApn , Loki Astari , Loufylouf , M. Viaz , Mike Dvorkin , Nicol Bolas , Patryk , Praetorian , Rakete1111 , RamenChef , Ryan Haining , Sergio , Serikov , Snowhawk , teivaz , Yakk , ygram
56	Макет типов объектов	Brian , Justin Time
57	Массивы	Cheers and hth. - Alf , Isak Combrinck , manlio , Matthew Brien , Wen Qin , Wolf , ΦΧοcε Περεύρα Ψ
58	Метапрограммирование	anderas , Barry , Brian , celtschk , Colin Basnett , DawidPi , deepmax , dmi_ , Holt , Jarod42 , Justin , manlio , Matthieu M. , Nicol Bolas , Oz. , rhynodegreat , rtmh , sth , TartanLlama , Venki , W.F. , ysdx , πάντα ρεῖ
59	Методы рефакторинга	asantacreu , Cody Gray , Edward , Jarod42 , JVApn , RamenChef
60	Многопоточность	Alejandro , amchacon , Brian , CaffeineToCode , ComicSansMS , Dair , defube , didiz , Diligent Key Presser , Galik , James Adkison , james large , Jason Watkins , Jeremi Podlasek , mpromonet , Niall , nwp , Rakete1111 , Stephen Cross , Sumurai8 , Yakk , ysdx , Yuushi
61	Модель памяти C ++ 11	NonNumeric
62	Мьютексы	didiz , hyoslee , JVApn

63	Неопределенное поведение	Ami Tavory , AndreiM , Ben Steffan , Brian , Cody Gray , cshu , Dovahkiin , Elias Kosunen , emlai , Emma X , FedeWar , fefe , ggrr , GIRISH kuniyal , Hiura , Jeremi Podlasek , Johannes Schaub - litb , JVApEn , kd1508 , Ken Y-N , manetsus , manlio , Marco A. , Mat , mceo , Motti , Naor Hadar , nbro , Nicol Bolas , Peter , Rakete1111 , ralismark , RamenChef , Sebastian Ärleryd , Tannin , Trevor Hickey , Tyler Durden
64	Области применения	deepmax , Error - Syntactical Remorse
65	Обратный тип возврата	Brian , define cindy const , Torbjörn
66	Общие ошибки компиляции / компоновщика (GCC)	Asu , immerhart
67	Одно правило определения (ODR)	Brian , Jarod42
68	Операторы бит	Loki Astari , Mads Marquart , manlio , txtechhelp , Алексей Неудачин
69	оптимизация	chema989 , ralismark
70	Оптимизация в C ++	4444 , JVApEn , lorro , mindriot
71	Основные ключевые слова	amanuel2 , Brian , Kerrek SB , RamenChef
72	Пакеты параметров	Marco A.
73	Параллельность с OpenMP	Andrea Chua , JVApEn , Nicol Bolas , Sumurai8
74	Перегрузка оператора	ArchbishopOfBanterbury , Archie Gertsman , Ates Goral , Barry , Brian , Candlemancer , chrisb2244 , defube , enzom83 , James Adkison , Rakete1111 , Sergey , start2learn , Xeverous , Yakk
75	Перегрузка функции	Bakhtiar Hasan , Barry , Cody Gray , CoffeeandCode , didiz , Galik , Jatin , Johannes Schaub - litb , Justin Time , JVApEn , Rakete1111 , Sean , Sumurai8 , Tim Straubinger
76	Перегрузка функциональных шаблонов	Johannes Schaub - litb , Kunal Tyagi , RamenChef
77	Переместить семантику	Barry , Cheers and hth. - Alf , ChemiCalChems , David Doria ,

