



無料電子ブック

学習

C++

Free unaffiliated eBook created from
Stack Overflow contributors.

#C++

.....	1
1: C ++	2
.....	2
.....	2
Examples	2
.....	2
.....	2
.....	4
1	4
C/	4
.....	5
.....	5
.....	5
.....	6
.....	6
.....	7
.....	7
.....	7
-	8
.....	8
C ++	10
.....	10
2: 1ODR	12
Examples	12
.....	12
.....	12
ODR	13
3: 350	15
Examples	15
.....	15
.....	

3.....	17
.....	19
4: Arithmetic.....	20
.....	20
Examples.....	20
Olog n.....	20
5: C ++ 11.....	22
.....	22
.....	22
.....	23
.....	23
.....	23
-	23
.....	24
Examples.....	24
.....	24
.....	26
6: C ++	27
.....	27
Examples.....	27
C ++.....	27
7: C ++	30
.....	30
Examples.....	30
.....	30
.....	31
.....	31
.....	31
.....	32
.....	

.....	32
iostream.....	33
.....	33
.....	33
.....	33
.....	34
.....	34
STL.....	34
.....	35
.....	35
8: C ++.....	36
Examples.....	36
C ++.....	36
9: C ++.....	39
.....	39
.....	39
Examples.....	39
.....	39
.....	42
.....	44
Fluent APIBuilder.....	45
.....	47
.....	47
10: C ++.....	49
.....	49
Examples.....	49
Google.....	49
.....	49
.....	49

11: C ++	51
Examples	51
FibonnaciFibonnaci	51
	51
12: C ++	53
	53
Examples	53
C ++	53
13: C ++	55
	55
Examples	55
	55
14: C ++	57
Examples	57
	57
	57
	58
	58
1	58
/	59
	59
	60
	60
	61
15: C ++	63
	63
Examples	63
	63
16: C ++	64
	64
	64

Examples.....	64
C ++segfault-valgrind.....	64
GDBSegfault.....	66
.....	67
.....	68
/.....	69
.....	69
.....	69
.....	69
.....	69
.....	69
.....	70
.....	70
.....	70
.....	70
.....	71
.....	71
.....	71
.....	71
17: C ++ "value by call" "reference by call".....	72
.....	72
Examples.....	72
.....	72
18: Const Correctness.....	74
.....	74
.....	74
Examples.....	74
.....	74
Const.....	75
Const.....	77
Const Correctness.....	79
const CV.....	79
const.....	80

19: constexpr	83
.....	83
.....	83
Examples	83
constexpr	83
constexpr	85
if	87
20: const	88
.....	88
.....	88
Examples	88
Const	88
Const	88
Const	89
constconst getter	89
21: C	92
.....	92
Examples	92
.....	92
.....	92
.....	92
22: Elision	93
Examples	93
.....	93
.....	94
elision	95
elision	95
elision	96
.....	96
23: ISO C ++	98
.....	98
.....	98

Examples.....	98
.....	98
C ++ 11.....	99
.....	99
.....	99
.....	99
.....	100
.....	100
.....	100
.....	100
.....	100
.....	100
.....	101
.....	101
C ++ 14.....	101
.....	101
.....	101
/.....	102
C ++ 17.....	102
.....	102
.....	102
C ++ 03.....	102
.....	102
C ++ 98.....	103
C89 / C90.....	103
.....	103
C ++ 20.....	103
.....	103
.....	103
24: OpenMP.....	104

.....	104
.....	104
Examples.....	104
OpenMP.....	104
OpenMP.....	105
OpenMPParallel For Loop.....	105
OpenMP/.....	106
25: Pimpl Idiom.....	108
.....	108
Examples.....	108
Pimpl.....	108
26: RAII.....	110
.....	110
Examples.....	110
.....	110
/ ScopeExit.....	111
ScopeSuccessc ++ 17.....	112
ScopeFailc ++ 17.....	113
27: RTTI.....	115
Examples.....	115
.....	115
dynamic_cast.....	115
typeid.....	115
C ++.....	116
28: SFINAE.....	117
Examples.....	117
enable_if.....	117
.....	117
void_t.....	118
decltype.....	120
SFINAE.....	121
enable_if_all / enable_if_any.....	122

is_detected.....	123
.....	125
29: static_assert.....	127
.....	127
.....	127
.....	127
Examples.....	127
static_assert.....	127
30: std :: any.....	129
.....	129
Examples.....	129
.....	129
31: std :: array.....	130
.....	130
.....	130
Examples.....	130
std :: array.....	130
.....	131
.....	133
.....	134
.....	134
32: std :: atomics.....	135
Examples.....	135
.....	135
33: std :: forward_list.....	138
.....	138
.....	138
Examples.....	138
.....	138
.....	139
34: std :: function.....	141

Examples.....	141
.....	141
std :: bindstd :: function.....	141
std :: functionlambdastd :: bind.....	142
``.....	143
std :: function.....	144
std :: tuple.....	146
35: std :: integer_sequence.....	148
.....	148
Examples.....	148
std :: tuple.....	148
.....	149
.....	149
36: std :: iomanip.....	151
Examples.....	151
std :: setw.....	151
std :: setprecision.....	151
std :: setfill.....	152
std :: setiosflags.....	152
37: std :: map.....	154
.....	154
Examples.....	154
.....	154
std :: mapstd :: multimap.....	155
.....	156
.....	157
std :: mapstd :: multimap.....	158
std :: mapstd :: multimap.....	159
.....	159
.....	160
.....	160
.....	160

std :: map	161
.....	161
38: std :: optional	163
Examples	163
.....	163
.....	163
.....	163
Sentinel	163
vs std::pair<bool, T>	163
.....	163
.....	164
.....	165
value_or	166
39: std :: pair	167
Examples	167
.....	167
.....	167
40: std :: setstd :: multiset	169
.....	169
.....	169
Examples	169
.....	169
.....	170
.....	170
.....	172
.....	172
.....	172
.....	173
.....	173
.....	174
41: std :: string	176

.....	176
.....	176
.....	176
Examples.....	177
.....	177
.....	177
.....	178
.....	178
.....	179
.....	179
[]n.....	180
atn.....	180
.....	180
.....	180
.....	180
constchar *.....	181
.....	182
/.....	183
.....	184
std :: wstring.....	185
std :: string_view.....	186
.....	187
/.....	187
.....	188
.....	189
std :: string.....	190
42: std :: unordered_map.....	192
.....	192
.....	192
Examples.....	192
.....	192
.....	

43: std :: variant	193
.....	193
Examples	193
::	193
.....	194
`std :: variant`	195
44: std :: vector	196
.....	196
.....	196
Examples	196
std :: vector	196
.....	197
:: vector	198
.....	199
.....	199
const	200
.....	201
.....	201
.....	201
.....	203
std :: vectorC	204
/	205
.....	206
.....	206
.....	206
.....	206
.....	206
.....	206
.....	207
.....	207

.....	207
.....	207
std :: vector.....	208
std :: vector.....	210
.....	210
.....	211
.....	213
.....	214
.....	214
.....	215
.....	216
.....	217
45: typedef.....	219
.....	219
.....	219
Examples.....	219
typedef.....	219
typedef.....	219
typedef.....	220
"using".....	220
46:	221
.....	221
.....	221
Examples.....	221
.....	221
.....	222
47:	223
Examples.....	223
C.....	223
.....	223
.....	224
.....	

	224
	225
	226
	226
	227
	227
	228
	229
	229
	230
48:	232
	232
Examples	232
	232
49:	233
	233
	233
	233
	233
	233
	234
Examples	234
	234
	234
	234
	234
50:	236
	236
Examples	236
	236

.....	237
.....	237
.....	238
C ++ 14.....	238
.....	239
51:	240
.....	240
Examples.....	240
.....	240
.....	242
.....	242
.....	243
.....	243
.....	243
52:	244
.....	244
.....	244
.....	244
Examples.....	246
asm.....	246
.....	246
noexcept.....	247
.....	248
.....	249
.....	249
53:	255
Examples.....	255
TCP.....	255
TCP.....	258
54: /	260
.....	260
.....	260

Examples.....	260
.....	260
.....	261
.....	262
.....	264
.....	265
.....	266
.....	267
.....	268
.....	269
.....	269
/.....	270
.....	275
.....	278
.....	283
/.....	284
55:	287
.....	287
Examples.....	287
.....	287
.....	289
.....	290
CV-.....	291
.....	294
56:	296
.....	296
.....	296
Examples.....	296
GCC.....	296
.....	297
Visual C ++.....	298
Visual Studio - Hello World.....	302

Clang.....	309
.....	310
C ++.....	311
Code :: BlocksGraphical interface.....	313
57: C ++C ++ 11C ++ 14C ++ 17C ++.....	318
Examples.....	318
.....	318
58:	320
.....	320
Examples.....	320
.....	320
.....	321
.....	322
.....	322
59:	324
Examples.....	324
.....	324
.....	324
60:	326
.....	326
.....	326
Examples.....	328
.....	328
.....	334
.....	335
61:	337
.....	337
.....	337
Examples.....	337
std :: shared_ptr.....	337
std :: weak_ptr.....	339
std :: unique_ptr.....	341

C.....	343
auto_ptr.....	345
shared_ptr.....	346
std :: shared_ptr.....	347
value_ptr.....	348
62:	351
.....	351
.....	351
.....	351
Examples.....	351
.....	351
.....	352
std :: thread.....	352
.....	354
std :: threadstd :: async.....	355
.....	356
.....	356
.....	356
.....	357
.....	358
.....	358
.....	360
.....	362
63:	363
.....	363
Examples.....	363
std :: shared_lock.....	363
std :: call_oncestd :: once_flag.....	363
.....	364
std :: condition_variable_anystd :: cv_status.....	365
64:	366
.....	366

Examples.....	366
C ++ 11.....	366
.....	366
65:	368
Examples.....	368
.....	368
.....	368
.....	370
On²Onstd :: move.....	371
.....	374
.....	374
66:	376
Examples.....	376
.....	376
.....	377
.....	377
.....	378
67:	380
.....	380
Examples.....	380
.....	380
.....	380
.....	381
68:	383
.....	383
Examples.....	383
.....	383
vtable.....	384
`std :: function`.....	387
T.....	389
std :: any.....	391
69:	396
.....	

Examples.....396

.....396

.....396

.....396

.....396

std :: is_same.....397

.....398

.....399

70: 401

.....401

.....401

.....401

Examples.....403

.....403

.....404

.....405

.....406

.....406

.....407

.....408

.....408

auto.....409

unique_ptr.....410

.....410

.....411

.....414

71: 416

Examples.....416

.....416

.....416

72: 417

.....	417
.....	417
Examples.....	418
.....	418
73:	419
.....	419
Examples.....	419
.....	419
C.....	419
std :: bitset.....	419
.....	419
C.....	419
std :: bitset.....	419
.....	420
C.....	420
std :: bitset.....	420
.....	420
C.....	420
std :: bitset.....	420
nx.....	421
C.....	421
std :: bitset.....	421
.....	421
C.....	421
std :: bitset.....	421
.....	421
C.....	421
.....	422
2.....	423
.....	423

74:	425
.....	425
Examples.....	425
- AND.....	425
- OR.....	426
^ - XOR.....	426
NOT.....	427
<< -	428
>> -	429
75:	431
.....	431
.....	431
Examples.....	431
CMake.....	431
GNU make.....	432
.....	432
.....	432
.....	433
.....	434
SCons.....	434
.....	435
.....	435
NMAKEMicrosoft.....	435
.....	435
GNU.....	436
.....	436
76: I / O	437
.....	437
Examples.....	437
.....	437
.....	438
.....	

.....	441
.....	442
.....	442
ASCIIstd :: string.....	443
.....	444
`struct`.....	445
.....	446
.....	446
.....	447

77: 449

.....	449
.....	449

Examples..... 449

.....	449
.....	450
.....	452
.....	455
.....	456
X.....	457
#pragma once.....	459
.....	459

78: 461

.....	461
Examples.....	461
.....	461
.....	462
.....	462

79: 464

.....	464
Examples.....	464
.....	464
.....	

.....464

.....465

if.....465

else.....466

.....466

.....466

.....467

.....468

ifif..else.....468

.....469

80: 473

Examples.....473

gccgprof.....473

gperf2dot.....474

gccGoogle PerfCPU.....475

81: 478

.....478

Examples.....478

.....478

.....478

.....479

.....480

82: 481

.....481

.....481

.....481

Examples.....481

.....481

.....481

.....482

.....483

.....	483
.....	484
.....	484
/	484
/	485
.....	485
83:	486
.....	486
std :: shared_timed_mutexstd :: shared_mutex	486
std :: shared_mutexMSVC14.1	486
std :: shared_timed_mutexMSVC14.1	487
std :: shared_mutexstd :: shared_timed_mutex2	491
Examples	494
std :: unique_lockstd :: shared_lockstd :: lock_guard	494
std :: try_to_lockstd :: ado_lockstd :: defer_lock	495
std :: mutex	496
std :: scoped_lockC ++ 17	496
.....	497
std :: lock	497
84:	498
.....	498
.....	498
Examples	498
.....	498
.....	500
std :: integer_sequence	501
.....	502
.....	503
C ++ 11	504
T	505
If-then-else	506
/	507

85:	508
.....	508
.....	508
Examples.....	508
.....	508
.....	509
.....	510
86:	512
.....	512
Examples.....	512
.....	512
.....	513
.....	513
.....	514
87:	516
Examples.....	516
double.....	516
.....	516
.....	517
.....	517
.....	518
88:	520
.....	520
.....	520
.....	520
Examples.....	521
.....	521
.....	523
.....	524
.....	525
.....	526
.....	527

.....	527
.....	529
.....	529
C ++ 03.....	531
.....	532
std::functionstd::function.....	532
2.....	532
Y-.....	533
lambdas.....	534
89:	536
.....	536
Examples.....	536
.....	536
.....	536
nullptr.....	536
.....	537
.....	537
90:	539
.....	539
Examples.....	539
.....	539
.....	541
91:	543
.....	543
.....	543
.....	543
Examples.....	543
Unix.....	543
CC ++.....	543
92:	545
.....	545
.....	545

.....	545
Examples.....	545
For.....	545
.....	548
while.....	550
.....	551
Do-while.....	552
.....	553
-	554
93: /GCC.....	556
Examples.....	556
'***'	556
.....	556
.....	556
`***'	557
***	558
94: CRTP.....	559
.....	559
Examples.....	559
CRTP.....	559
CRTP.....	560
95:	563
.....	563
Examples.....	563
TU.....	563
.....	563
void * value.....	564
reinterpret_cast.....	564
.....	565
.....	565
.....	566
.....	567

96:	568
.....	568
Examples.....	568
.....	568
.....	568
compare.....	569
C ++ 11.....	570
.....	571
std :: map.....	572
.....	573
97:	575
.....	575
Examples.....	575
.....	575
.....	576
.....	576
98:	578
.....	578
.....	578
Examples.....	578
C ++ 11.....	578
.....	579
.....	580
.....	581
.....	582
99:	584
Examples.....	584
.....	584
2.....	584
100:	586
Examples.....	586
.....	586

.....	587
.....	588
.....	588
.....	589
const.....	589
.....	590
std :: uncaught_exceptions.....	592
.....	593
101:	596
Examples.....	596
.....	596
.....	598
102:	600
Examples.....	600
.....	600
.....	600
xvalue.....	601
.....	601
glvalue.....	601
.....	602
103:	604
.....	604
Examples.....	604
std :: futurestd :: promise.....	604
.....	604
std :: packaged_taskstd :: future.....	605
std :: future_errorstd :: future_errc.....	605
std :: futurestd :: async.....	607
.....	608
104:	609
Examples.....	609
std :: recursive_mutex.....	609

105:	610
Examples	610
.....	610
switch	610
.....	611
.....	612
C ++11	613
106:	614
.....	614
.....	614
.....	614
Examples	614
.....	614
.....	615
.....	616
107:	618
.....	618
.....	618
Examples	618
.....	618
108:	620
Examples	620
.....	620
C ++	620
109:	622
Examples	622
.....	622
.....	622
.....	622
.....	622
while	623
for	623

110:	624
Examples	624
.....	624
.....	624
111:	626
Examples	626
.....	626
.....	626
.....	626
.....	627
112:	628
.....	628
.....	628
.....	628
Examples	628
.....	628
.....	630
.....	630
.....	631
.....	632
ADL	632
.....	633
/	635
.....	635
.....	636
.....	636
.....	637
113:	638
.....	638
.....	638
Examples	638
.....	638

operatorFunctor	639
114:	640
.....	640
.....	640
Examples	640
.....	640
.....	640
.....	641
115:	642
Examples	642
int	642
.....	642
.....	642
char16_t	642
char32_t	642
.....	643
.....	643
.....	643
.....	644
.....	644
wchar_t	644
116:	645
Examples	645
const	645
.....	645
.....	646
.....	646
.....	646
117:	648
Examples	648
.....	648
.....	649
.....	

118:	652
.....	652
Examples.....	652
.....	652
119:	654
.....	654
Examples.....	654
.....	654
120:	656
Examples.....	656
Char.....	656
.....	656
char	656
.....	656
char16_t char32_t	658
bool	658
wchar_t	658
.....	659
1.....	659
.....	659
.....	661
.....	662
.....	662
.....	662
121:	663
.....	663
.....	663
.....	663
Examples.....	663
.....	663

.....	663
.....	664
122:	665
.....	665
Examples.....	665
.....	665
.....	665
123:	667
.....	667
Examples.....	667
[[]].....	667
[[fallthrough]].....	668
[[deprecated]][[deprecated "reason"]].....	669
[[nodiscard]].....	669
[[maybe_unused]].....	670
124:	672
Examples.....	672
.....	672
vec.hhstd :: vector.....	674
expr.hh.....	675
main.ccsrc.....	678
.....	681
125:	685
.....	685
Examples.....	685
.....	685
.....	686
.....	686
126:	688
Examples.....	688
.....	688
127:	690

.....	690
Examples.....	690
1.....	690
2.....	691
3.....	692
128:	694
.....	694
.....	694
.....	694
Examples.....	695
.....	695
.....	696
.....	696
.....	697
.....	697
.....	698
.....	698
.....	699
void *T *.....	700
C.....	700
129:	702
.....	702
Examples.....	702
/.....	702
.....	702
130:	704
.....	704
.....	704
Examples.....	705
.....	705
voidreturn.....	705
.....	705
.....	

.....	706
.....	706
.....	707
1.....	708
.....	709
.....	709
.....	710
.....	710
.....	711
.....	711
`std`` posix`.....	712
.....	712
.....	713
.....	713
const.....	713
.....	714
.....	714
.....	714
.....	715
[noreturn].....	716
.....	716
.....	717
131:	718
.....	718
.....	718
Examples.....	718
.....	718
.....	718
132:	719
Examples.....	719
.....	719

133:	720
Examples	720
std :: for_each	720
std :: next_permutation	720
std :: accumulate	721
std :: find	722
std :: count	723
std :: count_if	724
std :: find_if	726
std :: min_element	727
std :: nth_element	728
134:	730
	730
	730
	730
Examples	730
regex_matchregex_search	730
regex_replace	731
regex_token_iterator	731
regex_iterator	731
	732
	733
	734
135:	735
Examples	735
	735
136:	737
	737
	737
Examples	737
	737
	739

.....741

.....741

.....742

.....743

NOT.....744

I / O.....744

.....745

.....749

137: 752

.....752

Examples.....752

.....752

AND.....753

&&||.....753

.....754

138: 756

Examples.....756

.....756

.....757

.....757

.....759

.....761

139: 764

.....764

Examples.....764

.....764

.....764

.....765

140: 766

.....766

.....766

Examples.....	766
.....	766
.....	767
.....	767
.....	768
extern.....	768
141:	770
.....	770
Examples.....	770
.....	770
.....	771
142:	773
.....	773
Examples.....	773
.....	773
.....	774
.....	774
.....	775
.....	776
.....	777
.....	778
.....	779
143:	781
.....	781
Examples.....	781
.....	781
.....	782
std :: vector.....	783
2D.....	783
std :: vector.....	784
.....	786
144:	788
.....	

Examples.....788

.....788

std :: tuple.....788

std :: array.....790

std :: pair.....790

.....790

.....791

.....793

std :: vector.....794

.....794

145: 796

.....796

Examples.....796

.....796

146: 797

.....797

.....797

Examples.....797

.....797

.....798

cv-qualifier.....799

147: 802

.....802

.....802

Examples.....802

.....802

.....803

.....804

.....806

Const Correctness.....808

.....810

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

1: C ++をいめる

'Hello World'プログラムは、にコンパイラとライブラリのをするためにできるのです。 `<iostream>` `std::cout` とにC ++ライブラリをし、コンパイルにユーザーエラーがするのをにえるために、コンパイルするファイルは1つだけです。

C ++プログラムをコンパイルするプロセスは、コンパイラとオペレーティングシステムによってになります。「[コンパイルとビルド](#)」のトピックには、さまざまなコンパイラのさまざまなプラットフォームでC ++コードをコンパイルするのがまれています。

バージョン

バージョン		
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		2017-01-01
C ++ 20		2020-01-01

Examples

こんにちは

このプログラムは、 `Hello World!` プリントし `Hello World!` ストリームにします。

```
#include <iostream>

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

それは[Coliru](#)にきててください。

このコードのをしくべてみましょう。

- `#include <iostream>`は、C++ヘッダファイル `iostream` を含むプリプロセッサディレクティブです。

`iostream`は、のストリームとストリームのをむのライブラリヘッダーファイルです。これらは、です `std`にまれています。

II/Oストリームは、プログラムがシステムはからをするをします。

- `int main() { ... }`は、`main`というのしいをします。により、`main`はプログラムのにひされます。C++プログラムには`main`が1つしかなくてはならず、に`int`のをすがあります。

ここで、`int`はのりと呼ばれるものです。`main`がすはコードです。

によって、プログラムコード`0`または`EXIT_SUCCESS`は、プログラムをするシステムによってとしてされます。そののりコードはエラーとしています。

`return`がない、`main`したがってプログラムはデフォルトで`0`をします。このでは、`return 0;`をにくはありません `return 0;`。

`void`をすをくのすべてののは、りのにってにをすがあります。まったくさないでください。

- `std::cout << "Hello World!" << std::endl;`「こんにちは」とされます。ストリームにします。

- `::std`はであり、`::`はのによるオブジェクトのルックアップをにするスコープです。

くがあります。ここでは、`::`をして `std` から `cout` をいたいことをします。については、[Scope Resolution Operator - Microsoft Documentation](#)をしてください。

- `std::cout`はストリームオブジェクトで、`iostream`でされ、`stdout`にされます。

- `<<`はこのでは、オブジェクトをストリームオブジェクトにするためにひされるストリームです。

ライブラリは、のデータにしてストリームにデータをする`<<`をします。 `stream << content content`をストリームにし、じでされたストリームをします。これにより、ストリームのをさせることができます `std::cout << "Foo" << " Bar";` コンソールに "FooBar"をします。

- "Hello World!" リテラル、または "テキストリテラル"です。リテラルのストリームは、ファイル `iostream` されています。

- `std::endl`、ファイルにされたな**II/O**ストリームマニピュレータオブジェクト、ある `iostream`。マニピュレータをストリームにすると、ストリームのがされます。

ストリームマニピュレータ `std::endl`は2つのことをいいますにのをし、ストリームバッファをフラッシュして、コンソールにテキストをさせます。これにより、ストリームにされたデータがにコンソールにされます。ストリームデータは、バッファにされ、

すぐにフラッシュするをいて、バッチで「フラッシュ」されます。

フラッシュをけるのはのとおりです。

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

`\n`は、のためのエスケープシーケンスです。

- 。セミコロン`;`は、ステートメントがしたことをコンパイラにします。すべてのC++ステートメントとクラスには、セミコロン`/`がです。

コメント

コメントは、のテキストをソースコードのにれ、C++コンパイラがそれをする事なくなです。コメントは、プログラムのやをするためにされます。

C++には2のコメントがあります

1 コメント

フォワードスラッシュシーケンス`//`は、がコメントになるまですべてのテキストをマークします

```
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
}
```

C スタイル/ブロックコメント

シーケンス`/*`はコメントブロックのをするためにされ、シーケンス`*/`はコメントのわりをするためにされます。テキストがそれではなC++であっても、シーケンスとシーケンスののすべてのテキストはコメントとしてされます。このコメントはC++のであるCからされているため、これらは「Cスタイル」のコメントとばれることがあります。

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

どのブロックコメントでも、あなたはかをくことができます。コンパイラがシンボル`*/`すると、ブロックコメントをします。

```
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;
}
```

のはなC++およびCコードです。ただし、ブロックコメントのに/*をすると、のコンパイラでがされることがあります。

ブロックコメントは、1でおよびすることもできます。えば

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

コメントの

すべてのプログラミングとに、コメントにはいくつかのがあります。

- みやすく/しやすくするコードのな
- コードのとの
- コードのまたはの
- /ライセンス、プロジェクトノート、な、クレジットなどをソースコードにする。

しかし、コメントにもがあります。

- コードのをするためにそれらをするがあります
- のコメントは、コードをみにくくするがあります

コメントのは、でなコードをくことによってらすことができます。なは、、、およびになをすることです。にするタスクをのにすることは、これとしています。

コードをにするコメントマーカー

にコメントをして、コードのをせずにすばやくにすることもできます。これはテストやデバッグにはですが、なのスタイルにはしていません。これはしばしば「コメントアウト」とばれます。

に、バージョンシステムをしてコードのをべるのとして、ファイルをさせ、をしないため、でコメントのいバージョンをコメントにすことはです。

は、のステートメントをすコードののです。

はまたはをけれ、のをすことができます。をするには、のにびしをし、びしのをりにきえます。

すべてののにシグニチャのとりがあります。

は、きとのからインスピレーションをかけています。

- C++はにきであり、のなやにわない。

はくの、のタスクをすることをしています。プログラムののからびすことができます。は、プログラムのどこかでびされるにされ、されなければなりません。

- なは、のインクルードファイルにされることがあります、くのファイルにされることがい。これはヘッダーファイルのなです。

は、そのとシグネチャをつのをコンパイラにします。はのとおりで。

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

のでは、`int add2(int i)`はコンパイラにしてのようになっています。

- りのは`int`です。
- のは`add2`です。
- へののは1です
 - のは`int`です。
 - のは、ので、`i`ます。

はオプションです。のはのようにもなります。

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

1つのルールでは、のシグネチャをつは、C++コンパイラにされるC++コードベースで1のみまたはできます。つまり、のシグニチャをつをすることはできません。だけするがあります。したがって、はなC++ではありません。

```
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.
```

がもさない、そのりのは`void`としてきされ`void`。パラメータがない、パラメータリストはでなければなりません。

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

びし

はされたにびすことができます。たとえば、のプログラムは、`main`ので`add2`をびします。

```
#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;
}
```

ここで、`add2(2)`はびしのです。

*はとていますが、でがびされたときにされるコードもまれています。

`add2`のは、のようになります。

```
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.
}
```

のオーバーロード

じでなるパラメータをつのをすることができます。

```
int add2(int i)        // Code contained in this definition will be evaluated
{                      // when add2() is called with one parameter.

    int j = i + 2;
    return j;
}

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.
}
```

どちらのも`add2`というじでびされますが、びされるのはびしのパラメータのとにします。ほとんどの、C++コンパイラはびすをできます。によっては、そのタイプをにするがあります。

デフォルトパラメータ

パラメータのデフォルトは、でのみできます。

```
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.
}
```

このでは、1つまたは2つのパラメータをし`multiply()`をびすことができます。パラメータが1つだけされた、`b`デフォルトは7になります。デフォルトのは、ののにするがあります。えば

```
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
```

なびし

C++には、`name_of_function(value1, value2, value3)`とはなるをつなびしがし
`name_of_function(value1, value2, value3)`。もなはのです。

!などのコンパイラによってびしにされるのシーケンス!、+、-、*、%、<<などがあります。これらは、プログラミングのにけられているか、またはにされていますたとえば、+はにC++プログラミングとのでとしてされます。

C++では、これらのシーケンスをなでします。には、のはびしにされます。たとえば、のC++はのとおりです。

```
3+3
```

のびしとじです。

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

すべてののは`operator`まります。

C++ののCでは、C++ではなるシグネチャをつをすることによって、になるをりてることはできませんが、これはです。1つのユニークなののにのをすことは、C++でのオーバーロードとばれ、C++ではですがではありません。

のプロトタイプとの

C++では、にコードをまたはするがあります。たとえば、のようにコンパイルエラーがします。

```
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
{
}
```

これをするには2つのがあります `foo()` または `main()` するにくことです。ここに1つのがあります

```
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.
}
```

しかし、のに "プロトタイプ"だけをいて、をですることによってを ""することもです

```
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
}
```

プロトタイプでは、りの `void`、の `foo`、およびリストのtypes `int` をするがありますが、のはです。

これをソースファイルのにするなの1つは、すべてのプロトタイプをむヘッダーファイルをすることです。

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

のでなをしてください

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

コンパイルされたら、するオブジェクトファイル`foo.o`をコンパイルされたオブジェクトファイルにリンクします。ここで、リンクフェーズでされます`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
```

プロトタイプとびしがしても、がされていないと、"のシンボル"エラーがします。これらは、なリンクまでコンパイラがエラーをしないため、エラーをするためにコードでジャンプするがわからないため、するのがしいがあります。

のC ++コンパイルプロセス

なC ++プログラムコードは、コンパイラによってされます。

コンパイラは、コードをプログラミングからのにするプログラムです。このは、コンピュータでなよりくのです。コンパイラをしてコードをすることをコンパイルとびます。

C ++は、コンパイルプロセスのフォームを ""Cからします。は、C ++でのコンパイルの4つのなステップをすリストです。

1. C ++プリプロセッサは、インクルードされたヘッダファイルのをソースコードファイルにコピーし、マクロコードをし、#defineをとってされたシンボルをそれらのにきえます。
 2. C ++プリプロセッサによってされたされたソースコードファイルは、プラットフォームにしたアセンブリにコンパイルされます。
 3. コンパイラによってされたアセンブラコードは、プラットフォームのなオブジェクトコードにアセンブルされます。
 4. アセンブラによってされたオブジェクトコードファイルは、ファイルをするためにされるライブラリのオブジェクトコードファイルとリンクされています。
- のコンパイルみコードはリンクされていますが、なプログラムはされません。、この「リンクされた」コードは、のプログラムでできるにパッケージすることもできます。この「パッケージされたなコードの」は、C ++プログラマがライブラリとぶものです。

くのC ++コンパイラでは、コンパイルプロセスののをマージまたはアンマージして、にまたはのをうこともできます。くのC ++プログラマはさまざまなツールをしますが、すべてのツールはに、プログラムのにわるときにされたこのプロセスにいます。

のリンクは、このディスカッションをし、つらしいグラフィックをします。 [1] [http :
//faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html](http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html)

プリプロセッサ

プリプロセッサは、 [コンパイラ](#)のなです。

それは、ソースコードをし、いくつかのビットをりり、のものをし、のものをします。

ソースファイルには、プリプロセッサディレクティブをめることができます。これらのディレクティブは、プリプロセッサにのアクションをするようします。ディレクティブはしいのでまりま

す。

```
#define ZERO 0
```

あなたがたすのプリプロセッサディレクティブはおそらく

```
#include <something>
```

。それがをやるかは、 `something` をすべてって、があったファイルにします。 [ここにちはこのプログラムは、](#)

```
#include <iostream>
```

これは、のとをできるようにするとオブジェクトをします。

プリプロセッサもするCでは、C++とじの[ヘッダファイル](#)がありませんが、C++ではすべてのCヘッダファイルができます。

のなはおそらく

```
#define something something_else
```

。これは、ファイルにむいて、 `something` で `something_else` した `something` きえられるべきことをプロセッサにえます。とたようなものにすることもできますが、それはおそらくC++とみなされます。

`something_else` はありませんが、 `something` をもしないと、プリプロセッサのディレクティブのに `something` すべてのがします。

これはにあるためで、です `#if`、`#else` と `#ifdef` ディレクティブ。これらののはのとおりです。

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

#ifdef thing_that_you_want_to_know_if_is_defined
//code
#endif
```

これらのディレクティブは、ビットにあるコードをし、ビットをします。これは、コードをきえることなく、のオペレーティングシステムにのみまれるコードのビットをするためにできます。

[オンラインでC++をいめるをむ](https://riptutorial.com/ja/cplusplus/topic/206/c-plusplus) <https://riptutorial.com/ja/cplusplus/topic/206/c-plusplus> [をいめる](#)

2: 1つのルール ODR

Examples

されたをする

1つのルールのもなは、リンケージをつインラインは、することはできますが、プログラムで1のみすることです。したがって、なるからのヘッダをめることができるため、このようなをヘッダにするべきではありません。

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();
}
```

このプログラムでは、fooはヘッダーfoo.hされています。foo.hはfoo.cppから1、main.cppから2main.cppます。したがって、にはのfooのがまれています。foo.hのインクルードガードは、foo.cppとmain.cppにfoo.h々にmain.cppれているので、これをぐものではないことにしてください。このプログラムをビルドしようとする、fooがされているとするリンクエラーがするがもいです。

このようなエラーをするには、ヘッダーにをし、する.cppファイルでするがありますのを。

インライン

inlineされたは、すべてののがであれば、のすることができます。また、それがされているすべてののですがあります。したがって、インラインはヘッダーにするがあり、ファイルにするはありません。

プログラムは、ののがあるかのようにします。

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();
}
```

このでは、よりな`foo`はヘッダーファイルでインラインでされ、よりな`bar`はインラインではなく、ファイルでされています。 `foo.cpp`と`main.cpp`のには`foo`がまれています、このプログラムは`foo`がインラインであるためです。

クラスでされたメンバまたはフレンドかもしれないはインラインです。したがって、クラスがヘッダにされている、がのにまれていても、そのクラスのメンバをクラスにすることができます。

```
// in foo.h
class Foo {
    void bar() { std::cout << "bar"; }
    void baz();
};

// in foo.cpp
void Foo::baz() {
    // definition
}
```

`Foo::baz`はでされているため、インラインではないので、ヘッダにははいけません。

オーバーロードによるODR

インラインのトークンがじであっても、のがじエンティティをしていない、ODRにするがあります。のように`func`をえてみましょう

- 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)`  
}
```

overloaded. 複数のODRは、同じになるエンティティをしています。

オンラインで1つのルールODRをむ <https://riptutorial.com/ja/cplusplus/topic/4907/1つのルール-odr->

3: 3、5、0のルール

Examples

ファ イブのルール

C ++ 11

C ++ 11では、moveコンストラクタとmoveの2つの新しいメンバがされています。 C ++ 03で [Rule of Three](#) を [る](#) のとじて、はC ++ 11のRule of Fiveにいたいとウクラスが5つのなメンバーのうちの1つをとし、それらのうちのすべてをとするがもい。

ただし、Rule of Fiveにわないことは、エラーとはみなされませんが、3つのルールがききされるり、のをすことになります。コンパイラーがするときコンストラクターまたはができないは、であればコピーセマンティクスをし、なコピーのためにががします。セマンティクスがクラスにしてましくない、コンストラクタまたはをするはありません。

3つのルールのとじ

```
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);
}
};

```

わりに、コピーとものを、コピー・アンド・スワップ・イディオムのをにするためにまたはのわりにでインスタンスをるのできえることもできます。

```

Person& operator=(Person copy)
{
    swap(*this, copy);
    return *this;
}

```

3つのルールから5つのルールへのはパフォーマンスのからですが、ほとんどの、にはではありません。コピーコンストラクタとをすると、をしてもメモリがリークすることはありませんはそのはにコピーにります、びしがしなかったコピーをすることになります。

ルールゼロ

C++ 11

たちは、5つのルールとRAIIのをみわせることで、よりなインターフェースをすることができます。ゼロのルールするがあるリソースはすべてのタイプにするがあります。そのタイプはRule of Fiveにわなければなりません、そのリソースのすべてのユーザーは5つのなメンバーのいずれかをくはなく、それらのすべてをdefaultすることができます。

3つのルールのでしたPersonクラスをして、cstringリソースオブジェクトをできます。

```

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 */
}

```

```
};
```

これがかれば、`Person`クラスはもっとになります

```
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 */
};
```

`Person`のなメンバーは、にするはありません。コンパイラは、`Person`のについてにデフォルトまたはします。したがって、はゼロのルールのもあります。

```
struct Person {
    cstring name;
    int arg;
};
```

`cstring`がmove-onlyで、`delete d`のコピーコンストラクタ/をする、`Person`はにmove-onlyになります。

ゼロのルールというは、[R. Martinho Fernandes](#)

3つのルール

c++ 03

3つのルールでは、にユーザのコピーコンストラクタ、コピー、またはデストラクタがなは、3つすべてをつがあるということです。

ルールのは、3つのリソースのいずれかをとするクラスは、そのリソースをしてするためにいくつかのリソースファイルハンドル、にりてられたメモリなどをするがあるからです。コピーは、オブジェクトがオブジェクトでどのようにコピーされるかをし、デストラクタはRAIIのってリソースをします。

リソースをするをえてみましょう

```
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;
    }
};

```

`name`はコンストラクタに与えられていたため、デストラクタはメモリのリークをけるためにデストラクタをします。しかし、そのようなオブジェクトがコピーされるとどうなりますか

```

int main()
{
    Person p1("foo", 11);
    Person p2 = p1;
}

```

まず、`p1`がされます。に、`p2`が`p1`からコピーされます。しかし、C++でされたコピーコンストラクタは、そのコンポーネントをそのままコピーします。これは、`p1.name`と`p2.name`が同じを指していることをします。

`main`すると、デストラクタが呼びされます。の`p2`のデストラクタが呼びされます。をします。に、`p1`のデストラクタが呼びされます。ただし、はすでにされています。にされたメモリの`delete`を呼びすると、のがします。

これをけるには、なコピーコンストラクタをするがあります。1つのアプローチは、なる`Person`インスタンスが同じデータをするシステムをすることです。コピーがされるたびに、カウントがインクリメントされます。デストラクタはカウントをデクリメントし、カウントがゼロの時にのみメモリをします。

あるいは、[セマンティクスとディープコピー](#)をすることもできます。

```

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;
}

```

コピーのは、のバッファをするがあるためです。コピーとスワップのテクニックは、しいバッフ

アをするオブジェクトをします。スワップ^{*this}および^{copy}してをえる^{copy}のバッファの。 ^{copy}は、がすときに、に^{*this}によってされていたバッファをします。

りて

コピーをくときは、のにくことがにです。つまり、これをするがあります

```
SomeType t = ...;  
t = t;
```

りては、、そのようななではしません。には、さまざまなコードシステムをるをしてわれます。りてのには、2つの^{Person}ポインタまたはがあり、それらがじオブジェクトであることはわかりません。

あなたがいたコピーは、これをにれることができなければなりません。

これをうなは、りてロジックのすべてをのようなでラップすることです。

```
SomeType &operator=(const SomeType &other)  
{  
    if(this != &other)  
    {  
        //Do assignment logic.  
    }  
    return *this;  
}
```

りてについてえることがであり、りてがしたときにコードがしくすることをすることがです。しかし、りてはにまれであり、にのケースをにするのをぐためにされています。のケースがはるかになので、りてのためのペシミングは、コードをにさせるがあります。

として、をするためののは、^{copy and swap idiom}です。このののでは、りてがテストされることはありませんりてはコピーがされるためです。そのは、ののペシミーゼーションはよりにするのではるかにであることがされているからです。

c++ 11

もりてからするがあります。しかし、くのそのようなのロジックは^{std::swap}づいており、これはじメモリとのスワッピングをうまくできます。したがって、りてロジックがのスワップにぎないは、りてのではありません。

そうでないは、とのをずるがあります。

オンラインで3、5、0のルールをむ <https://riptutorial.com/ja/cplusplus/topic/1206/3-5-0のルール>

4: Arithmetic メタプログラミング

き

これらは、コンパイルにアルティティックをするにC++テンプレートメタプログラミングをします。

Examples

Olog nにおけるパワーの

これは、テンプレートメタプログラミングをしてをするなをしています。

```
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;
};
```

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

C++ 14

これものをいます

```
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;
}

```

オンラインでArithmiticメタプログラミングをむ

<https://riptutorial.com/ja/cplusplus/topic/10907/arithmiticメタプログラミング>

5: C++ 11 メモリモデル

なくとも1つのがストアとしてもられているである、じメモリにアクセスしようとするなるスレッドがデータにする。これらのデータは、のをきこします。それらをするには、これらのスレッドがこのようなするをにするのをぐがあります。

プリミティブミューテックス、クリティカルセクションなどは、そのようなアクセスをすることができる。C++ 11でされたメモリモデルは、マルチスレッドでメモリへのアクセスをさせる2つのしいなをしています。アトミックとフェンスです。

アトミックオペレーション

アトミック・ロードおよびアトミック・ストアをすることにより、のメモリーをみきすることがになりました。、これらは`std::atomic<t>`テンプレートクラスにラップされています。このクラスは`t`のをラップしますが、はオブジェクトへのロードとストアはアトミックです。

テンプレートはすべてのタイプでできるわけではありません。なタイプはですが、これには、なまたはすべてのタイプとポインタタイプがまれます。したがって、`std::atomic<unsigned>`
`std::atomic<std::pair<bool, char>>`ほとんどされませんが、`std::atomic<unsigned>`と
`std::atomic<std::vector<foo> *>`はになります。

アトミックには、のプロパティがあります。

- すべてのアトミックは、のをきこすことなく、のスレッドからにできます。
- アトミック・ロードは、アトミック・オブジェクトがされた、またはらかのアトミック・ストアをしてアトミック・オブジェクトにきまれたのいずれかをします。
- じオブジェクトへのアトミックストアは、すべてのスレッドでじになります。あるスレッドがすでにらかのアトミックストアのをている、そののアトムロードでは、じ、またはそののアトミックストアによってされたがされます。
- アトミックなりード・モディファイ・ライト・オペレーションにより、アトミック・ロードとアトミック・ストアは、のアトミック・ストアをさずにできます。たとえば、のスレッドからカウンタをアトミックにインクリメントすることができ、スレッドのになくインクリメントはわれません。
- アトミックは、オプションの`std::memory_order`パラメーターを`std::memory_order`ます。このパラメーターは、がのメモリーにしてどのようなプロパティーをつかをします。

std :: memory_order	
std::memory_order_relaxed	のなし
std::memory_order_release → std::memory_order_acquire	load-acquireがstore-releaseによってされたをした、store-releaseするにシーケンスがstore-releaseされてからload-acquireされたにロードがけされる

std :: memory_order	
std::memory_order_consume	memory_order_acquire ようなもの memory_order_acquire が、 のみです
std::memory_order_acq_rel	load-acquire と store-release みわせる
std::memory_order_seq_cst	

これらのメモリオーダータグでは、シーケンシャル、リラックス、およびリリースでのリリースがされる、3つとなるメモリオーダーリングがです。

シーケンシャル

アトミックにメモリーがされていない、はデフォルトでになります。このモードは、std::memory_order_seq_cst オペレーションをタグけすることでにすることもできます。

このでは、メモリはアトミックをできません。すべてのメモリは、アトミックがアトミックおよびアトミックがするに、アトミックがするにわれ、そのにけられたすべてのメモリがわれます。このモードはおそらくもなものです、パフォーマンスにのをもたらしめます。また、アトミックをえてをべえることをみるのあるすべてのコンパイラーのをします。

リラックスオーダー

シーケンシャルのは、やかなメモリけである。これはstd::memory_order_relaxed タグでされます。されたアトミックは、のメモリにをさない。っているのは、がとしてであることです。

リリースをする

アトミックストアにはstd::memory_order_release タグをstd::memory_order_release ことができ、アトミックロードにはstd::memory_order_acquire タグをけることができます。のはアトムストア・リリースとばれ、2のはアトムロード・アクティブとばれます。

load-acquireがstore-releaseによってきまれたをと、store-releaseがロードのににえるようになるすべてのストアが、ロードにけられたロードにされます。

アトミックリードモディファイライトオペレーションは、タグstd::memory_order_acq_rel もけることができます。これにより、のはのロードをし、ストアのはストアのになります。

コンパイラーは、アトミック・ストア・リリースにストアをすることはできません。また、アトミックなロードまたはロードのに、ロードをすることもできません。

のロード・リリースやアトミック・ストア・アクセスがないことにもしてください。このようなをしようとすると、がされます。

リリース - をする

このみわせは`release-acquire`とていますが、はに `std::memory_order_consume` タグがけられ、 なになります。このモードは、 `load-consume`によってロードされたにじてロードをしたにけられたロードのうち、けられたロードのでは、 `release-acquire`とじです。

フェンス

フェンスはまた、メモリがスレッドでけられることをにする。フェンスはリリースフェンスまたはフェンスをします。

フェンスのにリリースフェンスがした、フェンスのにシーケンスされたロードにリリースフェンスがされるにけられたストアがされます。フェンスのにリリースフェンスがすることをするために、されたをむのプリミティブをすることができ。

Examples

メモリモデルの

```
int x, y;
bool ready = false;

void init()
{
    x = 2;
    y = 3;
    ready = true;
}

void use()
{
    if (ready)
        std::cout << x + y;
}
```

あるスレッドは`init()`をびし、のスレッドまたはシグナルハンドラは`use()`をびします。 `use()`は5か、もしないことをしているかもしれませ。これは、のようなでずしもてはまるとはりませ。

- CPUは、にされるコードがのようになるように、 `init()` でするきみをべえることができす

```
void init()
{
    ready = true;
    x = 2;
    y = 3;
}
```

- CPUは、にされるコードがのようになるように、 `use()` でするみりをべえることが`use()`。

```

void use()
{
    int local_x = x;
    int local_y = y;
    if (ready)
        std::cout << local_x + local_y;
}

```

- C++コンパイラは、のでプログラムをべえることにします。

このようなべえは、スレッドが`init()`と`use()`びしをインターリーブできないため、スレッドでされているプログラムのをすることはできません。、マルチスレッドでは、のスレッドによってされたきみのがえるかもしれませんが、`use()`は`ready==true`、`x`または`y`またはそののガーベッジをることがあります。

C++メモリモデルは、マルチスレッドプログラムがどおりにできるように、プログラマがどのリオーダリングがされているかどうかをすることをにします。のは、スレッドセーフなでのようにきすことができます。

```

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;
}

```

ここで、`init()`はストアのをします。これは、をただでなく、`true`のに`ready`するだけでなく、それはそれのにされるきみのに、このをすることはできませんコンパイラにします。

`use()`は、アトミックなロードをいます。これは`ready`ののをみみ、コンパイラがアトミックなロードをするにシーケンシングされたみりをするをします。

これらのアトミックはまた、コンパイラに、CPUにしてなべえをえるためになハードウェアをくようにします。

アトミック・ストア・リリースはアトミック・ロードとじメモリにあるため、ロード・アクイジション・オペレーションがストア・リリースによってきまれたをすると、`init()`によってされるすべてのきみは、その`store-release`よりのスレッドは、`load use()`のスレッドがロードにするロードにしてになります。つまり、`use()`が`ready==true`れば、`x==2`と`y==3`します。

コンパイラとCPUは、まだ`x`にきむに`y`にきむことがされています。に、`use()`これらのからのみみは、のでできます。

フェンスの

のはフェンスとやかなですることもできます

```
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;
    }
}
```

アトミック・ロード・オペレーションがアトミック・ストアによってきまれたをと、ストアはロードのにし、フェンスもです。リリース・フェンスは、フェンスのにし、リリース・フェンスがされるの_xおよび_yフェンスにく `std::cout` ステートメントに `std::cout` ます。

フェンスは、、、またはそののをらすことができればです。えば

```
void block_and_use()
{
    while (!ready.load(std::memory_order_relaxed))
        ;
    atomic_thread_fence(std::memory_order_acquire);
    std::cout << x + y;
}
```

`block_and_use()` は、やかなのけをりて `ready` フラグがセットされるまでします。に、のフェンスをして、なメモリをする。

オンラインでC ++ 11メモリモデルをむ <https://riptutorial.com/ja/cplusplus/topic/7975/c-plusplus-11メモリモデル>

6: C ++ コンテナ

き

C ++ コンテナは、のコレクションをします。テンプレートには、プリミティブintなどやカスタムクラスMyClassなどのコレクションが含まれています。

Examples

C ++ コンテナのフローチャート

どのC ++ コンテナをするかをするのはいいかもしれないので、ここでは、コンテナがそのジョブにしているかどうかをするためのなフローチャートを行います。

グラフィックは、 [Megan Hopkins](#)のものです

オンラインでC ++コンテナをむ <https://riptutorial.com/ja/cplusplus/topic/10848/c-plusplus> コンテナ

7: C++ ストリーム

`std::istream_iterator` デフォルトコンストラクタは、ストリームのわりをすイテレータをします。したがって、`std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(), ...)` は、`ifs` ののからにコピーすることをします。

Examples

ストリングストリーム

`std::ostringstream` は、オブジェクトがストリームのようにえるクラスですつまり、`operator<<` をしてオブジェクトにきむことはできますがにきみをし、ストリームのでします。

のいコードをえてみましょう。

```
#include <sstream>
#include <string>

using namespace std;

int main()
{
    ostringstream ss;
    ss << "the answer to everything is " << 42;
    const string result = ss.str();
}
```

この

```
ostringstream ss;
```

そのようなオブジェクトをします。このオブジェクトは、にのストリームのようにされます。

```
ss << "the answer to everything is " << 42;
```

それにもかかわらず、のストリームはのようにないます。

```
const string result = ss.str();
```

の`result` は "the answer to everything is 42" としくなります。

これはに、ストリームのシリアルがされているクラスがあり、フォームがなにです。たとえば、あるクラスがあるとします

```
class foo
{
```

```
// All sort of stuff here.
};

ostream &operator<<(ostream &os, const foo &f);
```

`foo` オブジェクトのをするには、

```
foo f;
```

々はすることが出来ます

```
ostringstream ss;
ss << f;
const string result = ss.str();
```

`result` には、 `foo` オブジェクトのがまれます。

までファイルをむ

でテキストファイルをむ

`ifstream` ドキュメントでは、テキストファイルをまでみむなはです。のC++プログラマによる
ないや、ファイルをむなをえてみましょう。

のない

わかりやすくするために、ファイルのにがまれていないとします。

`ifstream` は `operator bool()` があり、ストリームにエラーがなくみりなになったときに `true` をします。
。さらに、 `ifstream::operator >>` がストリームへのをすので、にされたでEOFエラーだけでなく
をみんでチェックすることができます

```
std::ifstream ifs("1.txt");
std::string s;
while(ifs >> s) {
    std::cout << s << std::endl;
}
```

をむ

`ifstream::operator >>` は、がれるまでストリームをみみます。したがって、のコードでは、のを々のにします。までをすべてみむには、 `ifstream::operator >>` わりに `std::getline` します。 `getline` は、
、しているスレッドへのをします。したがって、じをできます。

```
while(std::getline(ifs, s)) {
```

```
std::cout << s << std::endl;
}
```

らかに、`std::getline`は、までのファイルを読み込むためにも使われるべきです。

にファイルをバッファに読み込む

に、やなどのことをめくことなく、からまでファイルを読み込みましょう。なファイルサイズまたはそのがされることになっているのは、のサイズをしてから読み込むことができます。

```
s.resize(100);
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
          s.begin());
```

それのは、をのにするがあります。したがって、`std::back_inserter`がです。

```
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
          std::back_inserter(s));
```

あるいは、イテレータのをつコンストラクタをして、ストリームデータをつコレクションをすることもできます。

```
std::vector v(std::istreambuf_iterator<char>(ifs),
              std::istreambuf_iterator<char>());
```

これらのは、`ifs`がバイナリファイルとしてかかれているにもできます。

```
std::ifstream ifs("1.txt", std::ios::binary);
```

ストリームのコピー

ファイルはストリームとイテレータでのファイルにコピーすることができます

```
std::ofstream ofs("out.file");
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
          std::ostream_iterator<char>(ofs));
ofs.close();
```

またはのあるインタフェースをつのタイプのストリームにリダイレクトされます。たとえば Boost.Asio ネットワークストリーム

```
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();
```

イテレータはポインタのとしてえることができるので、ののSTLコンテナはネイティブにきえることができます。をにするはのとおりで。

```
int arr[100];
std::copy(std::istream_iterator<char>(ifs), std::istream_iterator<char>(), arr);
```

がりてられたにオンザフライでサイズすることはできないため、バッファオーバーフローにしてください。たとえば、のコードに100のがまれるファイルがされた、にきしてのにしようとし

iostreamでコレクションをする

な

`std::ostream_iterator`すると、STLコンテナのをなループなしでのストリームにできます。
`std::ostream_iterator`コンストラクタの2のはりをします。たとえば、のコード

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " ! "));
```

する

```
1 ! 2 ! 3 ! 4 !
```

のキャスト

`std::ostream_iterator`は、コンテナのコンテンツタイプをにキャストできます。たとえば、のに3のをするように`std::cout`をしましょう。

```
std::cout << std::setprecision(3);
std::fixed(std::cout);
```

されたは`int`ままですが、`float`で`std::ostream_iterator`をインスタンスします。

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<float>(std::cout, " ! "));
```

のコードはをます

```
1.000 ! 2.000 ! 3.000 ! 4.000 !
```

`std::vector`にもかかわらず`int`がされます。

と

`std::generate`、`std::generate_n`および`std::transform`は、オンザフライでのデータのためのツールをします。例えば、ベクトルを

```
std::vector<int> v = {1,2,3,4,8,16};
```

の "x is even" のブールをにすることができます

```
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;
});
```

またはされたをする

```
std::transform(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

スペースでられたNのをする

```
const int N = 10;
std::generate_n(std::ostream_iterator<int>(std::cout, " "), N, std::rand);
```

テキストファイルをむセクションのように、これらののほとんどすべてがネイティブにされるがあります。たとえば、ネイティブからされたをします。

```
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;
});
```

ファイルの

STL コンテナにファイルをする

`istream_iterator` は、コードのなループなしで、やそののなデータのシーケンスを STL コンテナにみむのにです。

なコンテナサイズの

```
std::vector<int> v(100);
std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
          v.begin());
```

またはイテレータをして

```
std::vector<int> v;
std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
          std::back_inserter(v));
```

ファイルのは、ののとでできることにしてください。

テキスト テーブルの

`istream::operator>>`はまでテキストをみむので、なデータテーブルをするために`while`でできます。たとえば、2つのとそれになしがあるファイルがあるとします。

```
1.12 3.14 foo
2.1 2.2 barr
```

のようにすることができます

```
std::string s;
double a, b;
while(ifs >> a >> b >> s) {
    std::cout << a << " " << b << " " << s << std::endl;
}
```

は、`std::istream_iterator`でできます。それらのうちの1つは`std::transform`、データをオンザフライですることができます。たとえば、をみみ、それらに3.14をかけて、をコンテナにします。

```
std::vector<double> v(100);
std::transform(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
               v.begin(),
               [](int val) {
                   return val * 3.14;
               });
```

オンラインでC++ストリームをむ <https://riptutorial.com/ja/cplusplus/topic/7660/cplusplusストリーム>

8: C++でのデータ

Examples

C++でのリンクリストの

リストノードの

```
class listNode
{
    public:
    int data;
    listNode *next;
    listNode(int val):data(val),next(NULL){}
};
```

リストクラスの

```
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();
};
```

リストのにしいノードをする

```
void List::insertAtBegin(int val)//inserting at front of list
{
    listNode *newnode = new listNode(val);
    newnode->next=this->head;
    this->head=newnode;
}
```

リストのにしいノードをする

```
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;
}

```

リストののにする

```

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;
}

```

リストからノードをする

```

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);
}

```

リストをする


```
void List::print()//printing the list
{
    listNode *ptr=this->head;
    while(ptr!=NULL)
    {
        cout<<ptr->data<<" ";
        ptr=ptr->next;
    }
    cout<<endl;
}
```

リストのデストラクタ

```
List::~~List()
{
    listNode *ptr=this->head, *next=NULL;
    while(ptr!=NULL)
    {
        next=ptr->next;
        delete(ptr);
        ptr=next;
    }
}
```

オンラインでC++でのデータをむ <https://riptutorial.com/ja/cplusplus/topic/7485/c-plusplusでのデータ>

9: C++でのデザインパターンの

き

このページでは、デザインパターンがC++でどのようにされているかを見つけることができます。これらのパターンのについては、[デザインパターンのドキュメント](#)をしてください。

デザインパターンは、ソフトウェアのコンテキストでにするにするななソリューションです。

Examples

オブザーバーパターン

Observer Patternのは、オブジェクトで1のをして、1つのオブジェクトがをしたときにそのすべてのオブジェクトがにおよびされるようにすることです。

とオブザーバーは、のをします。オブザーバーは、のがするとオブザーバーにされるように、にしています。にじて、オブザーバーはしいでされることもあります。

ここでは、ガンマによるの「デザインパターン」のをします。

```
#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);
}

```

```

Digital time is 14:41:36
Analog time is 14:41:36

```

パターンのはのとおりで。

1. オブジェクト `DigitalClock` または `AnalogClock` オブジェクトテーマインタフェースを `Attach()` または `Detach()` のいずれかをオブザーバーとして、またはであるからする
`subject.Attach(*this);`、 `subject.Detach(*this);`
2. はくのオブザーバー `vector<Observer*> observers;` をつことができます。
3. すべてのオブザーバは `Observer` インタフェースをするがあります。このインタフェースには、のがされたときにびされる1つのメソッド `Update()` があります `Update(Subject &)`
4. `Attach()` メソッドと `Detach()` メソッドにえて、なサブジェクトは、がわるたびにのオブザーバをすべてするための `Notify()` メソッドをしています。しかし、この、それらのすべては、クラス `Subject` `Subject::Attach(Observer&)`、 `void Subject::Detach(Observer&)` および `void Subject::Notify()` ます。
5. Concrete オブジェクトには、そのをしするメソッドもあります。
6. なオブザーバは、 `Observer` インターフェイスをするのクラスです。オブザーバーは、をするなサブジェクト `subject.Attach(*this);` をサブスクライブし `subject.Attach(*this);`。
7. `Observer Pattern` の2つのオブジェクトはやかにされていますが、はできますが、おいのはほとんどありません。

とスロット

Signals and slotsはQtでされたであり、コードをけながらObserverパターンのをにします。は、コントロールウィジェットとしてもられているは、スロットとばれるなをしてのコントロールでできるイベントをむをできるということです。Qtのスロットは、そのようにされたクラスメンバでなければなりません。/スロットシステムは、グラフィカルユーザインタフェースのにしています。に、シグナル/スロットシステムは、ソケット、パイプ、シリアルデバイスなどをむイベント、またはタイムアウトイベントをなオブジェクトインスタンスおよびメソッドまたはにけるためにできます。QtのMeta Object CompilerMOCはなインフラストラクチャをにするため、//びしコードをするはありません。

Cでも、のがサポートされていますが、なるとがされています。イベントはシグナルのをたし、はスロットです。さらに、デリゲートはポインタのようにローカルになりますが、Qtのスロットはそのようにされたクラスメンバでなければなりません。

アダプタパターン

クラスのインターフェイスを、クライアントがするのインターフェイスにします。AdapterまたはWrapperをすると、のないインタフェースがでクラスができなくなります。アダプタパターンのけは、インタフェースをできるのにのソフトウェアをできるということです。

1. アダプタパターンは、オブジェクトのにします。
2. クライアントはAdapterオブジェクトにしてをびします。
3. アダプタはAdapteeをびしてをします。
4. STLでは、vectorからするスタックstackがpushをすると、にあるベクトルはvector :: push_backをします。

```
#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)

```

コードの

1. クライアントは、が `Rectangle` としているとえている
2. ターゲットは `Rectangle` クラスです。これは、クライアントがメソッドをびすためのものです。