		didiz , Guillaume Racicot , Justin Time , JVApen
78	перечисление	Denkkar , Fantastic Mr Fox , Jarod42 , Nicol Bolas , SajithP , stackptr , T.C.
79	Поведение, определяемое реализацией	2501 , Bo Persson , Brian , Dutow , Jahid , Jarod42 , jotik , Justin Time , Iz96 , manlio , Nicol Bolas , Peter
80	Поиск зависимого имени аргумента	Fanael , Johannes Schaub - litb
81	Полиморфизм	A. Sarid , Christophe , Jarod42 , Jeremi Podlasek , Justin Time , manlio
82	Пользовательские литералы	Brian , Cid1025 , Jarod42 , Roland , sigalor , sth
83	Поля бит	Ajay , Perette Barella
84	Потоки C ++	Ami Tavory , didiz , JVApen , mpromonet , Sergey
85	Потолочные манипуляторы	Nicol Bolas , Владимир Стрелец
86	Правило три, пять и ноль	Adrien Descamps , Barry , ChrisN , hello , honk , Johan Lundberg , Justin Time , JVApen , Loki Astari , mpromonet , Nicol Bolas , Nirmal4G , NonNumeric , Null , Peter , relgukxilef , Scott Weldon , T.C. , TriskalJM , Venemo
87	препроцессор	alain , callyalater , Cheers and hth. - Alf , CygnusX1 , Fantastic Mr Fox , Fox , Francisco P. , Ian Ringrose , immerhart , InitializeSahib , Johan Lundberg , Justin , Justin Time , Ken Y-N , Kieran Chandler , krOoze , manlio , Marco A. , Maxito , n.m. , Nicol Bolas , Peter , phandinhlan , Richard Dally , Sean , signal , silvergasp , Sumurai8 , T.C. , Tanjim Hossain , tenpercent , The Philomath , Владимир Стрелец , パスカル
88	Примеры клиентского сервера	Abhinav Gauniyal
89	приоритет оператора	an0o0nym , Brian , didiz , JVApen , start2learn , turon
90	Пространства имен	anderas , Andrea Chua , Barry , Brian , DeepCoder , emlai , Isak Combrinck , Jarod42 , Jérémy Roy , Johannes Schaub - litb , Julien-L , JVApen , Nicol Bolas , Null , Rakete1111 , randag , Roland , T.C. , tenpercent , Yakk

91	профилирование	Ami Tavory , paul-g
92	Разрешение перегрузки	Barry , Brian , didiz , Johannes Schaub - litb
93	Реализация шаблона проектирования в C ++	Antonio Barreto , datosh , didiz , Jarod42 , JVApen , Nikola Vasilev
94	Регулярные выражения	honk , Jonathan Mee , Justin , JVApen
95	Рекомендации	Andrea Corbelli , Asu , Daksh Gupta , darkpsychic , rockoder
96	Рекурсивный Мьютекс	didiz
97	Рекурсия в C ++	celtschk , R_Kapp
98	Сгибание выражений	AndyG , Barry , cpplearner , Firas Moalla , Marco A. , Rakete1111 , T.C. , Yakk
99	семафор	didiz
100	Системы сборки	Ami Tavory , celtschk , Florian , Jahid , Jason Watkins , Justin , JVApen , Nathan Osman , RamenChef , VermillionAzure
101	Сортировка	anatolyg , Barry , Daniel , Ivan Kush , maccard , manetsus , manlio , MikeMB , MKAROL , Nicol Bolas , Patrick , Ravi Chandra , SajithP , timrau , Trevor Hickey
102	Союзы	manlio , ThyReaper , txtechhelp
103	Специальные функции участников	Barry , krOoze , OliPro007 , Reuben Thomas , TriskalJM
104	Спецификаторы класса хранения	Brian , start2learn
105	Сравнение сборок с классическими примерами C ++, разрешенными с помощью C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17	wasthishepful
106	Средства и методы отладки и отладки C ++	Adam Trhon , JVApen , King's jester , Misgevolution
107	станд :: forward_list	Nikola Vasilev