```

Rectangle *r = new RectangleAdapter(x,y,w,h);
r->draw();

```

3. アダプタクラスはをすることにしてください。

```

class RectangleAdapter: public Rectangle, private LegacyRectangle {
    ...
}

```

4. Adapter `RectangleAdapter` すると、 `LegacyRectangle` は、のクラスをして、 `Rectangle` の `draw()` にします。
5. `LegacyRectangle` クラスは `Rectangle` とじメソッド `draw()` をちませんが、
Adapter (`RectangleAdapter`)

はRectangleメソッドびしをLegacyRectangle、oldDraw()、oldDraw()メソッドをびしてびします。

```
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();
    }
};
```

アダプタパターンは、あるクラスのインタフェースをのあるなるインタフェースにします。したがって、これはコンポーネントラッパーであるでプロキシパターンにしています。しかし、アダプタクラスとのクラスのインタフェースはなるがあります。

のでたように、このアダプターパターンは、のAPIにしてのインターフェースをしてのコードとのをにするのにです。また、アダプターパターンをすることで、インターフェースをして、したAPIをするようにすることができます。

ブリッジ・パターンはオブジェクト・アダプターとたですが、ブリッジはなるをとっています。インタフェースをからして、にしてできるようにします。アダプターは、のオブジェクトのインターフェースをするためのものです。

パターン

ファクトリパターンはオブジェクトをりし、インタフェースをしてですることができます。

```
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;
    }
};

```

Fluent APIをしたBuilderパターン

Builderパターンは、オブジェクトのをオブジェクトからりします。なアイデアは、オブジェクトがのにをうはないということです。なオブジェクトのしいアセンブリとなアセンブリは、なであるがあるため、このタスクをのクラスにすることができます。

CのEmail Builderにされて、はここでC ++バージョンをすることにめました。 Emailオブジェクトは、ずしもになオブジェクトではありませんが、パターンをすことができます。

```

#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;
}
```

以前のC++では、`std::move`オペレーションをし、オペレータから`&&`をすることができます。ただし、コピーがされます。

ビルダーは、`operator Email&&()`によってビルドされたメールをすると、をします。このでは、ビルダはオブジェクトであり、されるにメールをします。また、のわりに`Email`
`EmailBuilder::build() {...}`ようななをすることもできます。

ビルダーをりにす

Builderパターンがするらしいは、のアクタをしてオブジェクトをにするです。これは、ビルダをのアクタにすことによってわれます。これらのアクタは、それぞれがビルドされたオブジェクトにさらにをえます。これは、らかのクエリをしてフィルタやそののをするににです。

```
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;
}
```

デザインのオブジェクト

このパターンのデザインをニーズにわせてすることができます。は1つのをえるでしょう。

えられたでは、Emailオブジェクトはです。つまり、そのオブジェクトへのアクセスがないため、プロパティをすることはできません。これはましいでした。オブジェクトのにオブジェクトをするがあるは、そのオブジェクトにいくつかのセッターをするがあります。これらのセッターはビルダーにされるので、1つのクラスですべてのをうことをすることができますビルダークラスはもうありません。それにもかかわらず、は、にみみオブジェクトをにするをする。

オンラインでC++でのデザインパターンのをむ [https://riptutorial.com/ja/cplusplus/topic/4335/c-
plusplusでのデザインパターンの](https://riptutorial.com/ja/cplusplus/topic/4335/cplusplusでのデザインパターンの)

10: C ++でのユニットテスト

き

テストは、コードのとをするソフトウェアテストのレベルです。

C ++では、「コードの」はしばしばクラス、またはそのいずれかのグループをします。テストは、しばしばなやパターンをするな「テストフレームワーク」や「テストライブラリ」をしてされます。

このトピックでは、さまざまなとテストのまたはフレームワークについてします。

Examples

Googleテスト

Google Testは、GoogleがするC ++テストフレームワークです。 `gtest` ライブラリをビルドし、テストケースファイルをビルドするときにテストフレームワークにリンクするがあります。

の

```
// 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
```

キャッチ

Catchは、TDDとBDDのユニットテストスタイルをできるヘッダーのみのライブラリです。

のスニペットは、このリンクの Catchドキュメントページからです。

```
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 );
            }
        }
    }
}
```

に、これらのテストはにのようにされます

```
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
```

オンラインでC++でのユニットテストをむ <https://riptutorial.com/ja/cplusplus/topic/9928/cplusplusでのユニットテスト>

11: C++での

Examples

Fibonnaciシーケンスをくためのテールと**Fibonnaci**スタイルのの

をってFibonnaciシーケンスのNのをめるでもなは、これです

```
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);
}
```

しかし、このアルゴリズムは、よりいのためにスケールしないどんどんきくするために n 、のは、あなたがにさせるがあることをびします。これはなテールときえることができます。

```
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);
}
```

びしのたびに、Fibonnaciシーケンスののがすぐにされるため、びしのは n にします。

メモとの

はにになるがあります。それらがなじでびされたときににじをし、にしたりしないである、すでにされたをすることによってメモリをにしてかなりにすることができます。

は、メモによるフィボナッチシーケンスのです

```
#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;
}
}

```

なをしているにもかかわらず、のびしでこのは\$ On\$です。そののじのびしでは、もちろん\$ O1\$です。

ただし、このはリエントラントではありません。また、されたをりくこともできません。のでは、マップをのとしてすことができます。

```

#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;
    }
}

```

このバージョンでは、びしはされたでマップをするがあります。これにより、がリエントラントになり、びしがになったをしてメモリをできるというがあります。それはカプセルをとというがあります。びしはマップになをしてをすることができます。

オンラインでC ++でのをむ <https://riptutorial.com/ja/cplusplus/topic/5693/c-plusplusでの>

12: C++での

C++はセットをしません。また、UTF-8やUTF-16のをサポートするコンパイラもあります。しかし、なANSI / ASCIIのものがされることはではありません。

したがって、すべてのサポートはされており、しているプラットフォーム、オペレーティングシステム、コンパイラにしています。

プラットフォームのなサポートをするためにできるいくつかのUnicodeなど。

Examples

C++について

```
#include <iostream>
#include <string>

int main()
{
    const char * C_String = "This is a line of text w";
    const char * C_Problem_String = "This is a line of text \u005C";
    std::string Std_String("This is a second line of text w");
    std::string Std_Problem_String("This is a second line of \u005C\u005C \u005C");

    std::cout << "String Length: " << Std_String.length() << '\n';
    std::cout << "String Length: " << Std_Problem_String.length() << '\n';

    std::cout << "CString Length: " << strlen(C_String) << '\n';
    std::cout << "CString Length: " << strlen(C_Problem_String) << '\n';
    return 0;
}
```

プラットフォームWindows、OSXなどおよびコンパイラGCC、MSVCなどによっては、このプログラムがコンパイル、なるの、またはじのにすることがあります。

Microsoft MSVCコンパイラの

のさ 31

のさ 31

CStringのさ 24

CStringのさ 24

これは、MSVCでは、されるがの「」とみなされ、このプラットフォームはをにサポートすることを行っています。

しかし、このではないことにしてください。これらはUnicodeとしてにされているため、にはバイトのさです。しないエラーがすることがあります

GNC / GCCコンパイラでは、プログラムのようになります。

のさ31

のさ36

CStringのさ24

CStringのさ26

これは、このLinuxプラットフォームでされているGCCコンパイラがこれらのをサポートしているが、々のをするために しくいくつかのバイトもすることをしている。

この、Unicodeのがですが、このシナリオでの "" のさはみみなのはではなくバイトであることをえておくがあります。

これらのいは、ながプラットフォームごとにどのようにされるか、さらになことですが、このでされるCおよびC ++はバイトとなすことができるため、C ++では**char**を1バイトにする。

オンラインでC ++でのをむ <https://riptutorial.com/ja/cplusplus/topic/5270/c-plusplusでの>

13: C++での

ライブラリ `<iostream>` は、のストリームをほとんどしていません。

stream	description
cin	standard input stream
cout	standard output stream
cerr	standard error (output) stream
clog	standard logging (output) stream

の4つのストリームのうち、`cin` はにユーザーにされ、の3つはデータをするためにされます。またはほとんどのコーディングでは、`cin` コンソールまたははキーボードであり、`cout` コンソールまたははモニターです。

```
cin >> value

cin    - input stream
'>>'  - extraction operator
value - variable (destination)
```

ここの`cin` は、ユーザがしたをし、のでします。ユーザーがENTERキーをしたにのみがされます。

```
cout << "Enter a value: "

cout          - output stream
'<<'        - insertion operator
"Enter a value: " - string to be displayed
```

`cout` はされるをけり、それをまたはモニタにします

4つのストリームはすべての`std`にありますので、ストリーム`stream`をするために`std::stream`をするがあります。

コードにはマニピュレータ`std::endl`もあります。ストリームでのみできます。それはストリームのに`\n`をし、それをフラッシュします。にがされます。

Examples

ユーザーと

```
#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;  
}
```

オンラインでC++でのをむ <https://riptutorial.com/ja/cplusplus/topic/10683/c-plusplusでの>

14: C ++での

Examples

のクラスの

オブジェクトは1バイトをめることはできません。したがって、こののメンバーはアドレスをちます。したがって、`sizeof(T)>=1`にする。また、したクラスがそのクラスのどれよりもさくすることはできません。ただし、クラスがの、そのサイズはずしもクラスにされるわけではありません。

```
class Base {};  
  
class Derived : public Base  
{  
public:  
    int i;  
};
```

この、オブジェクトのタイプごとにアドレスをつために、`Derived`の`Base`バイトをりてるはありません。のクラスのがわれパディングは、`sizeof(Derived) == sizeof(int)`、つまりのベースにしてのりてはわれません。これはのクラスでもですC ++では、のがじをつことができないので、それからししません。

これは、`Derived`ののメンバーのがクラスのいずれかとなるにのみできます。これには、またはながまれます。ベースの1つとじタイプまたはのベースがあるの、じタイプのなる2つのオブジェクトがじアドレスをたないようにするには、なくとも1バイトをりてるがあります。

パフォーマンスの

CおよびC ++はとしてよくられています。に、コードのカスタマイズがであるため、ユーザーはのによってパフォーマンスをできます。

するは、するコードをベンチマークし、コードのをにすることがです。

なミスはのとおりでです。

- のなコードは、にするがあり、とをします。まずに、されたコードではなく、でなコードをくことです。
- ったユースケースの1のオーバーヘッドをすることは、のg9
- マイコンパイラはこれをにいい、マイクロはコンパイラがコードをさらにするをなうことさえあります

なはのとおりでです。

-

をなくする

- よりなアルゴリズム/をするには
- ハードウェアをよりにするには

されたコードには、をむマイナスのがあります。

- メモリの
- なコード - みみやがしい
- APIとコードデザインの

よりないコードで

のもなアプローチは、コードをなくすることです。このアプローチは、、コードのさをする事なく、のスピードアップをします。

このアプローチはなスピードアップをしますが、これはコードがくびされるにのみなをもたらします。

にたないコードをする

```
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から、コンパイラは、このコードをして、りりとするりりをりくことができます。

コードを1だけする

```
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();
}

```

このにするのアプローチをして、した`unique`なバージョンをすることができます

```

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;
}

```

なりてやコピー/の

のでは、すでに`std::set`のルックアップをいでしたが、`std::vector`まだストレージをりてするがあるアルゴリズムがえています。これは、にしいサイズにすることでぐことができます。

```

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;
}

```

なコンテナの

になデータをすることですること、コードのさをえることができます。

```

// 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;
}
```

なるをするコンテナツリーではなくハッシュコンテナをすることで、をNにできます。として、`std::string`のをlessされたがじバケットにわるはずです。

```
// 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;
}
```

さなオブジェクトの

さなオブジェクトのは、えは`std::string`にはShort / Small String Optimizationとばれるのような、レベルのデータでされるです。これは、コンテンツがみのスペースにまるのにさいには、りてられたメモリのわりにスタックスペースをバッファとしてすることをします。

なメモリオーバーヘッドとなをすることにより、なヒープりてをしようします。こののは、にし、ってされたでもパフォーマンスをなうがあります。

このをしてをするになは、のとおりです。

```
#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 = &_amp;smallBuffer[0];
        }
    }
    // Other methods, including other constructors, copy constructor,
    // assignment operators have been omitted for readability
};

```

のコードでかるように、`new`や`delete`をぐために、いくつかのさがされています。これにえて、クラスには、2つのをいてされないのである、よりきなメモリフットプリントがあります。

くの、`_buffer`ポインタの`bool _isAllocated`をビットで`_buffer`して、のインスタンスのサイズをしようします `_buffer`ビットサイズを8バイトさくすることができます。は、プラットフォームのアラインメントルールがわかっているにのみです。

いっしますか

このではくのさがわるため、このをすべてのクラスですることはおめしません。にされるレベルのデータでしばしばするでしょう。なC++ 11 standard libraryでは、`std::basic_string<>`および`std::function<>`をつけることができます。

このは、されたデータがバッファよりもさいにのみメモリりてをするため、クラスがさなデータでよくされるにのみメリットをもたらします。

こののなは、バッファをするときにながとなり、バッファがされないよりもがになることである

。これは、バッファにPODタイプがまれているににてはまります。

オンラインでC ++でのをむ <https://riptutorial.com/ja/cplusplus/topic/4474/c-plusplusでの>

15: C ++でののビヘイビアー

き

C ++がどうやってっていけるかのをもっとる

のからの

Examples

リストのメンバーをする

コンストラクタがをするにイニシャライザリストのメンバをすると、のがするがあります。このですべてのメンバーがされているわけではないため、このになります。から

§12.7.1コンストラクタがをするにオブジェクトのメンバまたはクラスをすることで、のコンストラクタをつオブジェクトの、のがします。

```
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
    }
};
```

オンラインでC ++でののビヘイビアーをむ [https://riptutorial.com/ja/cplusplus/topic/9885/c-
plusplusでののビヘイビアー](https://riptutorial.com/ja/cplusplus/topic/9885/cplusplusでののビヘイビアー)

16: C++ デバッグとデバッグツールとテクニック

き

C++のくのは、デバッグにやされます。このトピックはこのタスクをし、テクニックのインスピレーションをえるものです。のツールにするツールやマニュアルでされたやのリストをしないでください。

このトピックはまだではありませんが、のテクニック/ツールのがにちます

- よりくのツールの
- バイナリツールUBSan、TSan、MSan、ESan ...など
- CFI ...
- ファジー

Examples

のC++プログラムはsegfault-valgrindでわかります

なプログラムを試みましょう

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p3 << std::endl;
    }
}

int main() {
    fail();
}
```

ビルドするデバッグをめる-gをする

```
g++ -g -o main main.cpp
```

```
$ ./main
Segmentation fault (core dumped)
$
```

valgrindでデバッグしましょう

```
$ 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== 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)
$
```

まずこのブロックにをてます

```
==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
```

のは、segfaultが4バイトをみることによってしていることをしています。2と3はコールスタックです。これは、main.cppのmain、line 13でびされるmain.cppの8のfail() でなみりがわれたことをします。

main.cppの8をと

```
std::cout << *p3 << std::endl;
```

しかし、にポインタをチェックするので、がなのですかのブロックをチェックします

```
==8515== Conditional jump or move depends on uninitialised value(s)
==8515==    at 0x400813: fail() (main.cpp:7)
==8515==    by 0x40083F: main (main.cpp:13)
```

それは7にユニットされたがあり、それをんでいることがわかります

```
if (p3) {
```

これは、p2のわりにp3をべるをしています。しかし、p3はどのようにされていないがありますか
する

```
int *p3 = p1;
```

Valgrindは`--track-origins=yes`ですようアドバイスします。それをやりましょう

```
valgrind --track-origins=yes ./main
```

valgrindのはvalgrindのです。たちのプログラムのにくと、プログラムにされます。

はほぼじですが、いは1つだけです

```
==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)
```

これは、7でしたされていないが3でされたことをしています。

```
int *p1;
```

されていないポインタにいてくれます。

GDBによるSegfault

このでは、とじコードをします。

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p2 << std::endl;
```

```

    }
}

int main() {
    fail();
}

```

まずコンパイルします

```
g++ -g -o main main.cpp
```

gdbでできるようにする

```
gdb ./main
```

はgdbシェルになります。をします。

```

(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;

```

セグメンテーションフォルトが11で起っているのがわかります。このでされるのはポインタp2です。のをべることが出来ます。

```

(gdb) print p2
$1 = (int *) 0x0

```

は、p2がNULLをす0x0にされていることがわかります。このでは、NULLポインタをしようとしていることがわかります。だから々はそれをします。

クリーンコード

デバッグは、デバッグしようとしているコードをすることからまります。

いコード

```

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; }

```

よりいいコード

```

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;
}

```

したコーディングおよびスタイルをつことで、コードをしやすくなります。

のコードをと、とデバッグをさせるためのいくつかのをできます。

々のアクションのための々のの

々のをすると、にがないは、デバッグののをスキップすることができます。こののケースでは、データのやにはがなく、ソートだけにんでください。

もうつののは、コードをんでいくうちに、よりないコードをむそしてするがあることです。 `main()` の3のコードをむのは、ではなく、するためだけです。

3のは、るコードがなくてみ、このバグをでするのにされたをけることです。

したIの

したとののは、コードからをりき、テキストではなくコードにするのをにします。「しい」についてくのがわれました。そのスタイルにかかわらず、コードにのしたスタイルをたせることで、しみやすさがし、コードにするのがになります。

コードはのかかるであるため、のツールをすることをおめします。ほとんどのIDEはこれになくともらかのサポートをしており、よりものあるフォーマットをうことができます。

スタイルがやにされているわけではなく、フリースタイルとメンバをさせてコンテナのIをするはありません。 `v.begin()` VS `std::end(v)`。

コードのなにをける。

するスタイルにかかわらず、のコードには、がかをすヒントをえるマーカーがいくつかまれています。

- `optimized`たコメント
- `sortVector()` ののは、たちがなことをしていることをしています
- `std::ref()` は、 `sortVector()` でかがこっていることをします。

クリーンなコードがあれば、コードのにち、デバッグにながされます。2のでは、コードレビューがにしてバグをするかもしれませんが、のバグはにれているかもしれません。PSバグは₂としています

とは、のバグにリンクされたパターンをチェックするです。このをするとコードレビューよりもががかりませんが、チェックはツールでプログラムされたものにされます。

チェックには、がされていないかどうかをするなグラフィアルゴリズムまで、`if`のろにあるなセミコロ `if (var);` をめることができます。

コンパイラの

をにするのはですが、もシンプルなバージョンはにコンパイラにみまれています。

- `clang++ -Wall -Weverything -Werror ...`
- `g++ -Wall -Weverything -Werror ...`
- `cl.exe /W4 /WX ...`

これらのオプションをにすると、コンパイラはのがしていないバグをし、のコンテキストであるかであるかもしれないテクニックにりをすことにづくでしょう。 `while (staticAtomicBool);` `while (localBool);`でもけられるかもしれませんが `while (localBool);`そうではありません。

コードレビューとはって、コードをし、なバグをたくさんえてくれるツールとっていることがあります。こののケースでは、ローカルでをするがあります。

のオプションはすべてのをにするので、ましくないをにするがあります。なぜあなたのコードはC++ 98とがありますかもしそうなら、そののをににすることができます

- `clang++ -Wall -Weverything -Werror -Wno-error-to-accept ...`
- `g++ -Wall -Weverything -Werror -Wno-error-to-accept ...`
- `cl.exe /W4 /WX /wd<no of warning>...`

コンパイラのがにところでは、コンパイルがかなりくなります。そのため、デフォルトではににするとはいりません。デフォルトでそれらをするか、よりなチェックまたはそのすべてとのなをにします。

ツール

あなたがらかのなをうことにめた、のツールのはそんなにしくありません。[clang-tidy](#)のようなツールには、いをカバーする[チェックリスト](#)があります。いくつかのがあります

- のバグ
 - スライスの
 - をうアサート
- チェック
 - をくみ
 - のをする
- チェック
 - `make_unique`をする
 - `nullptr`をする
- パフォーマンスチェック
 - なコピーをす
 - なアルゴリズムびしをつける

Clangはすでにくのコンパイラのをけているので、リストはそれほどきくないかもしれませんが、のコードベースにづくでしょう。

そののツール

のをつのツールがします

- ツールとしての[ビジュアルスタジオスタティックアナライザ](#)
- [clazy](#)、QtコードをチェックするためのClangコンパイラプラグイン

C++のためのくのツールがし、どちらもツールとしてコンパイラにみまれています。それらをしてみると、なセットアップのためにそれほどがからず、コードレビューですのあるバグがつかるでしょう。

セーフスタックスタック

スタックのは、てなバグです。スタックがれているので、デバッガは、スタックとスタックをスタックトレースできないことがよくあります。

これはスタックがするです。スレッドに1つのスタックをするわりに、セーフスタックとなスタックの2つをします。スタックは、とじようにしますが、のはスタックにします。

スタックのどのがするのですか

スタックをするのあるすべてののがセーフスタックからします。スタックのがしされるか、またはがこののアドレスをけるとすぐに、コンパイラはこれをなスタックのわりに2のスタックにりてることをします。

として、これらのポインタでうこれらのポインタ/についてメモリにえたは、2のスタックのメモリにのみします。セーフスタックにいポインタをることはないので、スタックはスタックをすることができず、デバッガはスタックのすべてのをみみ、らしいトレースをえることができます。

それはにのためにわれていますか

なスタックは、よりいデバッグをするためにされたものではありませんでしたが、なバグにするいです。それは[コードインジェクタのCPIプロジェクト](#)のとして、コードインジェクションをぐためにリターンアドレスのオーバーライドをぎます。いえれば、らはハッカーコードをしないようにしようとしています。

このため、このはクロムでされ、CPUオーバーヘッドが1であるとされています。

それをにする

のところ、このオプションは[clangコンパイラ](#)でしかできません。この[コンパイラ](#)では、`-fsanitize=safe-stack`をコンパイラにすことができ`-fsanitize=safe-stack`。GCCのじをするがなされました。

スタックのは、スタックがになっているにデバッグするのがになります。パフォーマンスのオーバーヘッドがいたため、ビルドではデフォルトでにすることもできます。

オンラインでC++デバッグとデバッグツールとテクニックをむ

<https://riptutorial.com/ja/cplusplus/topic/9814/c-plusplusデバッグとデバッグツールとテクニック>

17: C ++ "value by call" と "reference by call"

き

このセクションのは、びしにのパラメータがどうなるかについてののいをすることです。

には、パラメータは、びしのとのでしてることができます。ここでは、これらのへのなとアクセスは、それらをすメソッドによってなります。

さらに、このトピックでは、びしのとそのそれぞれののについてもします。

Examples

によるびかけ

をびすと、プログラムスタックにしいがされます。これには、にするいくつかのと、パラメーターとりのスペースメモリーもまれています。

パラメータをにすときは、されたまたはリテラルのがパラメータのメモリにコピーされます。これは、じをつ2つのメモリがすることをします。のでは、パラメータメモリのだけでします。

をすると、プログラムスタックのメモリがポップされ、でされたパラメータのメモリをむびしのすべてのデータがされます。したがって、でされたはのにしません。

```
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;

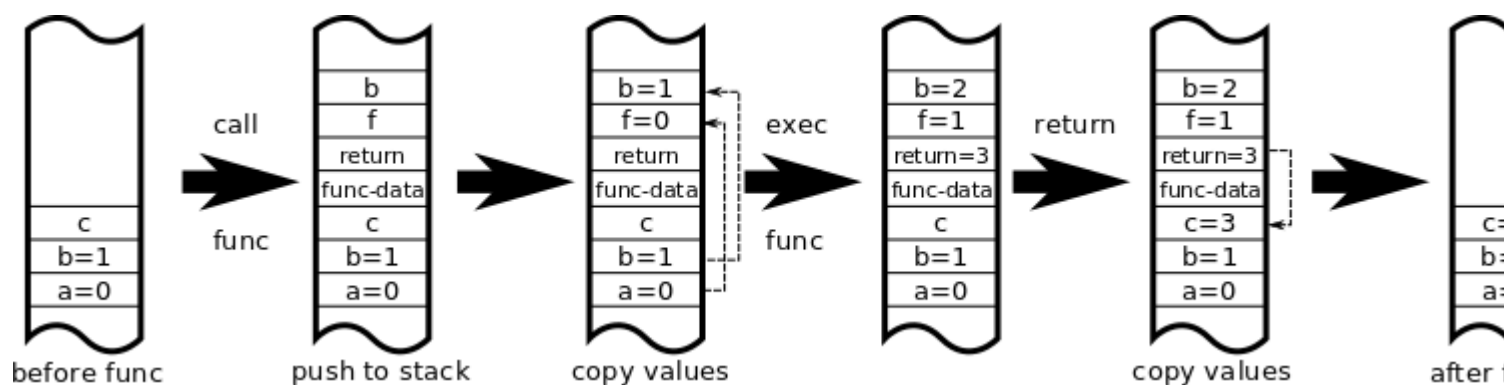
    c = func(a,b);
    //the return value is copied to c

    //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
}
```

このコードでは、mainにをします。これらはりてられたをします。をびすと、 f と inner_b 2つの

しいがされます。ここで、 b はとをし、メモリをしません。 $a \leftrightarrow f$ と $b \leftrightarrow b$ はじです。

のは、スタックでがこっているのか、そしてバリバリ b がないのかをしています。グラフィックはにではありませんが、をしています。



をすのではなく、これらののだけをすので、「によるびし」とばれます。

オンラインでC ++ "value by call"と "reference by call"をむ

<https://riptutorial.com/ja/cplusplus/topic/10669/c-plusplus--value-by-call-と--reference-by-call->

18: Const Correctness

- クラスClassOne {publicbool non_modifying_member_functionconst {/ * ... * /}};
- int ClassTwo :: non_modifying_member_functionconst {/ * ... * /}
- void ClassTwo :: modification_member_function{/ * ... * /}
- char non_param_modding_funconst ClassOneone、const ClassTwo * two{/ * ... * /}
- float parameter_modifying_functionClassTwoone、ClassOne * two{/ * ... * /}
- いClassThree :: non_modding_non_param_modding_fconst ClassOneconst {/ * ... * /}

`const` correctnessは、プログラマーがコードをしているがあるをすばやくできるため、にな
トラブルシューティングツールです。また、`Const Correct Function Parameters`されているよう
しないエラーがしくコンパイルされずにかねないようにします。

のためのクラスをするがはるかにです `const` それはからするよりも、`const` のクラスにさを。なは、
することができ、のクラス `const` それがあるようにしい `const` とにでそれをするをするために、し
いを。

これは、にじて `const` とじて `volatile` しさにすることもできますが、これはあまりにされません。

リフレクション

ISO_CPP

`const`のさにをる

C++チュートリアル

Examples

`const` しさは、インスタンスをするがあるコードだけがインスタンスをできる つまりきみアクセス
が、にインスタンスをするのないコードはできないようにコードをするプラクティスですそうす
なわち、みりアクセスのみをする。これにより、インスタンスがせずにされるのをぎ、コードの
エラーをらし、コードがインスタンスのをするかどうかをします。また、インスタンスをするが
ないときはいつでも `const` としてい、にするがなければ `const` としてして、をうことなくインスタン
スをすることができます。

これは、メンバーに `const` CVをえ、ポインタ/パラメータを `const` にすることによってわれま
だし、きみアクセスがなをく。

```
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.
```

`const`ののから、これはクラスのメンバからまり、にかってします。あなたがびそうと `const` からメンバを `const` インスタンス、またはから `const` としてわれているインスタンス `const`、コンパイラはあなたにそれがCV-をうことについてのエラーがします。

Constしいクラスデザイン

`const` -correctクラスでは、をしないすべてのメンバは、`this` `cv`された `const` ち、オブジェクトをしなことをします `const` インスタンスでもにできるの `mutable` フィールド; `const` `cv`されたがをす、そのも `const` なければなりません。 `const T*`は `T*`または `const T*`いずれかにバインドすることができますので、とCVインスタンスのでびすことができます。これにより、は、をうことなく、されたパラメータをするがないときに `const` としてすることができます。

さらに、`const` しいクラスでは、`Const Correct Function Parameters` でしたように、しされたすべてのパラメータは `const` しいものになるため、がにすることができるにのみできます。

まず、`this` `cv-qualifier`を試みましょう

```
// 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();
}

```

にこれを Const Correct Function Parameters とみわせると、クラスがに const correct になります。

```

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 {}

```

これは、についてオーバーロードとみわせることができます `const` インスタンスがある、々は1つのをしたいには、ネス `const`、およびなるるいそうでないは、コンテナが `const` にのみをするアクセスをすることをにします。

```

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]; }

    // ...
};

```

これはに、ほとんどのコンテナをるためにオーバーロードをするとともに、ライブラリでされている `const` にネスを。

Constのパラメータ

`const` -correctでは、がまたはにそれらをしないり、されたすべてのパラメータは `const` としてマークされ、プログラマがってしたくないものをするのをぎます。これは、が `const` インスタンスと `cv` インスタンスのをできるようにし、メンバがびされたときにインスタンスの `this` を `const T*` にします。ここで、`T` はクラスのです。

```

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

```

これではなく、すぐになにえるのものよりですが、`const` しいクラスその `const -correct` や `const -incorrect` クラスコンパイルエラーがします、`const -correct` クラスと `const -incorrect` のがにコンパイルされます、`const` しいですは、エラーがたくさんキャッチします `const` のような、すりけかせなを。[ただし、`const -incorrect` は、コンパイル・エラーがしますされた `const` それはするとき、インスタンスを `const` ものを。]

```

// 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];
}

```

ドキュメントとしての **Const Correctness**

`const` についてのよりなことの1つは、コードをすることとしてし、プログラマおよびのユーザにあるのをすることである。これらは、コンパイラによって `const` ためにされますが、コードが `const` しないことをします。

`const` **CV** メンバ—

- `const` メンバは、インスタンスをみもうとしてしているとできます。
 - びされたインスタスのなはしないでください。したがって、それらは、`mutable` をいて、びされたインスタスのメンバをしてはならない。
 - `mutable` をいて、インスタスのメンバをするのをびしてはいけません。
- に、`const` でないメンバは、インスタンをしようとしてしているとできます。
 - をしてもしなくてもよい。
 - をするのをびしてもびしなくてもよい。

これは、のをていなくても、えられたメンバがびされたに、オブジェクトのについてののをてるためにうことができます

```

// 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);
};

```

`const`のため、これらはコンパイラによってされます。

```

// 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`のパラメータ

- `const`である1つのパラメータをつは、それらのパラメータをみもうとしていけるとできます。
 - これらのパラメータをしたり、それらをするメンバをびしたりしないでください。
 - それらのパラメータをしたり、それらをするメンバをびすのにこれらのパラメータをしてはいけません。
- に、`const`ではない1つのパラメータをつは、これらのパラメータをするがあととできます
 - これらのパラメータをしたり、したりすることはできません。また、それらをするメンバをびすこともできます。
 - それらのパラメータをしたり、それらをするメンバをびすのにこれらのパラメータをしてもしなくてもよい。

これは、のをていなくとも、えられたにされたのパラメータのをにするためにできます。

```
// 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);
```

constのため、これらのはコンパイラによってされます。

```
// 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);
}

```

`const` **さ**を**する**ことはですが、これらの**を**ることはですが、これはプログラマーがにうがりますのMachiavelliでカプセル**を**るのとじです。

```

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.

```

しかし、これにより、プログラマは、コンパイラにして、`const`をし、コンパイラでしていることをコンパイラににえるがあるため、にしないり、`const`しいコードがそうしないとするのがにです。

オンラインでConst Correctnessをむ <https://riptutorial.com/ja/cplusplus/topic/7217/const-correctness>

19: constexpr

き

`constexpr` は、のを、でにできるとしてマークするためにできる **キーワード**、または C++ 17 コンパイルするブランチの1つだけをつ **if ステートメント** をマークします。

`constexpr` キーワードは C++ 11 でされましたが、C++ 11 がされてからがちましたが、なコンパイラではサポートされていませんでした。C++ 11 がされたで。C++ 14 のでは、すべてのなコンパイラが `constexpr` サポートしてい `constexpr`。

Examples

constexpr

`constexpr` とされたはに `const` あり、そのはとしてできます。

#define

`constexpr` は、`#define` づくコンパイルののわりにタイプセーフなものです。`constexpr` すると、コンパイルにされるがにきえられます。えは

C++ 11

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

のコードをします

```
cout << 12;
```

プリプロセッサベースのコンパイルマクロはなりません。してください

```
#define N 10 + 2

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

する

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

これはさらに `cout << 10 + 2;` される `cout << 10 + 2;`。しかし、コンパイラはよりくのをうがあります。また、しくされないとがします。

たとえば `#define`

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

フォーム

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

しかし、された `constexpr` はに ²⁴ をえるだろう。

const

`const` は、そのにメモリをとするです。 `constexpr` は `constexpr` はありません。 `constexpr` はコンパイルをしますが、これはできません。あなたは `const` がされないかもしれないとするかもしれません。ただし、

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

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

ほとんどのコンパイラでは、2のステートメントはしますたとえば、GCCでするがあります。のサイズは、あなたがっているように、でなければなりませんつまり、コンパイルのになります。2の `size2` は、にされるがりてられますコンパイラがコンパイルに ¹⁰ であることはわかっていますが、コンパイルではありません。

これは、 `const` がのコンパイルであるとそうでないがあることをします。の `const` がにコンパイルであることをしたりすることはできません。 `#define` することもできますが、のとしがあります。

したがって、にをしてください

C++ 11

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

    int arr[size];
}
```

`constexpr` は、コンパイルのにされなければなりません。したがって、あなたはできません

C++ 11

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

abs が constexpr ないり。

すべてののは constexpr でできます。

C++ 11

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

いことに、には、 auto うこともできます

C++ 11

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

constexpr

constexpr とされた constexpr はにインラインであり、そのようなをびすとにがられます。たとえば、のは、のでびされた、もされます。

C++ 11

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

したがって、びしのは、のバウンドまたはテンプレートのとして、または constexpr をするために constexpr ます。

C++ 11

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

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

のりのから constexpr をした、s は constexpr であるため、s へののはしません。また、コンパイル const をりてるがあります。に、Sum が constexpr でない、のサイズもではありませ constexpr。

constexpr のいは、ののようになうことができることです。

C++ 11


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

Sumは、constexprではなく、のとしてコンパイルされ、をとり、をします。2つのをくはありません。

このようなびしをconstにりてると、コンパイルされません

C++ 11

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

はです constexprはコンパイルをりてるがあります。しかし、のびしは、Sumをconstexpr Rはconstですが、Lはconstexprとしています。

constexprは、コンパイルもすがあります。はコンパイルされません。

C++ 11

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

a1はconstexprであり、がのconstexprではないconstexprです。それはconstexpr、それがりてもしません-のはパラメータはとしてです。 a a

C++ 11

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

さらに、もコンパイルされません。

C++ 11

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

abs(a)はではないのでabs(a) abs(10)もしませんabs(10)、absはconstexpr intさないためです

これはどうですか

C++ 11

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

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

々はのAbsあるconstexpr、とのAbsまた、のルールをついていません。また、びしサイト Sumでは、はconstexprされます。したがって、Sum(-10, 20)のびしはコンパイルのになり、30ます。

if ステートメント

C++ 17

if constexpr ステートメントは、きでコードをコンパイルするためにできます。はでなければなりません。されていないブランチはされます。テンプレートのされたはインスタンスされません。えは

```
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
}
```

さらに、されたステートメントでのみされるおよびはするはなく、されたreturnステートメントはのりのにはされません。

if constexprが#ifdefとなるif constexpr#ifdefきでコードをコンパイルしますが、にできるにのみづいています。たとえば、テンプレートパラメータのにして#ifdefしてコードをきでコンパイルすることはできません。、になコードをするためにif constexprをすることはできませんが、#ifdefはです。

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

オンラインでconstexprをむ <https://riptutorial.com/ja/cplusplus/topic/3899/constexpr>

20: constキーワード

- `const myVariable =;` // constをします。えられない
- `const タイプ myReference = myVariable;` // constへのをします。
- `const* myPointer = myVariable;` // constへのポインタをします。ポインタはできますが、ポインタをしてとなるデータメンバをすることはできません
- `タイプ* const myPointer = myVariable;` // constポインタをします。ポインタをのものをすようにりてすることはできませんが、になるデータ・メンバーはすることができます
- `const タイプ* const myPointer = myVariable;` // constへのポインタをconstにします。

マークされた`const`¹にすることはできません。`const`のオペレーションをびそうとすると、コンパイルエラーがします。

¹まあ、それは`const_cast`をしてすることができますが、あなたはほとんどそれをうべきではありません

Examples

Constローカル

と。

```
// 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
```

とポインタのバインディング

```
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
```

Constポインタ

```
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;

```

Constメンバ

クラスのメンバは`const`することができます。これはコンパイラとのに、このがオブジェクトをしないことをします。

```

class MyClass
{
private:
    int myInt_;
public:
    int myInt() const { return myInt_; }
    void setMyInt(int myInt) { myInt_ = myInt; }
};

```

`const`メンバでは、`this`ポインタは`const MyClass *`ではなく`const MyClass * MyClass *`です。つまり、のメンバーをすることはできません。コンパイラはをします。したがって、`setMyInt`は`const`できませんでした。

できるだけメンバを`const`としてマークするべきです。`const MyClass`では、`const`メンバーだけをびすことができます。

`static`メソッドは`const`としてできません。これは、メソッドがクラスにし、オブジェクトにしてびされないためです。したがって、オブジェクトのをしてすることはできません。したがって、`static`メソッドを`const`としてすることはになります。

constおよびconst getterメソッドでコードのをける

`const`だけがあるC++のメソッドでは、オーバーロードすることができます。によっては、のメンバへのをすゲッタの2つのバージョンがながあります。

`Foo`クラスとし、じをする2つのメソッドをち、`Bar`のオブジェクトへのをします。

```

class Foo
{
public:
    Bar& GetBar(/* some arguments */)
    {
        /* some calculations */
        return bar;
    }

    const Bar& GetBar(/* some arguments */) const
    {
        /* some calculations */
        return bar;
    }

    // ...
};

```

それらののいは、1つのメソッドがconstであり、constオブジェクトののにできるをし、2のメソッドがconstであり、constをすということです。

コードのをけるために、あるメソッドをのメソッドからびすというがあります。しかし、constメソッドをconstオブジェクトからびすことはできません。しかし、constのものからconstメソッドをびすことができます。constをするには 'const_cast'をするがあります。

はのとおりです。

```

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;
    }
};

```

のコードでは、々はconstのバージョンをびすGetBarconstからGetBar constにこれをキャストすることによって const_cast<const Foo*>(this) 。 constからconstメソッドをびすので、オブジェクトはconstであり、constをキャストすることはされています。

のよりなをべてください。

```

#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';
}

```

オンラインでconstキーワードをむ <https://riptutorial.com/ja/cplusplus/topic/2386/const> キーワード

21: Cの

き

これは、CコードがC++コンパイラでれることをします。

Examples

みのキーワード

のは、C++でなをっているキーワードですはCではですが、C++ではではありません。

```
int class = 5
```

これらのエラーはにできます。のをするだけです。

くけされたポインタ

Cでは、ポインタはC++でにキャストするがあるvoid*キャストすることができます。はC++ではですが、Cではです

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

なキャストをするとこのがになりますが、そののがじるがあります

またはスイッチ

C++では、gotoやswitchをスキップすることはできません。はCではですが、C++ではではありません。

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

これらのバグはがながあります。

オンラインでCのをむ <https://riptutorial.com/ja/cplusplus/topic/9645/cの>

22: Elisionをコピー

Examples

コピーエリートの

には、オブジェクトをするためにオブジェクトがコピーまたはされるがあります。コピーエリートにはリターンのとばれるは、あるので、コンパイラがコピーやをけることをされたです。

のをえてみましょう

```
std::string get_string()
{
    return std::string("I am a string.");
}
```

のないしによると、このはな `std::string` をし、それをりオブジェクトにコピー/してにします。が、コードがどのようにされるかはにです。

コピーエリートは、C++コンパイラがなコピーのとそののコピー/をできるようにするルールです。つまり、コンパイラはにをり、のりをすることができます。これはらかにパフォーマンスをします。

ただし、ユーザーには2つのにえるがあります。

1. にはびされたコピー/コンストラクタがです。たとえコンパイラがコピー/をエリートしたとしても、そのはまだコピー/できなければなりません。
2. コピー/コンストラクタのは、がこりうるではされません。のをしてください。

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();
}
```

びし `func` はをしますかなは `my_type` で、 `my_type` はなタイプなので、 "Copying" はしてされません。それでは "く" とされますか

コピーエリシエーションルールがなければ、に「」をするがあります。しかし、コピー・エリジス・

ルールがするため、ムーブ・コンストラクタはびされることもあればびされないこともあります。にします。

したがって、コピー/コンストラクタのびしにすることはできません。

elisionはであるため、コンパイラはすべてのケースでelisionをサポートしないがあります。また、コンパイラがのケースをエリートするかどうかにかかわらず、タイプは、されていないをききサポートするがあります。したがって、コピーがされたでも、びされなくてもはコピー・コンストラクタをするがあります。

されたコピー・エリジョン

C++ 17

、elisionはです。ほとんどすべてのコンパイラは、もなにはコピー・エリジョンをサポートしますが、elisionをすると、としてユーザーにのがかかります。すなわち、コピー/ムーブされていないタイプは、コピー/ムーブされていなければならない。

えば

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
    return std::lock_guard<std::mutex>(a_mutex);
}
```

これは、`a_mutex`があるシステムによって`a_mutex`されているミューテックスであるが、ユーザがスコープロックをつことをむにです。

`std::lock_guard`はコピーまたはできないため、これもではありません。すべてのC++コンパイラがコピー/をするにもかかわらず、ではそのタイプのがであるがあります。

C++まで17。

C++ 17では、ののをにしてコピー/がこらないようにするためにelisionをしています。のコードをえてみましょう。

pre-C++ 17のでは、そのコードはなものをし、そのなものをしてりにコピー/しますが、なコピーはできます。C++ののでは、それはまったくなものではありません。

C++ 17では、`prvalue`は、とじのオブジェクトをするためにされると、にされません。は、そのオブジェクトをします。りとじの`prvalue`をす、はコピー/コンストラクタをつはありません。したがって、C++の17のルールのでは、のコードはします。

`prvalue`のがされているとする、C++はします。の`get_lock`すると、コピー/はありません。

```
std::lock_guard the_lock = get_lock();
```

`get_lock`のはじのオブジェクトをするためにされるprvalueなので、コピー/はやりません。そのはしてなものではありません。 `the_lock`をするためにされます。 `elide elide`にはコピー/がないので、`elion`はありません。

したがって、「されたコピー・エリジョン」というのは**ただであるが、これはC++のためにされているようなのである**。それはまったくをするものではありません。それはコピー/をし、C++をして、コピー/がわれないようにします。

このは、prvalueをむにのみします。このように、これはのelisionルールをします

```
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;
}
```

これはコピー・エリートのなケースですが、C++ 17のルールではこのコピー/ムーブはされません。そのため、りのにするコピー/コンストラクタは、にはまだするがあります。 `lock_guard`はそうでない、これはまだコンパイルエラーです。では、トリビュアコピーのオブジェクトをしたるときに、コピー/をするをすることができます。これはレジスタでそのようなオブジェクトをかすことをにするためであり、のABIはびしでそれをするをすることができます。

```
struct trivially_copyable {
    int a;
};

void foo (trivially_copyable a) {}

foo(trivially_copyable{}); //copy elision not mandated
```

り elision

からprvalueをし、prvalueのがのりのとじ、prvalue temporaryからのコピー/はできます。

```
std::string func()
{
    return std::string("foo");
}
```

ほとんどすべてのコンパイラは、この、なをします。

パラメータ elision

にをし、がのパラメータのprvalueであり、このがでない、prvalueのはできます。

```
void func(std::string str) { ... }
```

```
func(std::string("foo"));
```

これは、な`string`をし、それをパラメータ`str`することをし`str`。Copy elisionは、な`+`をするのではなく、このが`str`でオブジェクトをできるようにし`str`。

これは、コンストラクタが`explicit`されているにすぎません。例えば、`func("foo")`とくことができ`string`が、`string`は`const char*`から`string`するのコンストラクタがあるだけです。そのコンストラクタが`explicit`であった、`explicit`コンストラクタを必ずため`explicit`なものをすることがされます。Copy elisionは、なコピー/をするがなくなります。

きり elision

`lvalue`をからすと、この`lvalue`

- そのにローカルなをします。 `return`にされます
- はパラメータではありません
- のはのりのとおりです

これらのすべてがする、からのコピー/はできます。

```
std::string func()
{
    std::string str("foo");
    //Do stuff
    return str;
}
```

よりなケースは`elision`にですが、よりなケースでは、コンパイラがにそれをするはなくなります。

```
std::string func()
{
    std::string ret("foo");
    if(some_condition)
    {
        return "bar";
    }
    return ret;
}
```

コンパイラはまだ`ret`すことができますが、そうするはなくなります。

のように、パラメータについては`elision`はされていません。

```
std::string func(std::string str)
{
    str.assign("foo");
    //Do stuff
    return str; //No elision possible
}
```

エリッションをコピーする

`prvalue`をしてをし、そのがprvalueとしをつ、コピーはできます。

```
std::string str = std::string("foo");
```

コピーはこれをに `std::string str("foo");` し `std::string str("foo");` わずかないがあります。

これはりでもします

```
std::string func()
{
    return std::string("foo");
}

std::string str = func();
```

copy elisionがなければ、`std::string`のmoveコンストラクタへの2のびしをきこします。Copy elisionは、これがコンストラクタを1または0びすことをし、ほとんどのコンパイラはをします。

オンラインでElisionをコピーをむ <https://riptutorial.com/ja/cplusplus/topic/2489/elision> をコピー

23: ISO C ++

き

1998には、C ++をにするののがありました。そのから、C ++はC ++のなるにしました。このページでは、のバージョンとして、すべてのなるのとそのをつけることができます。これらののいのは、よりなページでしています。

C ++についてすると、しばしば「」がされます。しかし、そのはですか

C ++にはいがあります。Bell LabsのBjarne Stroustrupによるなプロジェクトとしてされ、90めにはかなりしました。のがC ++コンパイラをしていたため、ユーザはいコンピュータでC ++コンパイラをすることができました。しかし、これをにするために、これらのするすべてのコンパイラは、ののをするがあります。

そので、Cはにされました。これは、のながかれたことをします。これはANSIにされ、のためにをき、その1989にした。1、なるでなるをするため、ISO、ギリシヤのisosからした、しいことをする。は、としてをした。

C ++では、からにがあったことはらかでした。ISOのワークグループがされましたWG21とばれ、SubCommittee 22にあります。このワークグループは、1995にのをしました。しかし、たちプログラマーがっているように、されたデリバリーにはのよりもはもなく、C ++にもこりました。1995に、STLというのクールなしいライブラリがしました。WG21でく々は、C ++ドラフトにスリムバージョンをすることにめました。ながら、これによりがしたばかりで、3にはにがしました。ISOはになであるため、C ++にはあまりでないISO / IEC 14882のがけられました。がされるにつれて、このなバージョンは148821998としてられていました。

、スタンダードをするというがありました。このは、C ++コンパイラのをにすることをとしたにドキュメントです。わずかなあいまいさでもするがあるので、2003までに148822003としてアップデートがリリースされました。しかし、これはC ++にのもしませんでした。は2のです。

この2のアップデートは、C ++ 0xとしてられていました。なぜなら、2008か2009かがかっていなかったからです。そのバージョンにもものがあり、148822011になりました。

いにも、WG21はそれをびこらせないことにめました。C ++ 11はをて、C ++にたなをせました。だから、いをするために、3のはからまで3で148822014になりました。

はそこでもわらなかった。C ++ 17がされ、C ++ 20のがされました。

Examples

の

されているすべてのISOは、ISO <http://www.iso.org> からされています。 C ++のはとでです。

なるバージョンの

- のC ++ 20またはC ++ 2aとばれることもある [のき HTML](#)
- C ++ 17またはC ++ 1zともばれる [20173のN4659](#)。
- C ++ 14にはC ++ 1yとばれる [201411](#)、[N4296](#)
- C ++ 11にはC ++ 0xともばれる [20112のドラフトN3242](#)
- C ++ 03
- C ++ 98

C ++ 11

C ++ 11は、C ++のなです。は、[isocppにするグループされた](#)のと、よりなドキュメントへのリンクです。

な

- [オート](#)
- [レンジ・フォー・ステートメント](#)
- [リスト](#)
- [とセマンティクス](#)
- [Rvalueとセマンティクス](#)
- [ラムダ](#)
- [をさせないためのnoexcept](#)
- [constexpr](#)
- [nullptr](#) - ヌルポインタリテラル
- [のコピーと](#)
- [インラインネームスペース](#)
- [ユーザーリテラル](#)

クラス

- [=デフォルトと=](#)
- [デフォルトのとコピーの](#)
- [コンストラクタの](#)
- [クラスメンバ](#)
- [されたコンストラクタ](#)
- [オーバーライドコントロールオーバーライド](#)
- [オーバーライドコントロールfinal](#)
-

そののタイプ

- enumクラス
- long long - よりい
-
- された
- されたPOD

テンプレート

- Externテンプレート
- テンプレートエイリアス
- バリデリックテンプレート
- テンプレートとしてのローカル
- メモリモデル
- によると
- [スレッドローカルストレージ](#)

そのの

- C ++ 11の__cplusplusのはですか
- のりの
- み
-
- [static_assertコンパイルアサーション](#)
- のリテラル
-
- アラインメント
- C99の

ライブラリ

- unique_ptr
- shared_ptr
- weak_ptr
- ガベージコレクションABI
- タプル
- タイプ
- とバインド
-
- タイムユーティリティ
-

- スコープキアローケータ

コンテナとアルゴリズム

- アルゴリズムの
- コンテナの
- `unordered_*`コンテナ
- `std::array`
- `forward_list`
- スレッド
-
- ロック
-
- アトミック
- と
-
- プロセスをする

C++ 14

C++ 14は、しばしばC++ 11のバグとされます。これには、C++ 11のものがもくまれているのリストのみがまれています。は、[isocppにするグループされた](#)のと、よりなドキュメントへのリンクです。

-
- バイナリリテラル
 - されたりの
 - `decltype`
 - [されたラムダキャプチャ](#)
 - [なラムダ](#)
 - テンプレート
 - `constexpr`
 - `[[deprecated]]`
 - [り](#)

ライブラリ

-
- ロック
 - `std::types`のユーザーリテラル
 - `std::make_unique`
 - `_t`エイリアス
 - [タイプによってタブルをう](#) `get<string>(t)`
 - [な](#) `Functor` `greater<>(x)`

- `std::quoted`

りみ

- `std::gets`はC++ 11ではされ、C++ 14ではされました。
- `std::random_shuffle`はされていません

C++ 17

C++ 17はがされており、のためにされています。これらのをにサポートしているコンパイラでは、C++ 1zとされます。

-
- のりみ
 - `auto`テンプレートをする
 - されたコピーエリシジョン
 - コンストラクタのテンプレートパラメータ
 - バインディング
 - コンパクトなネストされた
 - しい `[[fallthrough]]` `[[nodiscard]]`、 `[[nodiscard]]`、 `[[maybe_unused]]`
 - `static_assert`デフォルトメッセージ
 - `if`と`switch`
 - インライン
 - `if constexpr`
 - の
 - オーバーライドされたデータのメモリリテ

ライブラリ

- `std::optional`
- `std::variant`
- `std::string_view`
- コンテナのための`merge()`と`extract()`
- `<filesystem>`ヘッダをつファイルシステムライブラリ。
- ほとんどのアルゴリズムの平行バージョン `<algorithm>`ヘッダー。
- `<cmath>`ヘッダにをする。
- マップ`<>`、`unordered_map <>`、`set <>`、および`unordered_set <>`のでノードをする

C++ 03

C++ 03は、にC++ 98のにします。これらのとはに、しいが1つされます。

-
-

C ++ 98

C ++ 98は、C ++ののされたバージョンです。Cのとしてされたので、C ++からCをするくのがされています。

C89 / C90にして

- クラス、クラス、メンバ、メンバ
- のオーバーロード、のオーバーロード
- 1コメントCでC99でされています
-
- と
- ブールCでC99でされています
- テンプレート
-
-
- のキャスト

ライブラリ

- テンプレートライブラリ

C ++ 20

C ++ 20は、のC ++ 17のであるC ++ののです。は[ISO cppウェブサイト](#)ですることができます。

のは、C ++のリリースでけられたもので、2020をとしています。

、はけられていません。

ライブラリ

ライブラリはけられていません。

オンラインでISO C ++をむ <https://riptutorial.com/ja/cplusplus/topic/2742/iso-c-plusplus>

24: OpenMPでの

き

このトピックでは、OpenMPをしたC++ののについてします。OpenMPは[OpenMPタグ](#)でよりにされています。

またはは、コードのをにします。

OpenMPはみみのコンパイラであるため、なヘッダやライブラリはありません。ただし、`omp_get_thread_num()`などのOpenMP APIをするは、`omp.h`とそのライブラリをインクルードするがあります。

コンパイルにOpenMPオプションがになっていないと、OpenMP `pragma`はされます。コンパイラのマニュアルでコンパイラオプションをすることができます。

- GCCは`-fopenmp`し`-fopenmp`
- Clangは`-fopenmp`
- MSVCは`/openmp`し`/openmp`

Examples

OpenMP パラレルセクション

このは、コードのセクションのをにするをしています。

OpenMPはビルトインのコンパイラであるため、サポートされているコンパイラでもライブラリをしません。あなたは、したいとうかもしれ`omp.h`あなたはOpenMPのAPIのいずれかをする。

サンプルコード

```
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;
```

ここでは、2つのなをし、オペレーティングシステムとハードウェアにしています。これは、そのようなからするのもしています。

A	B
こんにちはのわりをめる	のハロー・エンド

OpenMP パラレルセクション

これは、コードのをにするをしています

```
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";
```

- こんにちははにわります
- をめるこんにちはのわり
- こんにちはにのわりをめる
- にまるこんにちはのわり

はされていないため、ののいずれかをすることができます。

OpenMP Parallel For Loop

ここでは、ループをににしてにするをします。

```
// 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
...
```

* りてられたのインデックスはにはされないので、forループでされるベクトルのサイズをしないうようにしてください。

OpenMPギャザリング/リダクション

このは、std::vectorとOpenMPをしてまたはをするをしています。

々はのスレッドがのをするたちをけたいシナリオをとっているはず、intにするために、ここでされ、のデータにきえることができます。

これは、セグメンテーションフォールトやメモリアクセスをけるためにスレーブからのをマージするがあり、ライブラリやカスタムコンテナライブラリをしたくないににです。

```
// 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
...

```

オンラインでOpenMPでのをむ <https://riptutorial.com/ja/cplusplus/topic/8222/openmpでの>

25: Pimpl Idiom

にはポインタまたはチェシヤのとはementationをIMPLするためのp ointerPIMPL イディオムは、.cppファイルでされたに、すべてのプライベートデータメンバをさせることによって、クラスのコンパイルをさせます。

クラスはへのポインタをしています。このようにして、ヘッダーファイルはprivateメンバーでされるクラスを#includeするがないように、することができます。

pimplイディオムをする、プライベートデータメンバーをする、それにするクラスをコンパイルするはありません。

Examples

なPimpl イディオム

C ++ 11

ヘッダファイル

```
// 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
};
```

ファイル

```
// 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
}
```

`pImpl`は、`Widget`またはそのほとんどまたはほとんどがまれています。ヘッダーファイルでされているの`Widget`のわりに、でしかできません。

`pImpl`は「へのポインタ」のです。`Widget`の「の」は`pImpl`ます。

これが`unique_ptr`であるためには、`Impl`がにえるファイルのあるに`Impl ~Widget()`しなければなら
ないことにしてください。そこに`=default`することはでき`=default`が、`Impl`がされていないは
`=default`になり、プログラムはになになり、はありません。

オンラインでPimpl Idiomをむ <https://riptutorial.com/ja/cplusplus/topic/2143/pimpl-idiom>

26: RAII リソースがされる

RAIIはacquisitionにI S Iの initializationをesource Rのです。 SBRMScope-Based Resource ManagementまたはRRIDResource Release Is Destructionともばれることがあります。RAIIはリソースをオブジェクトのにびつけるためにされるイディオムです。C++では、オブジェクトのデストラクターは、オブジェクトがスコープになったときににされます。これをして、リソースのクリーンアップをオブジェクトのにびつけることができます。

にするのあるリソースロック、ファイルハンドル、りてみバッファなどをするがあるはいつでも、オブジェクトをしてそのリソースをすることをします。スタックのアンワインディングは、またはスコープのになくするので、リソースハンドラオブジェクトは、すべてのなおよびのコードパスをにすることなく、リソースをクリーンアップします。

RAIIがリソースのをえるをにしないことはにする。1つのケースは、らかに、デストラクタがびされないようにするクラッシュまたはexitコールです。プロセスの、OSはメモリのようなプロセスローカルリソースをクリーンアップするので、これはほとんどのではありません。しかし、システム・リソースすなわちパイプ、ロック・ファイル、メモリーでは、プロセスがそれのでクリーンアップしなかった、つまりロック・ファイルがあるはスタートアップ・テストでそれをするためのがです。にpidをつプロセスをし、それにじてします。

のは、UNIXプロセスがexec-familyからをびすとき、つまりfork-execのにしいプロセスをするときです。ここで、プロセスはメモリRAIIオブジェクトをむのフルコピーをちますが、execがびされると、そのプロセスではデストラクタのどれもびされません。、プロセスがforkされ、いずれのプロセスもexecをびさないは、のプロセスですべてのリソースがクリーンアップされます。これはにフォークにされたすべてのリソースにしてのみですが、システムリソースではのプロセスがリソースへのロックファイルへのパスのみをち、にしようとするプロセスはする。

Examples

ロッキング

いロック

```
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.
    }
}

```

これはmutexのロックとロックをするつたです。unlock() してミューテックスをしくリリースするためには、をしたすべてのフローがunlock() びすことをプログラマがするがあります。のように、これはなプロセスであり、メンテナがでパターンをするがあるためです。

にされたクラスをしてRAIIをすると、はです

```

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は、になクラステンプレートで、コンストラクタのにlock() をびし、へのをし、そのデストラクタのにしてunlock() をびします。つまり、lock_guardがになると、mutexロックがされることがされます。になったがかであるかどうかはありません。すべてのケースがされます。フローになく、たちはしくロックをするををしています。

に/ ScopeExit

いくつかのリソースをうなクラスをしたくないは、クラスをくことができます

```

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)}; }

```

```
}
```

そしてその

```
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)`
}
```

1をするためには、デストラクタについてののです。

- `~Finally() noexcept { f(); }` のに `std::terminate` が呼びされます
- `~Finally() noexcept(noexcept(f())) { f(); }` `terminate` は、スタックのきしにがしたにのみ呼びされます。
- `~Finally() noexcept { try { f(); } catch (...) { /* ignore exception (might log it) */ } }` `std::terminate` 呼びませんが、エラーをすることはできませんスタックのきしのでも。

ScopeSuccess ++ 17

C ++ 17

`int std::uncaught_exceptions()` おかげで、したときにのみされるアクションスコープにスローされたはありません `int std::uncaught_exceptions()` をできます。これまで、 `bool std::uncaught_exception()` は、スタックアンwindがされているかどうかをするだけです。

```
#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
    }
}

```

Success 1

ScopeFailc ++ 17

C ++ 17

`int std::uncaught_exceptions()` おかげで、スコープでスローされたでのみされるアクションをできます。これまで、`bool std::uncaught_exception()` は、スタックアンwindがされているかどうかをするだけです。

```

#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
    }
}