108	станд :: integer_sequence	Dietmar Kühl
109	станд :: iomanip	kiner_shah , Nikola Vasilev , Yakk
110	станд :: вариант	Yakk
111	станд :: вектор	2power10 , A. Sarid , Aaron Stein , alain , Alex Logan , Ami Tavory , anatolyg , anderas , Andy , AndyG , arunmoezhi , Bakhtiar Hasan , Barry , Benjamin Lindley , bluefog , bone , CHess , CinCout , Cody Gray , Colin Basnett , ComicSansMS , Community , cute_ptr , Daksh Gupta , Daniel , Daniel Stradowski , Dario , David G. , David Yaw , DeepCoder , diegodfrf , dkg , Dr t , Duly Kinsky , Ed Cottrell , Edward , ehudt , emlai , enrico.bacis , Falias , Fantastic Mr Fox , Fox , foxcub , gaazkam , Galik , gartenriese , granmirupa , Holt , honk , Hurkyl , iliketocode , immerhart , Isak Combrinck , Jarod42 , Jason Watkins , JHBonarius , Johan Lundberg , John Slegers , jotik , jpo38 , JVApn , Kevin Katzke , krOoze , Loki Astari , lordjohncena , manetsus , manlio , Marco A. , Matt , Michael Gaskill , Misha Brukman , MotKohn , Motti , mtk , NageN , Niall , Nicol Bolas , Null , patmanpato , Paul Beckingham , paul-g , Ped7g , Praetorian , Pyves , R. Martinho Fernandes , Rakete1111 , Randy Taylor , Richard Dally , Roddy , Romain Vincent , Rushikesh Deshpande , Ryan Hilbert , Saint-Martin , Samar Yadav , Samer Tufail , Sayakiss , Serikov , Shoe , silvergasp , Skipper , solidcell , Stephen , sth , strangeqargo , T.C. , Tamarous , theo2003 , Tom , towj , Trevor Hickey , TriskalJM , user1336087 , user2176127 , Vladimir Gamalyan , Wolf , Yakk
112	станд :: Карта	Andrea Corbelli , ankit dassor , ChrisN , CinCout , ComicSansMS , CygnusX1 , davidsheldon , diegodfrf , Fantastic Mr Fox , foxcub , Galik , honk , jmmut , manetsus , manlio , Meysam , Naveen Mittal , Null , Peter , Richard Dally , rick112358 , Savan Morya , user1336087 , vdaras , Volka , Wyzard
113	станд :: любой	demonplus , Marco A.
114	станд :: массив	CinCout , Daksh Gupta , Dinesh Khandelwal , Error - Syntactical Remorse , Malcolm , Nikola Vasilev , plasmacel
115	станд :: опционально	Barry , diegodfrf , Jahid , Jared Payne , JVApn , Null , Yakk
116	станд :: пара	Ajay , Bim , demonplus , kiner_shah , Nikola Vasilev , Ravi Chandra

117	станд :: строка	1337ninja , 3442 , Andrea Corbelli , Barry , Bim , caps , Christopher Oezbek , cplearner , crea7or , Curious , drov , Edward , Emil Rowland , emlai , Fantastic Mr Fox , fbrereto , ggrr , Holt , honk , immerhart , Jack , Jahid , Jerry Coffin , Jonathan Mee , jotik , JPNotADragon , jpo38 , Justin , JVApen , Ken Y-N , Leandros , Loki Astari , manetsus , manlio , Marc.2377 , Matthew , Matthieu M. , Meysam , Michael Gaskill , mpromonet , Niall , Null , Rakete1111 , RamenChef , Richard Dally , SajithP , Serikov , sigalor , Skipper , Soapy , sth , T.C. , Tharindu Kumara , Trevor Hickey , user1336087 , user2176127 , W.F. , Wolf , Yakk
118	Стандарт ISO C ++	Bakhtiar Hasan , Barry , C.W.Holeman II , ComicSansMS , didiz , diegodfrf , Guillaume Pascal , Ivan Kush , Johan Lundberg , Justin Time , JVApen , manlio , Marco A. , MSalters , Nicol Bolas , sth , vishal
119	Стандартные библиотечные алгоритмы	Ami Tavory , Barry , Daniel , Duly Kinsky , Edgar Rokyan , Guillaume Pascal , Jarod42 , NinjaDeveloper , Patryk Obara , Peter , Riom
120	СТД :: атомарных	Nikola Vasilev
121	Структуры данных в C ++	Gaurav Sehgal
122	Структуры синхронизации потоков	didiz , Galik , JVApen
123	Тестирование модулей в C ++	elvis.dukaj , VermillionAzure
124	Тип Erasure	Brian , celtschk , greatwolf , Jarod42 , Yakk
125	Тип Ключевые слова	Brian , Justin Time , Omnifarious , RamenChef
126	Типед и псевдонимы типов	Brian
127	Типовые черты	Jarod42 , silvergasp , TartanLlama
128	указатели	Baron , daB0bby , FedeWar , Hindrik Stegenga , Nicol Bolas , Nitinkumar Ambekar , Pietro Saccardi , Reverie Wisp , West
129	Указатели для участников	John Burger , start2learn
130	Умные указатели	Abyx , Ajay , Alexey Voytenko , anderas , Barry ,