```

Failure 2

オンラインでRAIIリソースがされるをむ <https://riptutorial.com/ja/cplusplus/topic/1320/raii-リソースがされる>

27: RTTI

Examples

タイプの

`typeid` `std::type_info` オブジェクトの `.name()` メンバーをして、にのをできます。

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

int main()
{
    int speed = 110;

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

```
int
```

dynamic_cast

`dynamic_cast<>()` をとしてすると、**な** をキャストダウンするのにちます。

いくつかのクラス `B` と `C` でのをしなければならないが、`class A` けったは、のようにく

```
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();
}
```

typeid キーワード

`typeid` **キーワード** はで、オペランドのがのである、そのオペランドにするのをします。 `const` `std::type_info` のをします。トップレベルの CV はされます。

```
struct Base {
    virtual ~Base() = default;
```

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

`typeid`はにすることもできます。この、のトップレベルのはりかれ、の`cv`はされます。したがって、のは、`typeid(Derived{})`わりに`typeid(Derived{})` `typeid(Derived)`されているがあります。

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

`typeid`がポリモーフィックなクラスではないにされた、オペランドはされず、されるはです。

```
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
```

C++でどのキャストをうか

のポインタ/をするには、**`dynamic_cast`**をします。

のには**`static_cast`**をします。

ビットパターンのレベルのために**`reinterpret_cast`**をしてください。なをってしてください。

`const` / `volatile`をキャストするために**`const_cast`**をしてください。あなたが`const-incorrect` APIをしてちしないり、これをけてください。

オンラインでRTTIをむ <https://riptutorial.com/ja/cplusplus/topic/3129/rtti->

28: SFINAEはりではない

Examples

enable_if

`std::enable_if`はboolをしてSFINAEをトリガするユーティリティです。これはのようにされます。

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

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

つまり、`enable_if<true, R>::type`はRエイリアスですが、`enable_if<false, T>::type`は、`enable_if`に`type`メンバーがないため`enable_if`です。

`std::enable_if`をしてテンプレートができます。

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

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

ここで、`negate(1)`びしはあいまいさのためにします。しかし、2のオーバーロードはにはされませんので、のようにできます。

```
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(); }
```

は、`!std::is_arithmetic<int>::value`が`false`であるため、`negate<int>`をインスタンス`negate<int>`がし`false`。SFINAEのため、これはハードエラーではなく、このはにセットからされます。その、`negate(1)`はなが1つしかなく、それがびされます。

それをいつするか

`std::enable_if`はSFINAEのヘルパー`std::enable_if`が、SFINAEをにさせるものではありません。
`std::size`とたをするためのこれらの2つの、つまりコンテナまたはのサイズをするオーバーロードセット`size(arg)`をえてみましょう


```

// 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]);

```

`is_sizeable`がにされているとすると、これら2つのSFINAEにしてにでなければなりません。くのがで、でレビューとがですか

ここで、ラップアラウンドまたはモジュラーをしてきオーバーフローをするヘルパーをするを試みましょう。つまり、たとえ`i`が`INT_MAX`つ`int`であっても、がにされるというにして、えは`incr(i, 3)`は`i += 3`とじになります。これらは2つのなです

```

// 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);

```

もうくのがで、でレビューとがですか

`std::enable_if`は、リファクタリングとAPIでどのようにするかです。`is_sizeable<Cont>::value`は、`cont.size()`がかどうかをするためのもので、`size1`にされるをするだけでよくなりますが、`is_sizeable`がいくつかのでされるかどうかによってなります。インプリメントが`incr1`のにれるときよりもはっきりとそのをする`std::is_signed`とです。

void_t

C++ 11

`void_t`はののを`void`にマップするメタです。 `void_t`のは、のをにすることです。

`std::void_t`はC++ 17のですが、それまではするのがにです

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

いくつかのコンパイラではしなるがです。

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

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

`void_t`のは、のをチェックするのをくことです。たとえば、にをらないメンバー`foo()`があるかどうかをべてみましょう。

```
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 {};
```

これはどのようにしますか `has_foo<T>::value`をインスタンスしようとする、コンパイラは`has_foo<T, void>`をします。たちには2つのがありますプライマリと、このなをインスタンスしなければならないセカンダリです。

- `T`メンバって `foo()`、にどのようにされます `void`、びをについてにましいです。したがって、`has_foo<T>::value`は`true`になり`true`
- `T`がそのようなメンバをたないまたはのをとする、のためのはし、フォールバックするプライマリテンプレートのみがあります。したがって、`has_foo<T>::value`は`false`です。

よりなケース

```
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 {};
```

これは`std::declval`または`decltype`しません。

`void`ののパターンにくかもしれません。これをすることができます

```
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...>;

```

これは `std::void_t` をし、のテンプレートとして `can_apply` たがのにきえられたかどうかをすインジケータのようにします。のは、 `can_apply` をって `can_apply` ようにきすことができます

```

template<class T>
using ref_t = T&;

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

```

そして

```

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?

```

のバージョンよりもシンプルにえます。

`can_apply` た `std` の `post-C ++ 17` のがあります。

`void_t` のは `Walter Brown` によってされました。は `CppCon2016` でらしい [プレゼンテーション](#) をしました。

テンプレートのの **`decltype`**

C ++ 11

の1つは、の `decltype` をしてりのをすることです。

```

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);
}

```

が `to_string()` を呼び出すことができるので `convert_to_string()` を呼び出すと、`details::convert_to_string()` は 2つの overload があり `details::convert_to_string()`。 からの overload であることが正しい。 `0` の `int` からの overload も 1 つの overload である `0` に...

`to_string()` を呼び出すことができないので `convert_to_string()` を呼び出すと、このテンプレートのインスタンスによって `decltype(to_string(val))` はありません。その、それはオーバーロードセットから選ばれます。 2つのテンプレートは選ばれていないので、代わりに `operator<<(std::ostream&, T)` があります。それが `oss << val` というにテンプレートスタックのハードコンパイルエラーがあります。

SFINAE とは

SFINAE は、は **N A N E** の error のよ **S** の substitution **F** の failure のです。テンプレートまたはクラステンプレートをインスタンスするために `std::enable_if` をすることによってされるコードは、ハードコンパイルエラーではなく、エラーとしてのみ扱われます。

テンプレートのインスタンスまたはクラステンプレートでののは、そのオーバーロードセットから選ばれます。これは、したがってなかったかのようにです。

```

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.

```

このコンテキストでは `begin(vals)` のみ `begin(vals)` とみなされ、はすべてハードエラーとみなされます。

```

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

な

のコードスニペットのように、テンプレートパラメータリストにバリデーションテンプレートパックがある

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

のC++ 17ライブラリは、のすべてのパラメータにSFINAEのをすこと**enable_if**をくためのなしな`Args`かのパラメータのいずれかの`Args`。C++ 17がしています**`std::conjunction`**と**`std::disjunction`**このをします。えは

```
/// 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) { //... };
```

C++ 17をできないは、これをするためのいくつかのソリューションがあります。そのうちの1つは、こののにされているように、ベースケースのクラスとなをする**こと**です。

あるいは、また、でのをしてもよい**`std::conjunction`**と**`std::disjunction`**かなりストレートフォワードなで。のでは、をデモンストレーションし、**`std::enable_if`**とみわけて2つの**`enable_if_all`**と**`enable_if_any`**をします。これらは、にされているものとまったくじです。これにより、よりスケラブルなソリューションがされるがあります。

`enable_if_all` および **`enable_if_any`**

のは、エミュレートしましょう**`std::conjunction`**して**`std::disjunction`**カスタマイズして**`seq_and`**と**`seq_or`**それぞれ

```
/// 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> {};
```

そのはかなりです

```
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>;
```

にいくつかのヘルパー

```
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;
```

いもです

```
/// 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

type_traitのをするためにSFINAEについて、ながdetected_or detected_t、 is_detected detected_or、 detected_t。

テンプレートパラメータtypename Default、 template <typename...> Opとtypename ... Args

- is_detected Op<Args...> std::false_type Op<Args...>にじてstd::true_typeまたはstd::false_typeエイリアス
- detected_t のエイリアスOp<Args...>やnonexistentのOp<Args...>
- detected_or をつのvalue_tさ is_detected、 およびtypeであるOp<Args...>またはDefaultのにOp<Args...>

のように、SFINAEのstd::void_tをしてできます。

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...>;

```

メソッドのをするためのは、にできます。

```

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");
```

のオプションをした

いくつかのオプションの中からするがあるは、`enable_if<>`で1つだけをにすることは、いくつかのものにするがあるため、かなりなことがあります。

オーバーロードのけは、わりに、つまりタグディスパッチをしてすることができます。

をとするものをテストするのではなく、のすべてのバージョンのをテストするわりに、なものだけをテストします。ましくは、のリターンの`decltype`で`decltype`します。

これにより、いくつかのオプションがにされ、イテレータのタグ`random_access_tag`とに、「タグ」をするものをするすることができます。これは、クラスのクラスよりもれているクラスよりも、がれているためにします。

```
#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>{});
}
```

がよりれている、よりもれているなど、オーバーロードをするためににされるのがあります。

しかし、タグディスパッチはののにすることができ、インテントではもうしです。

オンラインでSFINAEはりではないをむ <https://riptutorial.com/ja/cplusplus/topic/1169/sfinae-はりではない->

29: static_assert

- `static_assert bool_constexpr、message`
- `static_assert bool_constexpr / * C ++17 */`

パラメーター

パラメータ	
<code>bool_constexpr</code>	チェックする
メッセージ	<code>bool_constexpr</code> が <code>false</code> のときにするメッセージ

[ランタイムアサーション](#)とはなり、アサーションはコンパイルにチェックされ、されたビルドをコンパイルするときにもされます。

Examples

static_assert

アサーションとは、をチェックするがあり、`false`のはエラーです。 `static_assert()`、これはコンパイルにわれます。

```
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);
}
```

`static_assert()`にはこのパラメータ、つまり`bool`の`constexpr`があります。これは、のパラメータ、リテラルであるメッセージを、っているかもしれません。C ++ 17から、2のパラメータはオプションです。それに、それはです。

C ++ 17

```
template<typename T>
T mul10(const T t)
{
    static_assert( std::is_integral<T>::value );
    return (t << 3) + (t << 1);
}
```

のにされます。

- に、コンパイルのは、あるの`constexpr`でとされます

- テンプレートは、されたののプロパティをするがあります
- のようなテストケースをしたいとえています。
 - テンプレートメタ
 - constexpr
 - マクロメタプログラミング
- のがC++バージョン
- レガシーコードの、 sizeof(T) アサーション32ビットintなど
- プログラムがするためには、のコンパイラがですパッキング、のクラスなど

static_assert()はSFINAEにはしないことにしてください。のオーバーロード/がな、テンプレートメタプログラミング std::enable_if<> のわりにするべきではありません。されるオーバーロード/がすでにされているときにテンプレートコードでされるがありますが、そののがです。このような、SFINAEにするよりもなエラーメッセージがされることがあります。

オンラインでstatic_assertをむ <https://riptutorial.com/ja/cplusplus/topic/3822/static-assert>

30: std :: any

クラス `std::any` は、のののをれることができるセーフなコンテナをします。

Examples

な

```
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';
```

```
hello world
Wrong type
42
```

オンラインで `std :: any` をむ <https://riptutorial.com/ja/cplusplus/topic/7894/std----any>

31: std :: array

パラメーター

パラメータ	
class T	メンバーのデータをします。
std::size_t N	のメンバーをします。

std::arrayするには、`#include <array>`をして<array>ヘッダをめるがあります。

Examples

std :: arrayの

std::array<T, N>はスカラー、**N**は**T**の

Tがスカラーの、std::arrayはのようになります。

```
// 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 };
```

std::array<T, N>はスカラー、**N**は**T**ののです。

Tがスカラーの、std::arrayはのようになります。

```
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 };
```

アクセス

1. at(pos)

チェックをう_{pos}のへのをします。 _{pos}がコンテナのにないは、 `std::out_of_range`のがスローされます。

さはO1である。

```
#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]

チェックなしで_{pos}にあるへのをします。 _{pos}がコンテナのにない、セグメンテーションエラーがするがあります。このメソッドは、のとのアクセスをし、 `at(pos)` よりもです。

さはO1である。

```
#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>

このメンバは、チェックなしでコンパイル `pos` へのをします。 `pos` がコンテナのにない、セグメンテーションエラーがするがあります。

さは01である。

```

#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()

コンテナののへのをします。のコンテナの `front()` びしはされていません。

さは01である。

コンテナ `c` の、 `c.front()` は `*c.begin()` とです。

```

#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };
}

```

```
int a = arr.front(); // a is now 2

return 0;
}
```

5 back()

コンテナののへのをします。のコンテナで`back()`をびす`back()`はされていません。

さはO1である。

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.back(); // a is now 6

    return 0;
}
```

6 data()

ストレージとしてするへのポインタをします。ポインタは、`range [data(); data() + size())`コンテナがのでも`range [data(); data() + size())`はになですこの、`data()`はできません。

さはO1である。

```
#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;
}
```

のサイズの

Cスタイルのとして`std::array`なの1つは、`size()`メンバーをしてのサイズをできることです

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    cout << arr.size() << endl;
}
```


の

`std::array`はSTLコンテナであり、`vector`のようなコンテナとのベースのforループをできます

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    for (auto i : arr)
        cout << i << '\n';
}
```

にすべてのをする

メンバ`fill()`は、ににをするために`std::array`でできます

```
int main() {

    std::array<int, 3> arr = { 1, 2, 3 };
    // change all elements of the array to 100
    arr.fill(100);

}
```

オンラインで`std :: array`をむ <https://riptutorial.com/ja/cplusplus/topic/2712/std----array>

32: std :: atomics

Examples

タイプ

インスタンスと `std::atomic` テンプレートの `std::atomic` は、アトミックをします。あるスレッドがアトミックオブジェクトにきみをとっているのにスレッドがそれをみんだ、そのはにされていますデータのについてはメモリモデルをしてください

さらに、アトミックオブジェクトへのアクセスは、スレッドのをし、 `std::memory_order` されたアトミックメモリアクセスを `std::memory_order` ます。

`std::atomic` は、の `TriviallyCopyable` type `T`. `std::atomic` インスタンスできます `TriviallyCopyable` type `T`. `std::atomic` はコピーももできません。

ライブラリは、の `std::atomic` テンプレートのをします

1. レイアウト、なデフォルトコンストラクタ、なデストラクタ、およびをサポートしているをいて、 `std::atomic<T>` としてわれる `bool` とその `typedef` のながされています

typedef	フルスペシャライゼーション
<code>std::atomic_bool</code>	<code>std::atomic<bool></code>

2のようなとのtypedef

typedef	フルスペシャライゼーション
<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>
<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>

typedef	フルスペシャライゼーション
std::atomic_ullong	std::atomic<unsigned long long>
std::atomic_char16_t	std::atomic<char16_t>
std::atomic_char32_t	std::atomic<char32_t>
std::atomic_wchar_t	std::atomic<wchar_t>
std::atomic_int8_t	std::atomic<std::int8_t>
std::atomic_uint8_t	std::atomic<std::uint8_t>
std::atomic_int16_t	std::atomic<std::int16_t>
std::atomic_uint16_t	std::atomic<std::uint16_t>
std::atomic_int32_t	std::atomic<std::int32_t>
std::atomic_uint32_t	std::atomic<std::uint32_t>
std::atomic_int64_t	std::atomic<std::int64_t>
std::atomic_uint64_t	std::atomic<std::uint64_t>
std::atomic_int_least8_t	std::atomic<std::int_least8_t>
std::atomic_uint_least8_t	std::atomic<std::uint_least8_t>
std::atomic_int_least16_t	std::atomic<std::int_least16_t>
std::atomic_uint_least16_t	std::atomic<std::uint_least16_t>
std::atomic_int_least32_t	std::atomic<std::int_least32_t>
std::atomic_uint_least32_t	std::atomic<std::uint_least32_t>
std::atomic_int_least64_t	std::atomic<std::int_least64_t>
std::atomic_uint_least64_t	std::atomic<std::uint_least64_t>
std::atomic_int_fast8_t	std::atomic<std::int_fast8_t>
std::atomic_uint_fast8_t	std::atomic<std::uint_fast8_t>
std::atomic_int_fast16_t	std::atomic<std::int_fast16_t>
std::atomic_uint_fast16_t	std::atomic<std::uint_fast16_t>
std::atomic_int_fast32_t	std::atomic<std::int_fast32_t>
std::atomic_uint_fast32_t	std::atomic<std::uint_fast32_t>
std::atomic_int_fast64_t	std::atomic<std::int_fast64_t>
std::atomic_uint_fast64_t	std::atomic<std::uint_fast64_t>
std::atomic_intptr_t	std::atomic<std::intptr_t>
std::atomic_uintptr_t	std::atomic<std::uintptr_t>
std::atomic_size_t	std::atomic<std::size_t>

typedef	フルスペシャライゼーション
std::atomic_ptrdiff_t	std::atomic<std::ptrdiff_t>
std::atomic_intmax_t	std::atomic<std::intmax_t>
std::atomic_uintmax_t	std::atomic<std::uintmax_t>

std :: atomic_intをするな

```
#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
```

オンラインでstd :: atomicsをむ <https://riptutorial.com/ja/cplusplus/topic/7475/std----atomics>

33: std :: forward_list

き

`std::forward_list`は、コンテナのどこからでものをサポートするコンテナです。ランダムアクセスはサポートされていません。これはリンクリストとしてされており、にはCでのとしてオーバーヘッドがありません。`std::list`として、このコンテナはのがないに、よりなストレージをします。

リストのの、およびは、リストののをしているイテレータをにしません。ただし、をするイテレータまたはは、するがリストからされるとになります。`erase_after`をして。`std::forward_list`は `Container`のをたしています。サイズメンバをいて、`==`のさはにです、`AllocatorAwareContainer`と `SequenceContainer`。

Examples

```
#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';
}
```

```
words1: [the, frogurt, is, also, cursed]
```

```
words2: [the, frogurt, is, also, cursed]
words3: [the, frogurt, is, also, cursed]
words4: [Mo, Mo, Mo, Mo, Mo]
```

メソッド

メソッド	
operator=	コンテナにをする
assign	コンテナにをする
get_allocator	けられたアロケータをします。
-----	-----
アクセス	
front	のにアクセスする
-----	-----
イテレータ	
before_begin	ににイテレータをします
cbegin	ににのイテレータをします
begin	イテレータをにします
cbegin	constイテレータをにす
end	イテレータをまでします。
cend	イテレータをまでします。
empty	コンテナがであるかどうかをチェックします。
max_size	のなをします。
clear	をクリアする
insert_after	のろにをする
emplace_after	のろにをみる
erase_after	ののをする

メソッド	
push_front	にをする
emplace_front	にをインプレースでする
pop_front	のをします。
resize	されるのをする
swap	をする
オペレーション	
merge	2つのソートされたリストをマージします。
splice_after	のをforward_listからする
remove	のをたすをする
remove_if	のをたすをする
reverse	のをにする
unique	するをする
sort	をソートする

オンラインでstd :: forward_listをむ <https://riptutorial.com/ja/cplusplus/topic/9703/std----forward-list>

34: std::function びしなをラップする

Examples

ない

```
#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::bind でされる std::function

をつをコールバックするがあるをえてみましょう。 std::function std::bind される std::function は、にすようなになをします。

```
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 を lambda と std::bind でする

```

#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;
}
```

``オーバーヘッド

`std::function` はなオーバーヘッドをきこすがあります。 `std::function` は [value semantics] [1] があるため、されたびしコードをコピーまたはするがあります。しかし、それはのタイプのびしなものをすることができるので、にこれをうためにメモリをにりてるがあります。

`function` のには、さなオブジェクトポインタ、メンバポインタ、にさなのファンクタなどが `function` オブジェクトにされる、いわゆる「オブジェクト」があります。しかし、これは、が `noexcept move constructible` であるにのみします。さらに、C++では、すべてので1つをするはありません。

のをしてください。

```
//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);
}
```

テンプレートパラメータは `SortMyContainer` のソリューションですが、らかのでこれがではない、またはましいとほえません。 `SortMyContainer` は、のびしをえて `pred` をするはありません。しかし、えられたファンクタがなきさであれば、 `pred` はメモリをにりてることができます。

`function` はかをコピー/するがあるためメモリをりてます。 `function` はえられたびしコードのをします。しかし、 `SortMyContainer` はびしなものをするはありません。それはそれをしているだけです。ここでの `function` はです。かもしれませんが、そうでないかもしれません。

にびしコードをするライブラリはありません。だから、わりのがつかるか、オーバーヘッドであることをするがあります。

また、 `function` は、オブジェクトのメモリりてがどこからるかをするなはありません。はい、それはコンストラクタがある `allocator`、しかし[くのは、それらをしくする...か、まったくありません] [2]。

C++ 17

`allocator` をける `function` コンストラクタはもはやそのものではありません。したがって、りてをするはありません。

`function` びすことは、をびすよりもくなります。どのので `function` インスタンスがびしにすることができ、してコール `function`、でなければなりません。 `function` びしのオーバーヘッドは、びし

のオーダーです。

のびしなへの **std :: function** のバインド

```
/*
 * 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);
}

```

```

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

```

を `std::tuple` にする

いくつかのプログラムでは、のあるのびしのためにをすることがあります。

のは、`std::tuple` にされたをつをびすをしています。

```

#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);
}

```

```
}
```

ライブ

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

オンラインでstd :: functionびしなをラップするをむ

<https://riptutorial.com/ja/cplusplus/topic/2294/std---function-びしなをラップする>

35: std :: integer_sequence

き

クラステンプレート `std::integer_sequence<Type, Values...>` は、タイプ `Type` のシーケンスをします。 `Type` は、みみタイプの1つです。これらのシーケンスは、アクセスのをけるクラスまたはテンプレートをするときにされます。ライブラリには、からのシーケンスをする「ファクトリ」もまれています。

Examples

`std::tuple` をすのパラメータにする

`std::tuple<T...>` は、のをすためにできます。えは、のパラメータをらかののキューにするためにできます。このようなタプルをする、そのをびしにするがあります。

```
#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 });
}
```

クラスが `std::get<I>(object)` と `std::tuple_size<T>::value` サポートしているり、それはの `process()` でできます。は、ののにしません。

からなるパラメータパックをする

`std::integer_sequence` は、パラメータパックにできるのシーケンスをしています。そのなは、これらのシーケンスをする「ファクトリ」クラステンプレートです。

```
#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>());
}
```

`print_sequence()` テンプレートは、シーケンスをするときに `std::initializer_list<bool>` して、のをし、の[]をしません。

のインデックスをのコピーにする

カンマのインデックスのパラメータパックをですと、インデックスののコピーがされます。しいことに、`gcc` と `clang` はインデックスががなく、を `clang` とえています `gcc` はインデックスを `void` キヤストしてサイレントにすることができ `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";
}

```

オンラインでstd :: integer_sequenceをむ <https://riptutorial.com/ja/cplusplus/topic/8315/std---integer-sequence>

36: std :: iomanip

Examples

std :: 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;
```

これは、

```
10
      10
1234567890
```

のは、のオフセットをのけるためにそこにあります。

フィールドのをするがあるがあります。は、されたなレイアウトでをするがあります。これは **std::setw** の **std :: setw** をってうことができます。

std::setw ののはの std::setw です。

```
std::setw(int n)
```

ここで、n はするフィールドのさです

std :: setprecision

out << setprecision(n) または in >> setprecision(n) です、ストリームのパラメータをに n にします。このパラメータはであり、これはのしいです。

```
#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
```

std :: setfill

`out << setfill(c)`すると、ストリームのりつぶしが`out << setfill(c)`され`c`。

のりつぶしは、`std::ostream::fill`で`std::ostream::fill`ます。

```
#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
```

std :: setiosflags

`out << setiosflags(mask)`または`in >> setiosflags(mask)`ですと、ストリームのすべてのフォーマットフラグがマスクでされた`in >> setiosflags(mask)`またはされます。

すべての`std::ios_base::fmtflags`

- `dec` - I/Oの10ベースをする
- `oct` - I/Oのために8をする
- `hex` - I/Oの16ベースをする
- `basefield` - `dec|oct|hex|0` マスキングに
- `left` - のにりつぶしを
- `right` - のにりつぶしを
- `internal` - にりつぶしを
- `adjustfield` - `left|right|internal`。 マスキングにちます
- `scientific` - をしてをするか、またはされた
- `fixed` - をしてをするか、または
- `floatfield` - `scientific|fixed|(scientific|fixed)|0`。 マスキングにちます
- `boolalpha` - `bool`をでおよびする
- `showbase` - のベースをすをし、I/Oにインジケータをとする
- `showpoint` - のにでをする
- `showpos` - でないのための`+`をする
- `skipws` - ののにをスキップする
- `unitbuf`にをフラッシュする
- `uppercase` - のでのを`uppercase`きえ`uppercase`

マニピュレータの

```
#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;
}
```

オンラインでstd :: iomanipをむ <https://riptutorial.com/ja/cplusplus/topic/6936/std----iomanip>

37: std::map

- `std::map`または`std::multimap`のいずれかをするには、ヘッダーファイル`<map>`をめるがあります。
- `std::map`と`std::multimap`は、キーのによってソートされたをします。 `std::multimap`、じキーのにしてソートはわれません。
- ない`std::map`と`std::multimap`ことある`std::map` 1がじキーのをしない`std::multimap`ありません。
- マップはバイナリツリーとしてされています。したがって、 `search()`、 `insert()`、 `erase()` は $\Theta \log n$ をします。のをうには、 `std::unordered_map`します。
- `size()`および`empty()`は $\Theta 1$ のさをち、ノードのはこれらのがひされるたびにツリーをらないようにキャッシュされます。

Examples

へのアクセス

`std::map`は (key, value) ペアをとてけります。

の`std::map`のをえてみましょう。

```
std::map < std::string, int > ranking { std::make_pair("stackoverflow", 2),  
                                       std::make_pair("docs-beta", 1) };
```

`std::map`では、のをのようにできます。

```
ranking["stackoverflow"]=2;  
ranking["docs-beta"]=1;
```

のでは、キーの`stackoverflow`がすでにする、そのは2にされます。まだしないは、しいエントリがされます。

`std::map`では、キーにインデックスをけることによって、にアクセスできます。

```
std::cout << ranking[ "stackoverflow" ] << std::endl;
```

マップで`operator[]`をすると、にはされたキーとともにしいがマップにされます。これは、キーがすでにマップにされていても、`const std::map`でできないことをします。このをするには、がするかどうかたとえば、`find()``at()`、または`at()`のをのようにするかどうかをします。

`std::map`は、`at()` アクセスできます。

```
std::cout << ranking.at("stackoverflow") << std::endl;
```

コンテナにされたがまれていない、`at()`は`std::out_of_range`をスローします。

`std::map`と`std::multimap`のコンテナでは、イテレータをしてにアクセスできます。

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"
```

`std::map`または`std::multimap`の

`std::map`と`std::multimap`は、コンマでられたキーとのペアをすることでできます。キーとのペアは、`{key, value}`いずれかによってされるか、`std::make_pair(key, value)`によってにされ
`std::make_pair(key, value)`。`std::map`はキーをさず、カンマはからへするので、のペアはのじキーをつペアできされます。

```
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
```

どちらもイテレータでできます。

```
// 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}

```

の

すべてのをする

```

std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
mmp.clear(); //empty multimap

```

イテレータのけをりてどこかからをする

```

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}

```

のすべてのをする

```

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}

```

されたをつすべてのをキーとしてする

```

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}

```

predをたすをする

```
std::map<int,int> m;
auto it = m.begin();
while (it != m.end())
{
    if (pred(*it))
        it = m.erase(it);
    else
        ++it;
}
```

をする

が `std::map` まだしないのにのみ、を `std::map` することができます。例えば、

```
std::map< std::string, size_t > fruits_count;
```

- キーとのペアは、 `insert()` メンバーをして `std::map` されます。それはとして `pair` をとします

```
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));
```

`insert()` は、イテレータと `bool` からなる `pair` します。

- がした、イテレータはしくされたをし、 `bool` は `true` です。
- じ `key` つかにする、はします。それがこると、はをきこすをし、 `bool` はが `false` です。

とをみわせるには、のメソッドをできます。

```
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
}
```

- 、 `std::map` コンテナは、 `std::map` のにアクセスするえをし、しないはしいものをします。

```
fruits_count["apple"] = 10;
```

ですが、がすでにするかどうかをユーザーがするのをぎます。がない、
`std::map::operator[]` にそれをし、されたできするにデフォルトのコンストラクタでします。

- `insert()` は、ペアのブレースリストをしてのをにするためにできます。このバージョンの `insert` は `void` をします

```
fruits_count.insert({{"apricot", 1}, {"jackfruit", 1}, {"lime", 1}, {"mango", 7}});
```

- `insert()` は、 `value_type` のとをすイテレータをしてをするためにもできます。


```
std::map< std::string, size_t > fruit_list{ {"lemon", 0}, {"olive", 0}, {"plum", 0}};
fruits_count.insert(fruit_list.begin(), fruit_list.end());
```

```
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
    }
}
```

`std::map`はツリーとしてされているため、ののさは $O(\log n)$ です。

C++ 11

`pair`は`make_pair()`と`emplace()`を使ってにすることができます

```
std::map< std::string , int > runs;
runs.emplace("Babe Ruth", 714);
runs.insert(make_pair("Barry Bonds", 762));
```

しいがどこにされるかかっているなら、`emplace_hint()`を使ってイテレータの`hint`をすることができます。 `hint`にしいをできる、はのにできます。それのは、`emplace()`と同じようにします。

```
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);
```

`std::map`または`std::multimap`をりしする

`std::map`または`std::multimap`はのでトラバースできます

```
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
```

`std::map`または`std::multimap`をりしするは、ななをけるために`auto`することをお`std::multimap`しま

すはSOえをしてください。

std::mapまたはstd::multimapです

std::mapまたはstd::multimapでキーをするはいくつかあります。

- キーののイテレータをするには、find()をできます。キーがしなはend()します。

```
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.
```

- エントリがstd::mapまたはstd::multimapするかどうかをべるもう1つのは、count()をして、されたキーにけられているのをえす。std::mapはキーに1つのしかけていないため、count()は0キーがないまたは1するをすことができます。std::multimap、じキーにのがけられているがあるため、count()は1よりきいをします。

```
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;
```

あるがするかどうかだけがになる、findはにはいす。それはあなたのをし、multimaps、にするがつかるとします。

- std::multimap、じキーをつのがするがあります。このをするには、iteratorのとをつstd::pairをすequal_range()をします。キーがしなは、のイテレータはend()します。

```
auto eqr = mmp.equal_range(6);
auto st = eqr.first, en = eqr.second;
for(auto it = st; it != en; ++it){
    std::cout << it->first << ", " << it->second << std::endl;
}

// prints: 6, 5
//         6, 7
```

の

コンテナstd::mapは、マップがであるかどうかにしてtrueまたはfalseをすメンバーempty()があります。メンバーsize()は、std::mapコンテナにされているのをします。

```
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;
}
```

マップの

レギュラーマップ

マップは、キーとのペアをむコンテナです。

```
#include <string>
#include <map>
std::map<std::string, size_t> fruits_count;
```

のでは、 `std::string` はキータイプであり、 `size_t` はです。

キーはマップのインデックスとしてします。キーはでなければならず、するがあります。

- じキーをつのがなは、 `multimap` をしてください
- のタイプでがされていない、またはデフォルトのをオーバーライドするは、のいずれかをします。

```
#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;
```

`StrLess` が2つのキーにして `false` をす、のがなっているもじものとみなされます。

マルチマップ

マルチマップでは、じキーをつのキーとのペアをマップにできます。それのは、そのインターフェイスとはのマップににしています。

```
#include <string>
#include <map>
std::multimap<std::string, size_t> fruits_count;
std::multimap<std::string, size_t, StrLess> fruits_count2;
```

ハッシュマップけられていないマップ

ハッシュマップには、マップとのキーとのペアがされます。しかし、キーにしてをけることはありません。わりに、キーのハッシュをして、なキーとのペアにすばやくアクセスします。

```
#include <string>
#include <unordered_map>
std::unordered_map<std::string, size_t> fruits_count;
```

けられていないマップははですが、はなでされません。たとえば、`unordered_map`すべてのをすると、なでが`unordered_map`ます。

ユーザーをキーとして `std::map` をする

クラスのキーをマップのキーとしてできるようにするには、キーになのは、それが `copiable` で `assignable` です。マップのは、テンプレートの3のおよびコンストラクタがされているはそのによってされます。これはデフォルトでは `std::less<KeyType>` にされています。デフォルトは `<` ですが、デフォルトをするはありません。をくだけですできればオブジェクトとして

```
struct CmpMyType
{
    bool operator()( MyType const& lhs, MyType const& rhs ) const
    {
        // ...
    }
};
```

C++では、"compare"は**ない**でなければなりません。に、`compare(X,X)`はの`X`にして `false` をすがあります。つまり、`CmpMyType()(a, b)`が`true`をす、`CmpMyType()(b, a)`は`false`をさなければならず、が`false`をす、はしいとなされます。

ない

これは、2つのオブジェクトのをするなです。
はのとおりです。

`fx, y`と`fy, x`のが`false`の、2つのオブジェクト`x`と`y`はです。オブジェクトは、それにするによってにすることにしてください。

C++の、されたのオブジェクトが2つあるは、`<`としてのをすがあります。

```
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

/をするは、オブジェクトのタイプににします。

オンラインでstd :: mapをむ <https://riptutorial.com/ja/cplusplus/topic/681/std----map>

38: std :: optional

Examples

き

オプションMaybeともばれますは、コンテンツがするとしがないがあるをすためにされます。これらは、 `std::optional` クラスとしてC++ 17でされてい `std::optional` 。たとえば、のオブジェクト `std::optional<int>` タイプのいくつかのまれていてもよい `int` 、またはそれはをんでいなくてもよいです。

オプションは、しないのあるをすために、またはのあるをすことができないからのりのとしてにされます。

オプションののアプローチ

このをするには、 `std::optional` するが、ポインターをする、センチネルをう、 `pair<bool, T>` をするなど、どれもではないというをするはくあります。

オプションとポインタ

によっては、のオブジェクトまたは `nullptr` へのポインタをして、をすことができます。しかし、これはオブジェクトがすでにするにられ `optional` として、メモリりてにらずにしいオブジェクトをすこともできます。

オプションの Sentinel

なイディオムは、がであることをすためになをすることです。のは0または-1、ポインタのは `nullptr` になります。しかし、これはなのスペースをらしめますな0とな0をすることはできません。そして、くのタイプは、センチネルにしてなをっていません。

オプション VS `std::pair<bool, T>`

のなイディオムは、の1つががをつかどうかをす `bool` であるペアをすることです。

これは、エラーのにはデフォルトでなにしますが、これはいくつかのではありません、のにしてはですがましくありません。 `optional<T>` 、エラーの、もするはありません。

オプションのによるの

C++ 17よりでは、 `nullptr` をつポインタをつことはにがしないことをしていました。これは、にりてられ、ポインタですすでにされているきなオブジェクトのは、れたソリューションです。しかし、ポインタでにりてられたりされることはめったにない `int` ようなさなまたはプリミティブでは

、このはうまくしません。 `std::optional` は、こののになをします。

このでは、 `struct Person` がされています。がペットをつことはですが、ではありません。したがって、 `Person` の `pet` メンバーは `std::optional` ラッパーでされます。

```
#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;
    }
}
```

オプションをしてのをす

C++ 17よりののは、のいずれかののでをしました。

- ノルポインタがされました。
 - えはをびす `Delegate *App::get_delegate()` は、デリゲートをたない `App` インスタンスにして `nullptr` をします。
 - これは、にりてられたオブジェクト、またはポインタによってされ、でされているオブジェクトにしてはいですが、はスタックでりてられ、コピーによってされるさなオブジェクトにはしていません。
- りのののは、をすためにされていました。
 - たとえば、されていない2つの `unsigned shortest_path_distance(Vertex a, Vertex b)` をびすと、このをすゼロがされることがあります。
- は、されたがをつことをすために `bool` とペアになっています。
 - たとえば、ではないを `std::pair<int, bool> parse(const std::string &str)` をびすと、の `int` と `bool` が `false` されたペアがされ `false` 。

このでは、2のペット `Fluffy` と `Furball` が `John` にえられています。 `Person::pet_with_name()` をびして、`John` のペット `Whiskers` をします。 `John` は `Whiskers` というのペットをつていないので、はし、わりに `std::nullopt` がされます。

```

#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;
    }
}

```

オプションでり

```

std::optional<float> divide(float a, float b) {
    if (b!=0.f) return a/b;
    return {};
}

```

ここでは a/b どちらかのをしますが、されていないになります、わりにのオプションをします。

よりなケース

```

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 )`は、コンテナまたは`some_range`を7しいものをします。 `find_if`はでそれをいます。

つからなかったはのオプションか、するはイテレータをむオプションです。

これにより、のことがになります。

```

if (find( vec, 7 )) {
    // code
}

```

あるいは

```

if (auto oit = find( vec, 7 )) {
    vec.erase(*oit);
}

```

イテレータとテストの/をさせることなく

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`はオプションでされているをします。そこにストアがないはをします。

これにより、`null`のオプションをとって、にがなときにデフォルトのをえることができます。このようにすることで、「デフォルト」のは、あるエンジンのにならないようにデフォルトをするのではなく、もでちになにすことができます。

オンラインで`std :: optional`をむ <https://riptutorial.com/ja/cplusplus/topic/2423/std----optional>

39: std :: pair

Examples

ペアのとへのアクセス

ペアでは、2つのオブジェクトを1つのオブジェクトとしてうことができます。ペアはテンプレート `std::make_pair` けをりてにできます。

もう1つののは、ペアをしてでその `first` と `second` をりてることです。

```
#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;
}
```

オペレータの

これらののパラメータは `lhs` と `rhs`

- `operator== lhs` と `rhs` のがしいかどうかを `rhs` ます。りは、`lhs.first == rhs.first` と `lhs.second == rhs.second` が `true` は `true`、そうでないは `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!= lhs` と `rhs` ペアのがしくないかどうかをべます。 `lhs.first != rhs.first` または

`lhs.second != rhs.second` いずれかで `true`、 りは `true` です。 それのは `false` し `false`。

- `operator< lhs.first<rhs.first` は `true` し `true`。 それの、 `rhs.first<lhs.first` が `false` す それのは、 `lhs.second<rhs.second` が `true` し、 そうでないは `false` し `false`。
- `operator<=` **returns** `!(rhs<lhs)`
- `operator>` は `rhs<lhs` します。
- `operator>=` **returns** `!(lhs<rhs)`

ペアのコンテナをしたのこれは、コンテナをソートするがあるため、 `operator<` します。

```
#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)
    }
}
```

オンラインで `std::pair` をむ <https://riptutorial.com/ja/cplusplus/topic/4834/std---pair>

40: std :: set と std :: multiset

き

`set` は、がソートされユニークなコンテナです。 `multiset` はありますが、 `multiset` の、のエレメントがじをつことができます。

これらのでは、C++のさまざまなスタイルがされています。C++ 98コンパイラをしているはしてください。このコードのはできないがあります。

Examples

セットにをする

セットには3つのなるをできます。

- まず、のな。このメソッドは、びしががにしたかどうかをできるようにするペアをします。
- に、がされるのヒントをしてします。そのようににをすることがですが、のをすることはなケースではありません。そのはしてください。ヒントをえるはコンパイラのバージョンによってなります。
- に、ポインタとポインタをしてのをすることができます。のものはにまれ、のものはされま

```
#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;
}

```

はのようになります。

```

# 23 has been inserted!

# 23 already present in set!

Set under test contains:

5
7
12
20
23
30
45

```

マルチセットにをする

セットからのすべてのメソッドもマルチセットにされます。それにもかかわらず、`initializer_list` をするのがあります。

```

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

```

セットのデフォルトのソートの

`set`と`multiset`はデフォルトのメソッドがありますが、によってはそれらをオーバーロードするがあります。

をセットにしているとしましょうが、そのにはのみがまれていることがわかります。デフォルトでは、べえはのであるため、はのべえとしません。 `int`とのソートをするには、`compare`メソッドをオーバーロードするためのファンクタがです。

```
#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;
}
```

はのようになります。

```
### 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

```

のでは、を `std::set` にする3つのなるをつけることができ、それぞれがそのコンテキストです。

デフォルトのソート

これは、キーのテンプレートをしします。くの、キーは `std::less<T>` のためのいデフォルトをしします。このがされていないり、オブジェクトの `operator<` をしします。これは、のコードもいくつかのをしようとするににです。これはコードベースのをにするためです。

このようにコードをくと、キーがされたときにコードをするためのがされます。たとえば、2つのメンバーをむクラスは3つのメンバーをむクラスにされます。クラスの `operator<` をすると、すべてのオカレンスがされます。

このとおり、デフォルトのソートをするのがなデフォルトです。

カスタムソート

をしてオブジェクトをしてカスタムソートをするのは、デフォルトのがしないによくされます。のでは、がをしているためです。それのは、するオブジェクトについてスマートなポインタをするや、のためになるがなにされることがあります `std::pair` と `first` のを。

をする、これはしたソートでなければなりません。にのがされた、のがします。いとして、はデータ `const` メンバー、`const...` のみをうべきです。

ののように、メンバーをたないクラスはとしてよくかけるでしょう。これにより、デフォルトコンストラクタとコピーコンストラクタがされます。のコンストラクターでは、にインスタンスをすることができ、セットが `compare` のコピーをとるので、コピーコンストラクターがです。

ラムダソート

[Lambda](#) は、オブジェクトをくためのいです。これにより、をないにくことができ、コードをよ

りみやすくします。

ラムダのは、ラムダがコンパイルにのをすることです。したがって、じコンパイルcppファイルのコンパイルごとに`decltype(lambda)`がのコンパイル。このため、ヘッダーファイルでされた、オブジェクトとしてオブジェクトをすることをおめします。

このは、`std::set`がのローカスコープでわりにされているときにしばしばしますが、のまたはクラスのメンバーとしてされるときは、オブジェクトがされます。

そののべえオプション

`std::set`の`compare`はテンプレートであるため、[びしなオブジェクト](#)はすべて`compare`としてでき、のはのケースのみです。これらのびしオブジェクトには、のがあります。

- コピーなものでなければなりません
- それらは、キーのの2つのでびしでなければなりません。なはされますが、パフォーマンスをさせるがあるためされません

セットとマルチセットのの

えられたを`std::set`または`std::multiset`するにはいくつかのがあります

キーのののイテレータをするには、`find()`をできます。キーがしないは`end()`します。

```
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);
```

のがされる`count()`くのするは、でされたかをカウントする、`set / multiset`のに`set`、りとすることができるだけ0または1。とじをすると、のようになります。

```
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
```


`std::multiset`、`し`をつのがするがあります。このをするには、`equal_range()`をできます。`iterator`のをつ`std::pair`をします。キーがしない、のイテレータは、された`multiset`をソートするためにされる`compare`メソッドについてもいをします。

```
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'
```

セットからをする

もなは、あなたのセット/マルチセットをのものにリセットしたいのであれば、`clear`をう`clear`です

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

に、`erase`をすることができます。それはとじようないくつかのをします

```
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;
}
```

はのようになります。

```
Set under test contains:
```

```
10
```

```
15
```

```
30
```

これらのメソッドはすべて `multiset` もされます。 `multiset` からをするようにされ、するは、すべてののがされます。

オンラインで `std :: set` と `std :: multiset` をむ <https://riptutorial.com/ja/cplusplus/topic/9005/std----set> と `std----multiset`

41: std :: string

き

は、をすオブジェクトです。の`string`クラスは、テキストやののシーケンスをするに、`char`のなをすることにわって、シンプルででなをします。C++ `string`クラスは`std`のであり、1998にされました。

- //

```
std :: string s;
```

- // *const char *c-string*からの

```
std :: string s "こんにちは";
```

```
std :: string s = "こんにちは";
```

- //コピーコンストラクタをしてする

```
std :: string s1 "Hello";
```

```
std :: string s2s1;
```

- //からする

```
std :: string s1 "Hello";
```

```
std :: string s2s1、 0、 4; // s1の0からs2に4をコピーする
```

- //のバッファからする

```
std :: string s1 "Hello World";
```

```
std :: string s2s1、 5; // s1のの5をs2にコピーする
```

- //フィルコンストラクタをしてする`char`のみ

```
std :: string s5、 'a'; // sにはaaaaaaがまれています
```

- //コンストラクタとイテレータをしてする

```
std :: string s1 "Hello World";
```

```
std :: string s2s1.begin、 s1.begin+ 5; // s1のの5をs2にコピーする
```

`std::string`をするに、のヘッダー `iostream` にはまれていない//オーバーロードがまれるため、ヘッダー `string`をめるがあります。

`nullptr`で`const char *`コンストラクタをすると、のにつながります。

```
std::string oops(nullptr);
std::cout << oops << "\n";
```

`at`のメソッド`at`、`index >= size()`に`std::out_of_range`スローします。

`operator[]`のはしです。すべての、`index > size()`はのをし`index > size()`が、`index == size()`はしません。

C++ 11

1. `const`では、はです。
2. `const`では、が`CharT()`ヌルのへのがされます。

C++ 11

1. `CharT()`ヌルをつへのがされます。
2. このをすることはのです。

わりにするC++ 14は、`"foo"`、することをおめします`"foo"`_sのように、_sあり、[ユーザリテラルサフィックス](#)、`const char* "foo"`するために`std::string "foo"`。

リテラル_sをするには、`std::string_literals`または`std::literals std::string_literals`というをするがあります。

Examples

`std::string::substr`をするには`std::string::substr`をします。このメンバには2つのバリエーションがあります。

のは、されたのをとります。は、`(0, str.length())`ででなければなりません

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(11); // "bar and world!"
```

2は、しいのとをとります。さにかかわらず、はソースのをしてしません

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(15, 3); // "and"
```

あなたもびすことができることにしてください `substr`なしで、このには、のなコピーがされます

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(); // "Hello foo, bar and world!"
```

の

ポジションで

`std::string` をきえるには、 `std::string` から `replace` メソッドをで `std::string` 。

`replace` にはく のなオーバーロードがあります

```
//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!"
```

のをのにきえる

`replace` ののオカレンスだけを `str` with with `replace 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;
}

```

きえのすべてのをwith with strでreplace 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;
}

```

オーバーロードされた+と+=を使ってstd::stringをすることができます。+の

```

std::string hello = "Hello";
std::string world = "world";
std::string helloworld = hello + world; // "Helloworld"

```

+=をする

```

std::string hello = "Hello";
std::string world = "world";
hello += world; // "Helloworld"

```

リテラルをむCをすることもできます。

```

std::string hello = "Hello";
std::string world = "world";
const char *comma = ", ";
std::string newhelloworld = hello + comma + world + "!"; // "Hello, world!"

```

push_back() をって々のcharをプッシュバックすることもできます

```

std::string s = "a, b, ";
s.push_back('c'); // "a, b, c"

```

append() もありappend()。これは+=ています。

```

std::string app = "test and ";
app.append("test"); // "test and test"

```

キャラクターへのアクセス

std::stringからをするにはいくつかのがあり、それぞれになります。

```
std::string str("Hello world!");
```

[n

インデックスnのへのをします。

`std::string::operator[]`はチェックされておらず、をスローしません。びしは、インデックスがのにあることをします。

```
char c = str[6]; // 'w'
```

atn

インデックスnのへのをします。

`std::string::at`は **bounds**をチェックし、インデックスがのにないは `std::out_of_range`をスローします

```
char c = str.at(7); // 'o'
```

C++ 11

がの、これらののどちらも**の**になります。

のへのをします。

```
char c = str.front(); // 'H'
```

バック

のへのをします。

```
char c = str.back(); // 'l'
```

トークン

にもいものからもなものまでリストされています

1. `std::strtok`はでされるもっともなトークンメソッドであり、トークンでデリミタをすることもできますが、のC++では3つのがいます。

- `std::strtok`はの `strings` でにすることはできませんただし、 `strtok_s` など、これをサポートするためにいくつかのがされています
-

じから、`std::strtok`はのスレッドでにすることはできませんこれはされているがあります。 [Visual Studioのはスレッドセーフです](#)

- `std::strtok`をびすと、の`std::string`されるので、`const string s`、`const char*`、またはリテラルではできません。これらを`std::strtok`でトークンしたり、`std::string`のをするがある、をコピーしなければならず、にコピーをするがあります

に、これらのオプションのコストはトークンのりてコストにされますが、もいアルゴリズムがで`std::strtok`のしさがされないは、[でスピンされたソリューション](#)をしてください。

```
// 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);
```

2. `std::istream_iterator`は、ストリームのをりしします。 `std::string`がでられている、`std::strtok`オプションをすることができます。これにより、インライントークンがになり、`const vector<string>`がサポートされ、のいをる

```
// 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>());
```

3. `std::regex_token_iterator`は`std::regex`をとてにトークンします。これは、よりなりのをします。たとえば、りでられていないカンマやはのようになります。

C++ 11

```
// String to tokenize
const std::string str{ "The ,qu\\, ick ,\tbrown, fox" };
const std::regex re{ "\\s*((?:[^\t\\,]|\\\\\\\\.)*?)\\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()
};
```

については、[regex_token_iterator](#)をしてください。

constchar *への

`std::string`データへの`const char*`アクセスをするには、の`c_str()`メンバーをできます。ポインタは、`std::string`オブジェクトがスコープにあり、されていないり、であることにしてください。つまり、オブジェクトにして`const`メソッドだけがびされるがあります。

C++ 17

`data()` メンバーをして、`std::string` オブジェクトのデータをするためにできる `char*` をできます。

C++ 11

な `char*` は、の `&s[0]` アドレスをすることによってもできます。C++ 11では、のヌルがされることがされています。なお、`&s[0]` としてもよく、される `s` である、`&s.front()` である `s` です。

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();
```

のの

やのをつけるには、`std::string::find` うことができ `std::string::find`。ののののをします。するものがつからない、は `std::string::npos` します

```
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";
```

つかった21

のは、のによってさらにされます。

```
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
```

これらのをすると、のからをしたり、にないをしたりすることができます。にをします。

```
std::string str = "dog dog cat cat";
std::cout << "Found at position: " << str.find_last_of("gzx") << '\n';
```

つかった6

のはををするのではなく、にまれるをすることにしてください。この、`6`の、`g`がしましたのはわかりませんでした。

1ののトリミング

このでは、`<algorithm>`、`<locale>`、`<utility>`というヘッダがです。

C++ 11

シーケンスまたはをトリミングするとは、のにするすべてのおよびのまたはをすることをします。のは、をすることをわないたため、のをトリムし、のをトリムします。のは`std::basic_string` えば`std::string`と`std::wstring` のすべてのでし、またってシーケンスコンテナ `std::vector`や`std::list` でもすることにしてください。

```
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);
}
```

のをトリミングするには、としないのをつけてそこからするがあります。

```
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;
}
```

のをトリムするには、のをつけてそのとしないものをしし、そこまでするがあります。

```
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;
}
```

`std::string`を`std::isspace()` するためにをするには、`std::isspace()` をとしてできます。

```

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); });
}

```

に、 `std::wstring` などのために `std::wstring` `std::iswspace()` をできます。

トリムされたコピーであるしいシーケンスをするは、のをできます。

```

template <typename Sequence, typename Pred>
Sequence trim_copy(Sequence seq, Pred pred) { // NOTE: passing seq by value
    trim(seq, pred);
    return seq;
}

```

2つの `std::string` は、`==`、`!=`、`<`、`<=`、`>`、`>=` をってにできます。

```

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);

```

これらのはすべて、になる `std::string::compare()` メソッドをしてをし、なブールをします。これらのは、のになく、のようにされるがあります。

- `==`

`str1.length() == str2.length()` とペアがするは `true` をし、そうでないは `false` し `false`。

- `!=`

`str1.length() != str2.length()` または1ペアがしないは `true` し、そうでないは `false` し `false`。

- `<` または `>`

のなるペアをし、それらをしてブールをします。

- `<=` または `>=`

のなるペアをし、それらをしてブールをします。

ペアというのは、じののにするをします。を`str2`にするために、2つのが`str1`および`str2`であり、そのさがそれぞれ`n`および`m`ある、ののペアは、それぞれ`str1[i]`および`str2[i]`をし、 $i = 0, 1, 2, \dots, \max n, m$ である。するがしない i の、すなわち、 i が`n`または`m`で`m`、それはもいとみなされる。

に、`<`をするをします。

```
std::string str1 = "Barr";
std::string str2 = "Bar";

assert(str2 < str1);
```

はのとおりで。

1. のをすると、`'B' == 'B'` - します。
2. 2の`'a' == 'a'`する - する。
3. 3の`'r' == 'r'`する - する。
4. `str1`はまだをとっていますが、`str2`はいくされています。したがって、`str2 < str1`。

std::wstringへの

C++では、のシーケンスは、`std::basic_string`クラスをネイティブタイプであることによってされます。ライブラリでされている2つのコレクションは、`std::string`と`std::wstring`です。

- `std::string`は`char`のでされています
- `std::wstring`は`wchar_t`のでされています

2つのタイプをするには、`wstring_convert`し`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);
```

ユーザビリティやをさせるために、をするをすることができます。

```
#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);
}
```

サンプル

```
std::wstring a_wide_string = to_wstring("Hello World!");
```

かに `std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes("Hello World!")` よりもみやすくなります。

`char` と `wchar_t` はエンコーディングをするものではなく、サイズをバイトでしていないことにしてください。たとえば、`wchar_t` は2バイトのデータとしてにされており、Windows 2000 以前のバージョンでは UCS-2 で UTF-16 でエンコードされたデータを、UTF-32 でエンコードされた4バイトのデータ Linux。これは、C++ 11 でされた新しいタイプの `char16_t` および `char32_t` とはなり、UTF16 または UTF32 の "" または よりにはコードポイント をするのになきさであることがされています。

std::string_view クラスの

C++ 17

C++ 17 では、`std::string_view` されています。これは `const char` のであり、ポインタのペアまたはポインタとサイズのいずれかとしてできます。これはなデータをとするのためのれたパラメータです。C++ 17 よりには、このために3つのオプションがありました。

```
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
```

これらのすべてをのものにきえることができます。

```
void foo(std::string_view s);           // post-C++17, single argument, tighter coupling
                                       // zero copies regardless of how caller is storing
                                       // the data
```

`std::string_view`はそのデータをできないことにしてください。

`string_view`はなコピーをけたいにです。

これは`std::string`がうのなサブセットをしますが、のいくつかはなつたをします

```
std::string str = "lllloooonnnngggg sssstttrrrriinnnggg"; //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';
```

をループする

C++ 11

`std::string`はイテレータをサポートしているので、それぞれのをりしするためには、ベースのループをうことができます

```
std::string str = "Hello World!";
for (auto c : str)
    std::cout << c;
```

「な」 `for`ループをすると、すべてのをループできます。

```
std::string str = "Hello World!";
for (std::size_t i = 0; i < str.length(); ++i)
    std::cout << str[i];
```

への

をむ`std::string`は、をしてまたはにできます。

これらのはすべて、のがするとすぐにのをするので、`"123abc"`は`123`にされます。

`std::ato*`は、Cスタイルのをまたはにします。

```
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());
```

ただし、これらののは、のにしたに₀をすため、しないでください。たとえば、が "0" の、₀がなになるがあるため、がにしたかどうかをすることはできないため、これはいいことです。

よりしい `std::stoi` は、`std::string` をまたはにし、をできなかったはをスローします。であれば、これらのをするがあります。

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);
```

さらに、これらののは、`std::ato*` ファミリとはなり、8 と 16 のもします。2 のパラメータは、ののされていないここではしていませんへのポインタで、3 のパラメータはするです。₀ は ₀ まると 16 _{0x} または _{0X} まるので、そののはするです

```
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
```

エンコーディングの

C++ 11 ではエンコーディングのがで、ほとんどのコンパイラは `<codecvt>` と `<locale>` ヘッダをしてクロスプラットフォームででき `<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;
}

```

Visual Studioの2015は、これらののためのサポートをしますが、ことをえ**バグ**らのライブラリのでは、のためにのテンプレートをするが`wstring_convert`をうとき`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::ul6string& 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::ul6string& 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());
}

```

がののかどうかをする

C ++ 14

C ++ 14では、これは`std::mismatch`によってにされ、ののペアを2つのからします。

```

std::string prefix = "foo";
std::string string = "foobar";

bool isPrefix = std::mismatch(prefix.begin(), prefix.end(),
    string.begin(), string.end()).first == prefix.end();

```

`mismatch()` とのバージョンはC ++ 14よりにしていましたが、2のが2ののほうがいにはありません。

C ++ 14

私たちはまだ `std::mismatch()` のとバージョンをうことができますが、にのが2のとじきさであることをするがあります

```
bool isPrefix = prefix.size() <= string.size() &&
    std::mismatch(prefix.begin(), prefix.end(),
        string.begin(), string.end()).first == prefix.end();
```

C++ 17

`std::string_view` では、りてのオーバーヘッドやコピーをすることなく、をうことができます。

```
bool isPrefix(std::string_view prefix, std::string_view full)
{
    return prefix == full.substr(0, prefix.size());
}
```

`std::string` にする

`std::ostringstream` オブジェクトをストリーム `<< std::ostringstream` オブジェクトにし、
`std::ostringstream` を `std::ostringstream` にすることによって、ストリーミングなをにすることができます `std::string`。

`int` の

```
#include <sstream>

int main()
{
    int val = 4;
    std::ostringstream str;
    str << val;
    std::string converted = str.str();
    return 0;
}
```

のをくと、な

```
template<class T>
std::string toString(const T& x)
{
    std::ostringstream ss;
    ss << x;
    return ss.str();
}
```

パフォーマンスはなコードにはしていません。

にじて、ユーザーのクラスでストリームをできます。

```
std::ostream operator<< (std::ostream& out, const A& a)
{
    // write a string representation of a to out
}
```

```
    return out;
}
```

C ++ 11

ストリームとはに、C ++ 11、すべてののにしてオーバーロードされた`std::to_string` および `std::to_wstring` をして、そのパラメータのをすこともできます。

```
std::string s = to_string(0x12f3); // after this the string s contains "4851"
```

オンラインで`std :: string`をむ <https://riptutorial.com/ja/cplusplus/topic/488/std---string>

42: std :: unordered_mapの

き

std :: unordered_mapはなるのコンテナです。キーとそのマップでします。のりにキーは、マップでをつのにちます。マッピングされたは、キーにけられたコンテンツにぎません。このキーとマップのデータは、されたデータまたはユーザーのいずれでもかまいません。

がむにつれて、けられていないマップのはソートにされません。それらはハッシュによってされるため、けられていないマップのには、O1だけでをするなどのくがあります。のマップコンテナよりもです。このでは、[]としてするのがにであるため、マップされたにアクセスするのにちます。

Examples

と

すでにべたように、のタイプのけられていないマップをすることができます。とのfirstというのけられていないマップを試みましょう。

```
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
```

いくつかの

```
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
```

オンラインでstd :: unordered_mapのをむ <https://riptutorial.com/ja/cplusplus/topic/10540/std---unordered-map>の

43: std::variant

バリエントは、の`union`にわるものです。これはされ、どのかをっており、なにそのオブジェクトをにします。

それはほとんどしてではありませんそのコンテンツをきえてにつてくることができないコーナーケースでのみ、それはのにわるかどうかです。

これは`std::tuple`ようにし、`std::optional`とてい`std::optional`。

`std::get`と`std::get_if`は、はいえです。しいえはは`std::visit`であり、そこではあらゆるにできます。あなたのをさせる`visit`、`visit`よりう`visit`にのシーケンスをするのではなく、`if constexpr`を`visit`でできるかどうかをします。