		CaffeineToCode , Christopher Oezbek , Cody Gray , ComicSansMS , cpplearner , Daksh Gupta , Danh , Daniele Pallastrelli , DeepCoder , Edward , emlai , foxcub , Francis Cugler , honk , Jack Zhou , Jared Payne , Jarod42 , Johan Lundberg , Johannes Schaub - litb , jotik , Justin , JVApen , Kerrek SB , King's jester , Loki Astari , manlio , Marco A. , MC93 , Menasheh , Meysam , PcAF , Rakete1111 , Reuben Thomas , Richard Dally , rodrigo , Roland , sami1592 , sth , Sumurai8 , tysonite , user3684240 , Xirema , Yakk
131	Управление памятью	Andrei , Brian , callyalater , Daksh Gupta , Galik , JVApen , madduci , nnrales , RamenChef , ThyReaper
132	Управление потоком	anotherGatsby , Brian , JVApen , mkluwe , Qchmq5 , RamenChef , Tejendra
133	Управление ресурсами	Anonymous1847
134	Файловый ввод-вывод	anderas , ankit dassor , Anonymous1847 , AProgrammer , Bakhtiar Hasan , bitek , Chachmu , ComicSansMS , didiz , Dietmar Kühl , Dr t , Emanuel Vintilă , Galik , honk , Hurkyl , Jérémie Bolduc , John Strood , JVApen , Loki Astari , manlio , Mathieu K. , MikeMB , mindriot , Nicol Bolas , nwp , patmanpato , Rakete1111 , RomCoo , Serikov , sheng09 , shrike , svgspnr , Алексей Неудачин
135	Файлы заголовков	RamenChef , VermillionAzure
136	Функции нестатического пользователя	Justin Time , RamenChef
137	Функция C ++ «вызов по значению» против «вызов по ссылке»	Error - Syntactical Remorse , Henkersmann
138	Фьючерсы и обещания	didiz , Nicol Bolas
139	Характеристики связи	Brian
140	центровка	Brian , Marco A. , Nicol Bolas
141	Цифровые разделители	diegodfrf , JVApen
142	Шаблон дизайна Singleton	deepmax , Galik , Jarod42 , Johan Lundberg , JVApen , Stradigos

143	Шаблоны	Barry , Benjy Kessler , Brian , callyalater , cb4 , celtschk , CodeMouse92 , Colin Basnett , DeepCoder , Diligent Key Presser , Eldritch Cheese , eXPerience , FedeWar , Gabriel , Greg , Holt , honk , J_T , Johannes Schaub - litb , Justin , JVApn , Loki Astari , M. Viaz , manlio , Maxito , MSalters , Nicol Bolas , Pontus Gagge , Praetorian , Rakete1111 , Ricardo Amores , Ryan Haining , Sergey , SirGuy , Smeeheey , Sumurai8 , user1887915 , W.F. , WMios , Wolf , πάντα ῥεῖ
144	Шаблоны выписки	Ajay , BigONotation , celtschk , defube , Jarod42 , Jonathan Lee , Roland , T.C. , Yakk
145	Этот указатель	amanuel2 , Justin Time , RamenChef
146	Явные преобразования типов	4444 , Brian , JVApn , Nikola Vasilev