Examples

な`::`バリエントの

これにより、`int`または`string`できるバリエントタグきがされ`string`。

```
std::variant< int, std::string > var;
```

いずれかののいずれかをすることが出来ます

```
var = "hello"s;
```

そして、`std::visit`コンテンツにアクセスできます

```
// Prints "hello\n":
visit( [](auto&& e) {
    std::cout << e << '\n';
}, var );
```

ラムダまたはのオブジェクトをすことによって、

たちがどんなのものかかっているなら、それをすることが出来ます

```
auto str = std::get<std::string>(var);
```

しかし、それがっていると、これはげてられます。 `get_if`

```
auto* str = std::get_if<std::string>(&var);
```

っているとわれるは`nullptr`します。

バリエントは、ダイナミックメモリリテをしませんただし、まれているによってりてられるは。バリエントの1つのだけがそこにされており、まれにりてにがし、バックアウトするながない、バリエントがになることがあります。

バリエントをすると、のタイプを1つのにかつにできます。らはにスマートな、なunionです。

メソッドポインタの

これはなです。

タイプののためにバリエントをすることができます。

```
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)
            );
        };
    };
};
```

これは、にVariantをつoperator->*をオーバーロードするをします。

```
// 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)... );
};
```

では、それぞれにprintメソッドをつ2つのがありprint

```
struct A {
    void print( std::ostream& os ) const {
        os << "A";
    }
};

struct B {
    void print( std::ostream& os ) const {
```

```

    os << "B";
}
};

```

それらはのタイプであることにしてください。私たちはできる

```

std::variant<A,B> var = A{};

(var->*print)(std::cout);

```

たちのためにA::print(std::cout)にびしをA::print(std::cout)ます。わりにB{}でvarをすると、B::print(std::cout)デイスパッチされます。

しいタイプCをした

```

struct C {};

```

に

```

std::variant<A,B,C> var = A{};
(var->*print)(std::cout);

```

C.print(std::cout)メソッドがないため、コンパイルにします。

をすることで、フリーのprintをしてすることができます。おそらく、printメソッドでif constexprをすることです。

ライブのでは、してboost::variantわりにstd::variant。

`std :: variant`の

これはりてをとしていません。

```

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}

```

オンラインでstd :: variantをもむ <https://riptutorial.com/ja/cplusplus/topic/5239/std----variant>

44: std :: vector

き

ベクトルは、にされるをつです。ベクトルのには、のと同じようににアクセスすることができ、ベクトルのサイズをにできるがあります。

のでは、ベクトルデータははにりてられたメモリにされるため、わずかなオーバーヘッドがです。にC-arraysとstd::arrayは、されたになストレージをするためオーバーヘッドはありません。

std::vectorするには、`#include <vector>`をして<vector>ヘッダをめるがあります。

std::vectorはフリースタアにしてされます。std::vector<std::vector<int> >がstd::vector<std::vector<int> >ようにネストされている、ベクトルのはしていますが、ベクトルはフリースタアにのバッファをりてます。

Examples

std :: vectorの

std::vectorは、にいくつかのことができます

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}
```

あるベクトルは、いくつかのでのコンテナからできます。

v2データをコピーするのをベクターのみからコピーします。

```
std::vector<int> v(v2);
std::vector<int> v = v2;
```

C ++ 11

v2からデータをするのをベクトルのみからする

```
std::vector<int> v(std::move(v2));
```

```
std::vector<int> v = std::move(v2);
```

をvコピーするイテレーターcopy-construction

```
// 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

をvする、std::make_move_iteratorをしたイテレータの

```
// 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()));
```

assign()メンバーのけをりて、std::vectorはにできます

```
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}
```

をする

ベクトルののにコピー/によってをする

```
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

をそのですることによって、ベクトルののにをする


```
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.
```

パフォーマンスのから、`std::vector`は`push_front()`メンバがないことにしてください。にを
と、ベクトルののがすべてします。コンテナののにをを、わりに`std::list`または`std::deque`を
することができます。

ベクトルののにを

```
std::vector<int> v{ 1, 2, 3 };
v.insert(v.begin(), 9); // v now contains {9, 1, 2, 3}
```

C++ 11

をそのですることによって、ベクトルののにを

```
std::vector<int> v{ 1, 2, 3 };
v.emplace(v.begin()+1, 9); // v now contains {1, 9, 2, 3}
```

ベクターののにのベクターを

```
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
```

ベクトルののにを

```
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
```

のベクトルサイズがにわかっているは、のをに`reserve()`して、のりてをします **ベクトルサイ
ズとを**。

```
std::vector<int> v;
v.reserve(100);
for(int i = 0; i < 100; ++i)
    v.emplace_back(i);
```

このは`resize()`をびすことをえないでください。そうしないと、うっかり200のをつベクトルがさ
れ、の100だけがしたになります。

りし:: **vector**

いくつかの `std::vector` できます。のセクションについて、`v` はのようにされます。

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



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";
}
```



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";
}

```

をさせるためのをみるみみはありませんが、これをするのはです。 `begin()` と `end()` をしてイテレータをし、ラッパーオブジェクトでこれをシミュレートすると、ながられます。

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";
    }
}

```

const

C++ 11、`cbegin()` と `cend()` メソッドは、ベクトルが `const` であっても、ベクトルのをすることをにします。イテレータをすると、ベクトルのをみることはできますが、`const` のをするのにです。

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`はこれをのりしにします

```

for (auto const& e : std::as_const(v)) {
    std::cout << e << '\n';
}

```

これは、のバージョンのC++でするのはです

C++ 14

```

template <class T>
constexpr std::add_const_t<T>& as_const(T& t) noexcept {
    return t;
}

```

にする

クラス`std::vector`は、ににりてられたをするクラスであるため、[ここ](#)でしたのとじがC++ベクタにされます。のにうと、ベクトルのにインデックスでアクセスするほうがはるかにです。もちろん、ベクトルへのアクセスもにキャッシュにをきますが、にもされてきたとして、[ここ](#)と[ここ](#)、をのためのパフォーマンスのい`std::vector`のにべてできる。したがって、Cでののののじは、C++の`std::vector`もてはまります。

へのアクセス

`std::vector`にアクセスするなは2つあります

- インデックスベースのアクセス
- [イテレータ](#)

インデックスベースのアクセス

これは、え`[]`、またはメンバ`at()`どちらかですることができます。

とも`std::vector`のへのをします`vector<bool>`のをき`vector<bool>`。これにより、みみとができますベクトルが`const`でない。

`[]`と`at()`というでなる`[]`、のチェックをすることがされていない`at()`はありません。 `index < 0`または`index >= size`へのアクセスは`[]`にするので、 `at()`は`std::out_of_range`をスローします。

のでは、わかりやすくするためにC++ 11スタイルのをしていますが、はすべてのバージョンでできますC++ 11とマークされていない。

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
```

`at()`メソッドはチェックをし、をスローすることができるため、`[]`よりもくなります。これは、のセマンティクスがインデックスがにあることをする`[]`コードをします。いずれにしても、ベクトルへのアクセスはのにわれます。つまり、ベクトルのへのアクセスは、2の、3のなどをアクセスするのと同じコストをちます。

たとえば、このループをえてみましょう

```
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] = 1;
}
```

ここでは、インデックスことをっている`i`ににあるので、ことをするためにCPUサイクルのだろう`i`すべてのびしのためのにある`operator[]`。

`front()`と`back()`メンバーは、それぞれベクトルののとのへのをにします。これらのはにされ、アクセッサは`[]`をしたよりもみやすくなり`[]`。

```
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}
```

のベクトルにして`front()`または`back()`をびすことはのです。 `front()`または`back()`びす`front()`、`empty()`メンバーコンテナがであるかどうかをべる`empty()`をしてコンテナがでないことをするがあります。のベクトルをテストするために'empty'をするなはのとおりです

```
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;
}

```

のでは、1から10までのシーケンスをつベクトルをします。に、ベクトルがになるまで 'empty' をしてのをぐためにベクトルのをポップします。に、ベクトルののがされ、ユーザーにされます。

C ++ 11

`data()` メソッドは `std::vector` がそのをにするためにするのメモリへのポインタをします。これは、Cスタイルのがなレガシーコードにベクターデータをすときにもよくされます。

```

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

C ++ 11のに、`data()` メソッドをシミュレートするには、`front()` をびしてりのアドレスをします。

```

std::vector<int> v(4);
int* ptr = &(v.front()); // or &v[0]

```

これは、ベクタのが `operator&` オーバーライドしないとして、ベクタはにそのをしたメモリにすることがされるためです。もしそうなら、pre-C ++ 11で `std::addressof` をするがあります。また、ベクトルがではないとします。

イテレータ

イテレーターについては、「`std::vector` りしする」のとのイテレーターでしくしています。つまり、ベクトルのへのポインタとにします。

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
```

`std::vector<T>`のイテレータはに `T*` `s`であることがしていますが、ほとんどのライブラリはこれをしてしません。これをわないと、エラーメッセージがされ、のないうコードがキャッチされ、リリースビルドのデバッグチェックでイテレータをするためにできます。に、リリースビルドでは、になるポインタをラップするクラスがされます。

アクセスのために、ベクトルのへのまたはポインタをすることができます。のへのこれらのやポインタ `vector` をし、あなたが/またはののをしないり、アクセスがされたまま `vector`、またはあなたがと `vector` がします。イテレータをにするルールとじです。

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.
```

std :: vectorをCとしてする

`std::vector`をCとしてするにはいくつかのがありますたとえば、Cライブラリとののため。これは、ベクトルのがしてされるためにです。

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data();
```

のC ++をにづくソリューションとはに、メンバー `.data()` はのベクトルにもされるがあります。

C ++ 11のに、ベクトルのののアドレスをってのポインタをします。ベクトルがでない、これらののメソッドはがあります。

```
int* p = &v[0];           // combine subscript operator and 0 literal

int* p = &v.front();      // explicitly reference the first element
```

ベクトルがの、 `v[0]` および `v.front()` はされておらず、できません。

ベクタのデータのベースアドレスをするとき、 `push_back`、 `resize` などのくのがベクタのデータメモリをしてのデータポインタをにすることができることにしてください。えは

```
std::vector<int> v;
int* p = v.data();
v.resize(42);          // internal memory location changed; value of p is now invalid
```

イテレータ/ポインタの

`std::vector` のイテレータとポインタはになるがありますが、のをするのみです。なイテレータ/ポインタをすると、のになります。

イテレータ/ポインタをにするには、

- `vector` の `capacity` をするは、すべてのイテレータ/ポインタをにします。

```
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.
```

- をやさないは、とそれをぎたをすイテレータ/ポインタをにします。これには `end` イテレータがまれます

```
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.
```

- は、されたをすイテレータ/ポインタとされたをえたをにします。これには `end` イテレータがまれます


```
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=` `copy`、`move`、または `otherwise` と `clear()` は、ベクトルをすすべてのイテレータ/ポインタをにします。

の

のをする

```
std::vector<int> v{ 1, 2, 3 };
v.pop_back(); // v becomes {1, 2}
```

すべてのをする

```
std::vector<int> v{ 1, 2, 3 };
v.clear(); // v becomes an empty vector
```

インデックスによるの

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 3); // v becomes {1, 2, 3, 5, 6}
```

のではないをする `vector`、されたをえるすべてののは、ギャップをめるためにコピーまたはするがあります。のおよび `std::list` をしてください。

のすべてのをする

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 1, v.begin() + 5); // v becomes {1, 6}
```

のではベクトルののはされず、サイズのみがされます。 `ベクターサイズと` をしてください。

あるのをする `erase` メソッドは、しばしば `erase-remove` イディオムとしてされます。これは、のものである `std::remove` ベクトルのにし、いくつかのを、に `erase` チョップにそれらをオフに。これは、されたセグメントののすべてののがしいにされなければならないため、ベクトルののインデックスよりもさいインデックスの、なです。コンテナののをにするスピードクリティカルなアプリケーションについては、 `std::list` をしてください。

でをする

```
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}
```

でをする

```
// 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}
```

のをせずに、ラムダによるの

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()
);
```

ループからによるの

```
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
}
```

のには `it` ないのですが、ループでりしするは、のをするををするがあります。よりなとして `remove_if` をしてください。

ループからのによるの

```
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;
}
```

のループのいくつかのにしてください。

- いくつかのをすイテレータを `it` と、メソッドの `base` は、じをすリバーイテレータをえま
す。
- `vector::erase(iterator)` は、 `vector::erase(iterator)` すをし、えられたのにくにイテレータを
します。
- `reverse_iterator::reverse_iterator(iterator)` 、イテレータからイテレータをします。

、にラインれて `it = rev_itr(v.erase(it.base()))` うリバーイテレータを `it` している、 `v` がすを
します。られたイテレータを `it` をり、そこからをし、そしてにりて `it` 。

`v.clear()` をしてすべてのをしても、メモリはされません `capacity()` ベクトルの `capacity()` はされま
せん。スペースをするには、をします。

```
std::vector<int>().swap(v);
```

C++ 11

`shrink_to_fit()` はのベクトルのをします

```
v.shrink_to_fit();
```

`shrink_to_fit` はにスペースをすることをするものではありませんが、のではほとんどありません
。

std::vector のをつける

`<algorithm>` ヘッダでされた `std::find` は、 `std::vector` のをつけるためにできます。

`std::find` は、 `operator==` をしてがしいか `operator==` をします。このとしいをするののにイテレータ
をします。

のがつからない、 `std::find` は `std::vector::end` または `vector` が `const` は `std::vector::end` `std::find`
します。

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)
```

きなベクトルでくのをするがあるは、 [binary_search](#) アルゴリズムをするに、まずベクトルをソートすることをして [binary_search](#)。

をたすベクトルののをつけるには、 `std::find_if` ことができます。 `std::find` えられた2つのパラメータにえて、 `std::find_if` は、へのオブジェクトまたはポインタである3をくれます。はコンテナののをとしてけり、コンテナをせずに `bool` になをすがあります。

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
```

を**std::vector**にする

は**std::begin**と**std::end**を使ってに**std::vector**できます

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 + argc);
}
```

C++ 11の**initializer_list** <>を使って、ベクトルをにすることもできます

```
initializer_list<int> arr = { 1,2,3,4,5 };
vector<int> vec1 {arr};

for (auto & i : vec1)
    cout << i << endl;
```

ベクターにくの、にくのルールへの

セクション23.3.7は、**bool**をパックすることによってスペースをする**vector<bool>**がされるようにします。ビットはC++でアドレスではないので、これは**vector**いくつかのが**vector<bool>**かいていないことをします。

- されたデータはであるはないので、**bool**をとするC APIには**vector<bool>**すことはできません。
- **at()**、**operator []**、イテレータのは**bool**へのをしません。むしろ、をオーバーロードすることによって**bool**へののをにシミュレートするオブジェクトをします。として、のコードは、イテレータのがをさないため、**std::vector<bool>**にしてではないがあります。

C++ 11

```
std::vector<bool> v = {true, false};
for (auto &b: v) { } // error
```

に、`bool&`をするは、`vector<bool>`さ`at()` operator `[]`または`at()`の、またはそののをしたとともにすることはできません。

```
void f(bool& b);
f(v[0]);           // error
f(*v.begin());     // error
```

`std::vector<bool>`のは、コンパイラとアーキテクチャのにしています。は、`n`ブールをメモリのもいアドレスなセクションにパックすることによってされます。ここで、`n`はアドレスなメモリのビットのサイズです。のシステムでは、これは1バイトまたは8ビットです。つまり、1バイトに8つのブールをできます。これは、1ブールが1バイトのメモリにされるのよりもされています。

のは、のされた`vector<bool>`と々のバイトのなビットのをしています。これはすべてのアーキテクチャでにするとはいりません。しかし、これはをするいです。のでは、バイトは`[x、 x、 x、 x、 x、 x、 x、 x]`としてされます。

な `std::vector<char>` 8つのブールをする

C++ 11

```
std::vector<char> trad_vect = {true, false, false, false, true, false, true, true};
```

ビットの

```
[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]
```

Specialized `std::vector<bool>` 8つのブールをする

C++ 11

```
std::vector<bool> optimized_vect = {true, false, false, false, true, false, true, true};
```

ビットの

```
[1,0,0,0,1,0,1,1]
```

のでは、`std::vector<bool>`のバージョンでは、8つのブールが8バイトのメモリをめていましたが、`std::vector<bool>`のバージョンでは、メモリ。これはメモリのなです。`vector<bool>`をCスタイルのAPIにすがあるは、メモリとパフォーマンスがにさらされているは、をにコピーするか、APIをするよりいをつけるがあります。

ベクトルサイズと

ベクトルサイズはにベクトルのです。

1. のベクトルサイズは、 `size()` メンバによってされます。な `empty()` は、 `size` が0のに `true` し `true` 。

```
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. デフォルトのされたベクトルは、サイズ0であります

```
vector<int> v; // size is 0
cout << v.size() << endl; // prints 0
```

3. ベクトルに `N` をすると、 サイズが `N` えますたとえば、 `push_back()` 、 `insert()` または `resize()` による。
4. ベクトルから `N` をすると、 サイズが `N` します `pop_back()` 、 `erase()` または `clear()` 。
5. `Vector` のサイズにはのがありますが、するに `RAM` がするがあります。

```
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
```

よくあるい サイズは `int` ずしもまたはは `int` はありません

```
// !!!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] );
}
```

ベクトルのはサイズとはなりません。 **size** はベクトルののですが、メモリをりて/するのはになります。あまりにもきいサイズのりてがすぎることがあるため、これはです。

1. のベクトルのは、 `capacity()` メンバによっていわせられます。 はにサイズです。

```
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. `reserve( N )` によってで `reserve( N )` できますベクトルを `N` します。

```
// !!!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. なが `shrink_to_fit()` によってされることをすることができますしかし、はあなたにうはあり
ません。これは、みメモリをするのにです。

```

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)

```

Vectorは、にをにします。をすると、するがあります。は、に2または1.5をするのがきですはな
になりますが、のためではありません。、ベクトルはにしません。えは

```

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

```

ベクトルの

1つの `std::vector` は、メンバー `insert()` をしての `std::vector` にすることができます。

```

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());

```

しかし、では、 `insert()` えられたイテレーターがオブジェクトのとじにあってはならないという
ことをしているため、このソリューションはベクトルをにしようとするとはします。

c++ 11

ベクトルのメンバをうわりに、 `std::begin()` と `std::end()` をうことができます

```

a.insert(std::end(a), std::begin(b), std::end(b));

```

これはよりなです。たとえば、 `b` もであるがあるからです。ただし、このソリューションでは、

ベクターをそれにすることはできません。

ベクトルののがでないは、ベクトルのをすると、なコピーがされます。

```
if (b.size() < a.size())
    a.insert(a.end(), b.begin(), b.end());
else
    b.insert(b.end(), a.begin(), a.end());
```

ベクトルのをらす

`std::vector` はにじてににをやしますが、をしたはをらすことはありません。

```
// 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)
```

をらすために、ベクトルのをしいベクトルにコピーすることができます。しいベクトルは、のベクトルのすべてのををするのになをちます。のベクトルのサイズがであった、しいベクトルのはであるがい。に、のベクトルをなベクトルとして、されたをすることができます。

```
std::vector<int> (v).swap(v);
```

C++ 11

C++ 11では、`shrink_to_fit()` メンバをしてのをすることができます。

```
v.shrink_to_fit();
```

`shrink_to_fit()` メンバはリクエストであり、をらすことはできません。

のためのソートみベクトルの

`<algorithm>` ヘッダは、ソートされたベクトルをうためのくのなをします。

ソートされたベクトルをするためのなは、されたが<することです。

ソートされていないベクトルは `std::sort()` をってソートすることができます

```
std::vector<int> v;
// add some code here to fill v with some elements
std::sort(v.begin(), v.end());
```

ソートされたベクトルは、`std::lower_bound()` を使ってなをにします。 `std::find()` とはなり、これはベクトルにしてなバイナリをいます。は、ソートされたにしてなしかられないことです。

```
// 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!
}
```

されたがベクトルのでない、 `std::lower_bound()` は、されたよりきいのにイテレータをします。このるいによって、にソートされたベクトルのしいにしいをすることができます

```
int const new_element = 33;
v.insert(std::lower_bound(v.begin(), v.end(), new_element), new_element);
```

にたくさんのををするがあるは、にすべてのにして `push_back()` をびし、すべてのがされたら `std::sort()` をびすがです。この、べえのコストは、ベクトルのにしいををするコストをし、ではなくすことができます。

ベクトルにじののがまれている、 `std::lower_bound()` はされたののにイテレータをそうとします。しかし、されたのののろにしいををするがあるは、 `std::upper_bound()` をして、のをなくするがあります。

```
v.insert(std::upper_bound(v.begin(), v.end(), new_element), new_element);
```

とのイテレータがなは、 `std::equal_range()` をして、1のびしでをにできます。

```
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;
```

ソートされたベクトルにがするかどうかをテストするには、 `std::binary_search()` します。

```
bool exists = std::binary_search(v.begin(), v.end(), value_to_find);
```

きなベクトルをす

C++ 11

C++ 11では、コンパイラはされるローカルからにするがあります。さらに、くのコンパイラはくの **コピー-エリジョン** をし、そのをにすることができます。その、にできるきなオブジェクトをすがなくなりました。

```
#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

C++ 11よりのバージョンでは、ほとんどのコンパイラですでにcopy elisionがされされていました。しかし、このをしていないコンパイラバージョンでコンパイルするがあるレガシーコードまたはコードでは、セマンティクスがしないため、なコピーをぐためとしてされるベクタをつけることができます。

```
#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;
}
```

ベクトルのとのとそれぞれのインデックスをつける

ベクトルにされているまたはのをつけるには、それぞれ `std::max_element` および `std::min_element` メ

ソッドをできます。これらのメソッドは<algorithm>ヘッダでされています。いくつかのがのとし、メソッドはイテレータをのそのようにします。ベクトルのはv.end()をします。

```
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';
```

maxElementIndex3、maxElement10
minElementIndex1、minElement2

C++ 11

ベクトルのは、<algorithm>ヘッダーでされているstd::minmax_elementメソッドをしてにできます。

```
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';
```

2
10

をった

ベクトルはベクトルのベクトルとしてすることによって2Dとしてできます。

セルが0としてされた34のは、のようになります。

```
std::vector<std::vector<int> > matrix(3, std::vector<int>(4));
```

C++ 11

リストをしてするは、ベクトルのにています。

```
std::vector<std::vector<int>> matrix = { {0,1,2,3},
                                         {4,5,6,7},
                                         {8,9,10,11}
};
```

このようなベクトルのは、2Dとにアクセスできます

```
int var = matrix[0][2];
```

をすることは、ベクトルにしていますが、なががあります。

```
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;
    }
}
```

ベクトルのベクトルはをするなですが、もではありません。々のベクトルはメモリのりにし、データはキャッシュにしています。

また、なでは、すべてののさはじでなければなりませんこれはベクトルのベクトルのではありません。のは、エラーのとなるがあります。

オンラインでstd :: vectorをむ <https://riptutorial.com/ja/cplusplus/topic/511/std----vector>

45: typedef とのエイリアス

き

typedef および C ++ 11 using キーワードはのにしいをえるためにすることができます。

- typedef タイプ *seq init-declarator-list* ;
- *seq typedef seq init* リスト ; // C ++ 11
- *identifier -specifier-seq opt = type-id* をします。 // C ++ 11

Examples

な typedef の

typedef は、またはとじをちますが、 typedef というがまれてい typedef 。 typedef のにより、はまたはのわりにをします。

```
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"
```

タイプエイリアスがされると、タイプののと同じでできます。

```
typedef int A[100];
// S is a struct containing an array of 100 ints
struct S {
    A data;
};
```

typedef はなをしてしません。それは、のタイプをするのをするだけです。

```
struct S {
    int f(int);
};
typedef int I;
// ok: defines int S::f(int)
I S::f(I x) { return x; }
```

よりな typedef の

typedef は、のと同じをち、よりなをみきすることができるというがあり typedef 。

```
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"
```

これは、ではないメンバーへのポイントなど、がしているににです。

[illegible]

なプログラムであっても、ののをえておくことはです。

```
void (Foo::*Foo::f(const char*)) (int);
int (&q()) [100];
```

typedef をすると、みきがになります。

[illegible]

typedefでのをする

`typedef` キーワードはなので、ににされます。したがって、されたは、 `typedef` がしないにそのがつをし `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(*)()
```

"using"によるエイリアス

C ++ 11

using にですするがにき、がにきます。がどこにあるかをるためにスキャンするはありません。

```
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
```

usingでエイリアスをするのは、typedefをusingでエイリアスをするのとまったく同じです。これは、同じことをするためのものです。

typedefとはなり、usingをテンプレートにusingことができます。usingでされた "template typedef"はエイリアステンプレートとされます。

オンラインでtypedefとのエイリアスをむ <https://riptutorial.com/ja/cplusplus/topic/9328/typedefとのエイリアス>

46: アラインメント

き

C++のすべてののはしています。これは、そのタイプのオブジェクトをできるメモリアドレスのです。そのアドレスをオブジェクトのであるとである、メモリアドレスはオブジェクトのにです。

のはに2のです1をむ。

では、のことがされています。

- のは、そのサイズのです。たとえば、サイズが16バイトのクラスは、1,2,4,8,16のいずれかのアライメントをつことができますが、32ではできませんクラスのメンバーのサイズは14バイトですが、クラスはアラインメントをつがありますコンパイラは、2のパディングバイトをして、クラスのサイズを16にします。
- のきバージョンとなしバージョンのアラインメントはじです。
- `void`へのポインターは、`char`へのポインターとじわせをちます。
- のcv-qualifiedバージョンとcv-unqualifiedバージョンは、じわせをちます。

アライメントはC++ 03にするが、C++ 11まではアラインメント `alignof` とアライメントの `alignas` をがになるまでではなかったことにしてください。

Examples

のの

C++ 11

のは、としての `alignof` キーワードをしてできます。は、`std::size_t` です。つまり、コンパイルにできます。

```
#include <iostream>
int main() {
    std::cout << "The alignment requirement of int is: " << alignof(int) << '\n';
}
```

な

`int`のはのとおりです。

にすると、タイプのがされます。にすると、のがされます。オブジェクトではないため、にアラインメントがありません。

アラインメントの

C++ 11

`alignas` キーワードをすると、クラスデータメンバー、クラスのまたは、またはのまたはを、のサポートされているにすることができます。それは2つので

- `alignas(x)`、`x`では、エンティティにえる`x`サポートしている、。
- `alignas(T)` `T`では、エンティティにのアライメントにしいえる`T`すなわち、ある、`alignof(T)`サポートされている、。

じエンティティにの`alignas`がされている、もしいものがされます。

このでは、バッファ`buf`は、そのが`unsigned char`であっても、`int`オブジェクトをするためににされていることがされています。

```
alignas(int) unsigned char buf[sizeof(int)];  
new (buf) int(42);
```

`alignas`をして、このなしでがつよりさいをにえることはできません

```
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`は、がえられたとき、なアライメントをえなければならぬ。なはに2のべきであり、0よりきくなければなりません。コンパイラは、`std::max_align_t`のまですべてのなをサポートするがあります。これらは、これよりきなをサポートすることがありますが、そのようなオブジェクトのメモリのりてのサポートはられています。アライメントのはにします。

C++ 17では、オーバーライドされたのメモリをりてるために`operator new`サポートしています。

オンラインでアラインメントをむ <https://riptutorial.com/ja/cplusplus/topic/9249/アラインメント>

47: イテレータ

Examples

Cイテレータポインタ

```
// 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
```

このコードは、のようになに5のをします。

```
1
2
3
4
5
```

それをちく

```
const int array[] = { 1, 2, 3, 4, 5 };
```

このは、5つのをつしいをします。Cは、がしたブロックになにされるメモリへのポインタにぎません。

```
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);
```

これらのは2つのポインタをします。のポインタには、のののアドレスであるポインタのがえられ

ます。C++のsizeofは、のサイズをバイトでします。のサイズでって、これはののをえま
す。これをして、ののブロックのアドレスをつけることができます。

```
for (const int* i = first; i < afterLast; ++i) {
```

ここでは、イテレータとしてするポインタをします。それは、たちがするののアドレスでされ、
そして、たちはりしけるだろう i よりない $afterLast$ りをし、 i のアドレスをしているarray。

```
std::cout << *i << std::endl;
```

に、ループでたちはたちのアクセスすることができ i それをによってをしているの。 $*$ は、 i のア
ドレスのをします。

イテレータはポジションです

イテレータは、のをナビゲートしてするであり、ポインタのされたです。には、イテレータはで
あり、ではないことをえておくことがです。たとえば、のシーケンスをします。

```
A B C
```

シーケンスには3つのと4つのがまれます

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
```

はシーケンスのものです。は、シーケンスにのがあるがこるです。えは、つのAのまたはに、ない
に、のにします。の`erase(A)`は、にそのをつけてすることによってわれます。

イテレータからへ

からにするには、イテレータをします。

```
auto my_iterator = my_vector.begin(); // position
auto my_value = *my_iterator; // value
```

イテレータは、シーケンスでされるをするとえることができます。これは、シーケンスの`end()`イ
テレータをしてしないをするでににちます

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
↑           ↑
|           +-- An iterator here has no value. Do not dereference it!
+----- An iterator here dereferences to the value A.
```

C++ライブラリにあるすべてのシーケンスとコンテナで、`begin()` はイテレータをのにし、`end()` はイテレータをののではない までします。として、アルゴリズムにおけるこれらのイテレータのは、しばしば `first` と `last` ラベルけされる。

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
  ↑           ↑
  |           |
+- first      +- last
```

のシーケンスにもなくとも1つのがまれているので、 のシーケンスにするイテレータをすることもできます。

```
+---+
|   |
+---+
```

のシーケンスでは、`begin()` と `end()` はじになり、どちらもできません。

```
+---+
|   |
+---+
  ↑
  |
+- empty_sequence.begin()
  |
+- empty_sequence.end()
```

イテレータのは、のをマークすることです。

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
  ↑   ^   ^   ↑
  |           |
+- first      +- last
```

イテレータのはイテレータのにへのをします。このビューがになはのとおります。

- `insert` は、イテレータがすにをし、
- `erase` は、されたものとじにするイテレータをし、
- イテレータとそれにする **イテレータ** は、のじにされます

なイテレータ

は、えばのでそのがもはやシーケンスのではない、 になります。されたイテレータは、なにりてされるまでできません。えば

```
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
```

C++ライブラリの多くのアルゴリズムとシーケンスメンバーには、イテレータがなくなったときにされるルールがあります。アルゴリズムはイテレータをうおよびにするがなります。

イテレータをったナビゲーション

わかっているように、イテレータはシーケンスをナビゲートするためのものです。これをうためには、イテレータはシーケンスのをさせるがあります。はにむことができ、またはすることができます。

```
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.
```

`std::distance`の2のは、のからでなければならぬこととしてくださいます、 `first second`のにするがあります。

イテレータでをすることはできますが、すべてのタイプのイテレータにしてすべてのがされているわけではありません。 `a = b + 3`; ランダムアクセスイテレータではしますが、フォワードまたはイテレータではしません。これは、まだ `b = a; ++b; ++b; ++b`; ような3つのでめることができます `b = a; ++b; ++b; ++b`; 。したがって、イテレータのがわからないたとえば、イテレータをけるテンプレートなどになをすることをおめします。

イテレータの

C++には、いくつかのなるイテレータのがされています。これらは、するでどのようにするかによってグループされます。イテレータがモデルするはがかっていれば、それがするシーケンスになく、そのイテレータのをすることができます。それらは、くの、のイテレーターのコンセプトがのコンセプトよりもれているため、ものないにされます。

- ごとに1のみできます。めることができるだけで、に1つのポジションしかありません。
- Forward Iteratorsのできるイテレータ。
-

イテレータまた、にのを1つめることができ、イテレータ。

- ランダムアクセスイテレータにのをまたはにめることができるイテレータ。
- イテレータC++ 17のデータがメモリでしていることをするランダムアクセスイテレータ。

アルゴリズムは、えられたイテレータによってモデルされたにじてわることがあります。例えば、イテレータにしては`random_shuffle`をすることができるが、ランダムアクセスイテレータをとするよりなをすることができる。

イテレータの

イテレータのは、イテレータのプロパティにするされたインタフェースをします。、、ポインタ、、イテレータのカテゴリをすることができます

```
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;
}
```

イテレータのカテゴリをしてアルゴリズムをすることができます。

```
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());
}
```

イテレータのカテゴリはにイテレータのですが、イテレータはのタグをっていませんが、コードをすことがしています。

イテレータ

リストやベクトルをにしたいは`reverse_iterator`をうことができます。は、`base()`をじてアクセスできるメンバーとしてするまたはランダムアクセスからされます。

`rbegin()`するには、コレクションのわりのイテレーターとして`rbegin()`と`rend()`をそれぞれし、そ

それぞれコレクションのとしてします。

えは、にするには

```
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
```

イテレータは、`base()` メンバをしてイテレータにできます。そのは、イテレータが`base()` イテレータよりも1つのをすることです。

```
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()
      |               |
      |               |
begin()               rbegin().base()
rend().base()
```

イテレータがのをマークするビジュアライゼーションでは、はよりです。

```
+---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+
      ↑               ↑
      |               |
      |               |
      |               |
begin()               end()
rend()               rbegin()
rend().base()         rbegin().base()
```

ベクトルイテレータ

`begin`は、シーケンスコンテナののに`iterator`をします。

`end`は`iterator`をのの`end`します。

ベクトルオブジェクトが`const`、`begin`と`endbegin const_iterator`ます。あなたのベクトルが`const`でなくても`const_iterator`がされるようにしたいなら、`cbegin`と`cend`うことができます。

```
#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;
}
```

1 2 3 4 5

マップイテレータ

コンテナののにするイテレータ。

マップオブジェクトがconstされている、はconst_iteratorします。それのは、iteratorし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';
```

a => 200
b => 100
c => 300

ストリームイテレータ

ストリームイテレータは、シーケンスをみんだり、コンテナからされたデータをするがあるにです。

```
// 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, " -- "));
```


サンプルプログラムは、 1 -- 2 -- 3 -- 4 --をにします。

のジェネレータバックアップイテレータをする

のこのパターンは、オブジェクトの「ストリーム」をし、それをループするためにループコードをできるをつことです。

これをC++でモデルできます

```
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;
};
```

ライブラリ。

されたをにして、すでに行っているかどうかをにできます。

エンドジェネレーターイテレーターのはしてされないなので、 `std::function` 1コピーするだけのジェネレーターイテレーターをすることができます。デフォルトでされたジェネレーターイテレーターは、それとしく、のすべてのエンドジェネレーターイテレーターとされます。

オンラインでイテレータをむ <https://riptutorial.com/ja/cplusplus/topic/473/イテレータ>

48: インライン

き

インラインは、[1つのルール](#)にすることなく、のことができます。されている、リンクはすべてのをプログラムののオブジェクトにマージします。

Examples

クラスにデータメンバーをする

`inline`でされている、そのクラスのデータメンバーはクラスでにできます。例えば、のクラスをヘッダにすることができます。C++ 17よりには、`.cpp`ファイルをして`Foo::num_instances`をするがありました。これは、だけされるようになっていましたが、C++では、`inline Foo::num_instances`すべてに`int`オブジェクトをします。

```
// warning: not thread-safe...
class Foo {
public:
    Foo() { ++num_instances; }
    ~Foo() { --num_instances; }
    inline static int num_instances = 0;
};
```

なケースとして、`constexpr`データメンバーはにインラインです。

```
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;
```

オンラインでインラインをむ <https://riptutorial.com/ja/cplusplus/topic/9265/インライン>

49: インライン

き

`inline`でされたは、インラインです。インラインは、[1つのルール](#)にすることなくすることができます。したがって、リンケージをしてヘッダーですることができます。のインラインヒントをコンパイラにすると、コードにをインラインするがありますが、はされません。

- `inline function_declaration`
- `inline function_definition`
- クラス{`function_definition`};

、にされたコードがにさい、それはインラインするのにしています。なぜそうなのかがきく、ループでインラインされた、すべてのびしにして、きなコードがされ、されたバイナリサイズがらんでしまう。しかし、どのくらいのものでしょうか

インラインは、オーバーヘッドびしをするためのらしいであるとわれるが、それはマークされているすべてのではないことにすべきだ`inline`インラインされています。いえれば、`inline`でうと、それはではなく、コンパイラのヒントにすぎません。コンパイラはをインラインするはなく、することはです。のコンパイラはこのようなをうがれています。このキーワードは、プログラマによるインラインのがコンパイラによってにけめられたのです。マークされていないとしても`inline`、それはそうすることでをたとき、コンパイラによってインラインされています。

リンケージとしてのインライン

のC++の`inline`のよりなは、それをリンケージとしてすることからています。ではなく、のソースにインクルードされるヘッダーのをすると、にはこのののコピーがあり、[ODR One Definition Rule](#)となります。このは、などのが1つしかないことをまかにしています。このをするために、を`inline`でマークすると、にのリンケージがになります。

よくある

C++で/メソッドのキーワード「インライン」をくべきはいつですか

ヘッダーにをしたいときのみ。よりには、のがのコンパイルでされるのみです。コンパイラにコードをしなからするためのをえるので、ヘッダファイルにさなをすることをおめします。コンパイルもくなります。

C++で/メソッドのキーワード '`inline`'をくべきではありませんか

コンパイラがインラインすると、コードがよりくされるときは、`inline`しないでください。

メソッドをいつインラインするかはコンパイラがいつしますか

に、コンパイラはこれをあなたよりもれたものにすることができます。ただし、をたないコンパイラには、コードをインラインするオプションはありません。にされたコードでは、はすべてのプライベートメソッドがインラインされます。

- [メソッドのキーワード「インライン」はいつくべきですか](#)
- [まだインラインでしていますか](#)

Examples

メンバーインライン

```
inline int add(int x, int y);
```

メンバーのインライン

```
inline int add(int x, int y)
{
    return x + y;
}
```

メンバーインライン

```
// header (.hpp)
struct A
{
    void i_am_inlined()
    {
    }
};

struct B
{
    void i_am_NOT_inlined();
};

// source (.cpp)
void B::i_am_NOT_inlined()
{
}
```

インラインとはですか

```
inline int add(int x, int y)
{
    return x + y;
}
```

```
}

int main()
{
    int a = 1, b = 2;
    int c = add(a, b);
}
```

のコードでは、`add`がインラインされると、のコードはのようになります

```
int main()
{
    int a = 1, b = 2;
    int c = a + b;
}
```

インラインはどこにもえません。そのはびしのにインラインされます。た`add`インラインされていない、がびされます。しい[スタックフレームの](#)、のコピー、ローカルの、ジャンプキャッシュミスによるのローカリティのなど、をびすオーバーヘッドがするがあります。

オンラインでインラインをむ <https://riptutorial.com/ja/cplusplus/topic/7150/インライン>

50: オート

キーワード `auto` は、にされたをすです。

これはCからされたC ++ 98のみキーワードです。いバージョンのC ++では、にがあることをにすためにできます。

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

そのいがされました。

Examples

なサンプル

キーワード `auto` は、ののをします。

ロングタイプのをうとときににです

```
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" );
```

ベースのforループで

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

ラムダと

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

タイプのりしをけるために

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

でなコピーをけるために

```
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!
```

コピーのは、されるがには `std::pair<const int, float>` です。

およびテンプレート

`auto` は、テンプレートのにがじることもあります。

```
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.
```

そのは、ある `operator*valarray` あなたをするプロキシオブジェクト `valarray` のとしては。 `auto` をうと、ぶらがっているがされます。 `mult` わりに `std::valarray<int>` がされた、コードはに3をします。

、および

`auto` キーワードは、 `int` や `char` たをします。これは `const` キーワードと `&` をしてそれぞれ `const` またはをすようにできます。これらのはみわせることができます。

このでは、 `s` はそのは `std::string` とされるなので、 `for` ループ `for` りすたびにベクトルから `s` がコピーされます。

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

ループのが `s.append(" and stuff")` びすなどして `s` する `s.append(" and stuff")` 、の `strings` メンバーではなく、このコピーのみがされ `strings` 。

、 `s` が `auto&` でされている、それは `std::string&` とされるなので、ループのでベクトルのへのがりてられます。

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

このループのでは、 `s` をすると、する `strings` のにします。

に、 `s` が `const auto&` とされているは、 `const` になります。つまり、ループのりしでベクトルのへの `const` がりてられます。


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

このループのでは、`s`はできませんつまり、`const`メソッドはびすことができません。

する`auto`レンジベースとの`for`ループを、することがにである`const auto&`これは、なコピーをするので、ループは、にループされるをしない。

のりの

のりののでは`auto`がされます。

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

これは

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

に`std::declval<T>`わりにパラメータをする`decltype`とみわせるとです

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

ラムダ C++ 14

C++ 14

C++ 14では、`lambda`に`auto`をできます。

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

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

そのラムダは、

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

その

```
lambda print;

print(42);
```

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

およびプロキシオブジェクト

によっては、`auto`はプログラマーがしたとおりにしないことがあります。タイプのがでないでも、タイプはのをします。

たとえば、プロキシオブジェクトがコードでされる、のようになります。

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

ここで`flag`は`bool`ではなく、`std::vector<bool>::reference`です。テンプレート`vector bool`の、`operator []`は`operator bool`されたプロキシオブジェクトをします。

`flags.push_back(true)`がコンテナをすると、このはもはやしなくなったをしてダングリングするがあります。

また、のをにします。

```
void foo(bool b);

std::vector<bool> getFlags();

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

`vector`はちにされるため、`flag`はされたへのとなります。 `foo`びすと、のビヘイビアがびされます。

このような、`auto`でをし、したいにキャストしてすることができます

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

それで、に`auto`を`bool`きえるのがにかなっています。

プロキシオブジェクトがをきこすのケースは、[テンプレート](#)です。その、テンプレートはのためにのをえてにはされていないがあり、のプロキシオブジェクトをするとのがします。

オンラインでオートをむ <https://riptutorial.com/ja/cplusplus/topic/2421/オート>

51: オブジェクトタイプのレイアウト

のサイズもしてください。

Examples

クラスの

"class"とは、`class`または`struct`キーワードをしてされたをします `enum struct`または`enum class`はありません。

- のクラスでもなくとも1バイトのをします。したがって、にパディングからされます。これにより、`p`のクラスのオブジェクトをす、`p + 1`はのアドレスであり、のオブジェクトをしすことになります。ただし、のクラスは、クラスとしてすると、サイズが0になるがあります。のをしてください。

```
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
```

- クラスのオブジェクトには、クラスとメンバーのオブジェクトがまれます。したがって、たとえば、のクラス

```
struct S {
    int x;
    char* y;
};
```

サブオブジェクトとばれ、`x`のをむ`S`オブジェクトに`sizeof(int)`バイトのしたシーケンスがあり、`y`のをむ`sizeof(char*)`バイトをつのサブオブジェクトが`sizeof(char*)`ます。2つはインターリーブできません。

- クラスに`t1`, `t2`, ..., `tN`および/またはクラスがある、そのきさはのをして`sizeof(t1) + sizeof(t2) + ... + sizeof(tN)`でなければなりません。ただし、メンバーおよびクラスのにじて、コンパイラは、サブオブジェクトまたはオブジェクトのまたはにパディングをすることがあります。

```
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).

```

- アラインメントのためにパディングがオブジェクトにされた、サイズはメンバーとベースクラスのサイズのよりも大きくなります。 n バイトアラインメントの、サイズは、 n のであり、すべてのメンバーおよびベースクラスのサイズよりも大きくなります。 memN 、には、のアドレスにされる $\text{alignof}(\text{memN})$ そして n にはとなり alignof すべてのメンバーのうち $\text{alignof } S$ 。このため、 alignof のさいののが、よりきな alignof がすると、にすればのがしくわせされないがある。この、2つののにパディングアラインメントとしてもらえるがされ、のがのわせをすることができ。に、よりきな alignof のメンバーのろにさい alignof メンバーがく、はパディングはありません。このプロセスは「パッキング」ともばれます。
クラスのメンバーは、メンバーの alignof をの alignof であるため、クラスは、またはにまれるのみの alignof にされます。

```

// 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

- `alignas`してなをすると、パディングがされ、されたがのよりもさくなくてもにそのタイプがたされます。えは、のでは、`Chars<5>`には3あるいはそのパディングバイトをし、サイズが8になるようにします。アライメントが4のクラスは、そのクラスのをすることはであるため、5をします。したがって、パディングバイトをすることによって、サイズを4のに"りげ"なければなりません。

```

// 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");

```

- あるクラスの2つのメンバーがじアクセスをつ、でになるものは、オブジェクトのにることがされます。しかし、2つのメンバーがなるアクセスをつ、オブジェクトのなはされません。
- オブジェクトでクラスのサブオブジェクトがどのようなでされるか、それらがしてするかどうか、およびそれらがメンバーのサブオブジェクトのにされるかどうかはです。

いタイプ

`unsigned char`は、すべてのビットをして2をします。したがって、たとえば、`unsigned char`が8ビットの、`char`オブジェクトのな256のビットパターンは、256のなる{0,1、...、255}をします。42

は、ビットパターン00101010によってされることがされている。

`signed char`にはパディングビットはありません。つまり、`signed char`が8ビットのは、8ビットのをちます。

これらののは、いタイプのタイプにはされないことにしてください。

なしは、なバイナリシステムをしますが、パディングビットをむことがあります。えは、`unsigned int`が64ビットであるが、 $2^{32} - 1$ ののしかすることはできないしかし、そうではないかもしれないがある。の32ビットはパディングビットであり、きむべきではない。

きは、ビットとによってパディングビットをつバイナリシステムをします。きとするなしのにするは、じをちます。たとえば、`unsigned short`オブジェクトのビット・パターン0001010010101011が₅₂₉₁す、`short`オブジェクトとしてされたときの₅₂₉₁もします。

3つのシステムがすべてののをたしているので、2の、1の、またはのきさのがされるかどうかは、によってされます。

のは、です。もには、`float`と`double`はIEEE 754にし、32ビットと64ビットのさです `float`は23ビットのをち、8のビットと1ビットにいます。ただし、このではもされていません。は、しばしば「トラップ」をち、にするとエラーをきこします。

にはにパディングがありません。したがって、`T`のは、メモリにレイアウトされた`T`オブジェクトのシーケンスにすぎません。

はのであり、はにされます。たとえば、がある

```
int a[5][3];
```

`a`は3 `int`の5つののです。したがって、`a[0]`、3つのからされている`a[0][0]` `a[0][1]` `a[0][2]`に、メモリにレイアウトされている`a[1]`からされています `a[1][0]`、`a[1][1]`、`a[1][2]`これは、のなとばれます。

オンラインでオブジェクトタイプのレイアウトをむ <https://riptutorial.com/ja/cplusplus/topic/9329/>
オブジェクトタイプのレイアウト

52: キーワード

き

キーワードはC++でされたのをち、としてすることはできません。ライブラリヘッダーをむすべてのでプリプロセッサをしてキーワードをすることはです。ただし、キーワードはでなをいます。

- `asm` リテラル ;
- `noexcept` //1
- `noexcept` //2
- `noexcept` //2
- のサイズ
- `sizeof type-id`
- `sizeof ...` //C++ 11
- `typename` ネストされたの //1
- `typename` ネストされたテンプレート *opt simple-template-id* //1
- *opt* //2
- `typename ...` *opt* //2; C++ 11
- タイプ *opt* = タイプID //2
- テンプレート< *template-parameter-list* >... *opt* *opt* //3
- テンプレート< *template-parameter-list* > *opt* = *id-expression* //3

キーワードのなリストはのとおりです

- `alignas` C++ 11
- `alignof` C++ 11
- `asm`
- `auto` C++ 11より、C++ 11より
- `bool`
- `break`
- `case`
- `catch`
- `char`
- `char16_t` C++ 11
- `char32_t` C++ 11
- `class`
- `const`
- `constexpr` C++ 11
- `const_cast`
- `continue`
- `decltype` C++ 11
- `default`
- のメモリのための `delete` C++ 11
- `do`
- `double`

- `dynamic_cast`
- `else`
- `enum`
- `explicit`
- `export`
- テンプレートのとしての `extern`、リンケージの `extern`
- `false`
- `float`
- `for`
- `friend`
- `goto`
- `if`
- の `inline`、C++ 11、C++ 17
- `int`
- `long`
- `mutable`
- `namespace`
- `new`
- `noexcept` C++ 11
- `nullptr` C++ 11
- `operator`
- `private`
- `protected`
- `public`
- `register`
- `reinterpret_cast`
- `return`
- `short`
- `signed`
- `sizeof`
- `static`
- `static_assert` C++ 11
- `static_cast`
- `struct`
- `switch`
- `template`
- `this`
- `thread_local` C++ 11
- `throw`
- `true`
- `try`
- `typedef`
- `typeid`
- `typename`
- `union`
- `unsigned`
- `using` をするために、をエイリアスし、をします
- のための `virtual`、クラス
- `void`
- `volatile`
- `wchar_t`
- `while`

トークンの`final`と`override`はキーワードではありません。それらはとしてすることができ、のコンテキストでのみなをちます。

トークン`and`、`and_eq`、`bitand`、`bitor`、`compl`、`not`、`not_eq`、`or`、`or_eq`、`xor`、および`xor_eq`のスペルです `&&`、`&=`、`&`、`|`、`~!`、`!=`、`||`、`|=`、`^`、および`^=`とする。はそれらをキーワードとしていませんが、それらをする事やそれらがするのはであるため、すべてののキーワードとなります。

のトピックでは、のやフローのなどのなにつ、C++のキーワードのながまれています。

- [タイプのキーワード](#)
- [フロー](#)
- [リテラルキーワード](#)
- [タイプキーワード](#)
- [キーワード](#)
- [クラス/](#)
- [クラス](#)

Examples

asm

`asm` キーワードは、のオペランドをとります。これはリテラルでなければなりません。これはのをちますが、はのアセンブラにされ、アセンブラのはユニットにみまれます。

`asm` はではなくであるため、ブロックスコープまたはネームスペーススコープグローバルスコープをむのいずれかにれることがあります。しかし、インラインアセンブリはC++のによってをけることができないので、`asm` は`constexpr`のにれないことがあります。

```
[[noreturn]] void halt_system() {  
    asm("hlt");  
}
```

な

1. のコンストラクタにすると、そのコンストラクタがなをするのをぐことができます。

```
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
```

C++ 11ではイニシャライザリストがされているので、C++ 11では、のと同じで、ののをつコンストラクタに`explicit`にできます。

```
struct S {
    explicit S(int x, int y);
};
S f() {
    return {12, 34}; // ill-formed
    return S{12, 34}; // ok
}
```

C++ 11

2. にすると、そのをしてのをできなくなります。

```
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. そのオペランドのがをできるかどうかをする。びされるのはされないで、`noexcept`はのネガティブを`noexcept`があることにしてください。オペランドはされません。

```
#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
}
```

このでは、`bar()` がをスローすることはできませんが、`bar()` がをできないというはにされていないため、`noexcept(bar())` も`false`です。

2. をするとき、がをできるかどうかをします。では、がをできないことをします。カッコで
まれたをすると、のにじてがをできるかどうかをします。

```
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&) {}
}
```

このでは、`f4`、`f5`、および`f6`はをできないとしています。`f6`にがスローされることはありますが、され、からすることはできません。`f2`がをすることがあるとしました。ときに
`noexcept`がされ、それはとである`noexcept(false)`ので、たちは、にすることをした`f1`と`f3`、
がのにスローされることができないにもかかわらず、をすることができる`f3`。

C++ 17

ファンクションが`noexcept`あるかどうかは、ファンクションのタイプのです。つまり、のでは、
`f1`、`f2`、および`f3`タイプは`f4`、`f5`、および`f6`となります。したがって、ポインタ、テンプレート
などでは、`noexcept`もです。

```
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
```

タイプ

1. のにがある、`typename`はそのがのであることをします。これは、ネストされたがのインスタ
ンスのである、にテンプレートでとなることがよくあります。このでは、`std::decay<T>`テ
ンプレートパラメータに`T`ネストされたのにするためにので、`type`、々がののにけるがあり
`typename`。よりくのdetailsについては、「テンプレート」と「`typename`」のキーワードを
どこに、なぜれなければならないのかをしてください。

```
template <class T>
auto decay_copy(T&& r) -> typename std::decay<T>::type;
```

2. テンプレートのにパラメータをします。このでは、`class`とがあります。

```
template <typename T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

C++ 17

3. `typename`は、`class`のように、パラメータののにテンプレートテンプレートパラメータをするときにもできます。

```
template <template <class T> typename U>
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

のサイズ

そのオペランドのバイトを。またはのいずれかです。オペランドがの、されません。サイズは、`std::size_t`のです。

オペランドがのは、でむがあります。

- に`sizeof`をすることはです。
- `void`をむなに`sizeof`をすることはです。
- `sizeof`が`T&` or `T&&`にされる、`sizeof(T)`とです。
- `sizeof`クラスにすると、またはのパディングバイトをむ、そののなオブジェクトのバイトがられます。したがって、`sizeof`のはして0になりません。については、[オブジェクトのレイアウト](#)をしてください。
- `char`、`signed char`、および`unsigned char`のサイズは1です。に、1バイトは`char`オブジェクトをするためになメモリとしてされます。のシステムでは8ビットよりもい`char`オブジェクトがあるため、ずしも8ビットをするわけではありません。

`expr`がの、`sizeof( expr )`は`sizeof(T)`とです。ここで、`T`は`expr`のです。

```
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

`sizeof...`は、パラメータパックののを`sizeof...`。

```
template <class... T>
void f(T&&...) {
    std::cout << "f was called with " << sizeof...(T) << " arguments\n";
}
```

なるキーワード

void C++

1. のりのとしてされる、`void`キーワードは、がをさないことをします。のパラメータリストにすると、`void`はそのがパラメータをらないことをします。ポインターのでされる、`void`はポ

インターが「ユニバーサル」であることをします。

2. ポインタのがvoid *の、ポインタはconstまたはvolatileキーワードでされていないをすることができます。ポイドポインタは、のにキャストされていないりできません。 voidポインターは、のタイプのデータポインターにすることができます。
3. voidポインターはをすることができますが、C ++のクラスメンバーはすることができません。

```
void vobject;    // C2182
void *pv;       // okay
int *pint; int i;
int main() {
    pv = &i;
    // Cast optional in C required in C++
    pint = (int *)pv;
```

C ++

1. ハードウェアによってプログラムでオブジェクトをできることをするためにできる。

```
volatile declarator ;
```

C ++

1. virtualキーワードは、またはクラスをします。

```
virtual [type-specifiers] member-function-declarator
virtual [access-specifier] base-class-name
```

パラメーター

1. **type-specifiers**メンバーのりのをします。
2. **member-function-declarator**メンバをします。
3. **access-specifier**クラスpublic、protectedまたはprivateへのアクセスレベルをします。キーワードのにできます。
4. **base-class-name**にされたクラスのをします。

このポインタ

1. このポインタは、クラス、、またはのメンバーでのみアクセスなポインタです。これは、メンバーがびされるオブジェクトをします。メンバーにはこのポインタはありません。

```
this->member-identifier
```

オブジェクトのこのポインタはオブジェクトのではありません。オブジェクトのsizeofのにはされません。わりに、メンバがオブジェクトにしてびされると、オブジェクトのアドレスはのしと

してコンパイラによってされます。たとえば、のびし

```
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、catch ステートメント C++

1. C++でをするには、try、throw、catchをします。
2. まず、tryブロックをして、をスローするのある1つのステートメントをみます。
3. スローは、tryブロックでしばしばエラーがしたことをします。のタイプのオブジェクトをスローのオペランドとしてできます。、このオブジェクトは、エラーにするのにされます。ほとんどの、`std::exception`クラス、またはライブラリでされているクラスのいずれかをすることをめします。そのうちの1つがでないは、`std::exception`からのクラスをさせることをめします。
4. スローされるのあるをするには、tryブロックのに1つのcatchブロックをします。それぞれのcatchブロックは、できるのタイプをします。

```
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");
    // ...
}

```

tryのコードは、コードのされたセクションです。throwのスロー - つまりraise - です。catchのコードブロックはハンドラです。これは、throwとcatchのがあるにスローされるをキャッチするハンドラです。

```

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;
}

```

C++

1. によっては、クラスのメンバーでないまたはのクラスのすべてのメンバーにメンバーレベルのアクセスをえるがです。クラスのだけが、そのがであるかをすることができます。またはクラスは、をクラスのフレンドとしてすることはできません。クラスでは、friendキーワードとメンバーまたはのクラスのをして、クラスのプライベートメンバーおよびされたメンバーへのアクセスをします。テンプレートでは、タイプパラメータをフレンドとしてできます。
2. にされていないフレンドをすると、そのはむクラススコープにエクスポートされます。

```

class friend F
friend F;
class ForwardDeclared;// Class name is known.
class HasFriends
{
    friend int ForwardDeclared::IsAFriend();// C2039 error expected
};

```

1. フレンドは、クラスのメンバーではなく、クラスのプライベートメンバーおよびされたメン

バーにアクセスできます。フレンドはクラスメンバーとはみなされません。なアクセスがえられているのです。

2. フレンドはクラスのスコープにはなく、のクラスのメンバーでないりメンバー。と ->をしてびされません。
3. friendは、アクセスをしているクラスによってされています。フレンドは、クラスのどこにでもできます。アクセスキーワードのをけません。

```
#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
}
```

としてのクラスメンバー

```
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
```

オンラインでキーワードをむ <https://riptutorial.com/ja/cplusplus/topic/4891/キーワード>

53: クライアントサーバーの

Examples

ハローTCPサーバー

に[Beej's Guide to Network Programming](#)をんで、それをすばやくんで、こののほとんどをもうししくするがあるといいます。ここでは、すべてのにして「Hello World」とうなTCPサーバーをします。すべきもうつのことは、サーバーがクライアントとりしすることです。これはに1つのクライアントをします。それぞれのびしとソケットにするながまれているがあるので、するマニュアルページをずチェックしてください。

ポートでサーバーをするので、ポートのもります。コードをいめる -

```
#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, socketFD 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;
}

```

のプログラムは -

```

Detecting addresses
(1) IPv4 : 0.0.0.0
(2) IPv6 : ::
Enter the number of host address to bind with: 1

```

ハローTCPクライアント

このプログラムは、Hello TCP Serverプログラムとです。おいのをチェックするためにどちらかをすることができます。プログラムのれは、Hello TCPサーバではよくあることですので、あまりにもそれをてください。

ここにコードがあります -

```

#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;
}

```

オンラインでクライアントサーバーのをむ <https://riptutorial.com/ja/cplusplus/topic/7177/クライアントサーバーの>

54: クラス/

- `variable.member_var =`;
- `variable.member_function`;
- `variable_pointer-> member_var =`;
- `variable_pointer-> member_function`;

のいがあります `struct` と `class` のキーワードがデフォルトで、のメンバ、メンバ、およびクラスと
いうことである `struct` あり `public` にしながら、 `class` らは `private` 。 `C++` のプログラマは、コンス
トラクタとデストラクタをっていればそれをクラスとぶがあり、そのインバリエントをすること
ができます。なのまりであればですが、`C++` はしません。

Examples

クラスの

クラスはユーザーです。クラスは、 `class` 、 `struct` または `union` キーワードでされています。では
、「クラス」というは、ユニオンクラスのみをします。

クラスは、クラスメンバーのコレクションであり、のようになります。

- メンバー「フィールド」ともばれる
- メンバー「メソッド」ともばれる
- メンバまたはえは、"れクラス"、
- メンバテンプレート、、クラス、エイリアステンプレート

`class` や `struct` メンバーおよびのためのデフォルトのアクセスはとされたクラスのための「プライ
ベート」であることをいて、キーワード、クラスキーとばれ、にがあり、 `class` でされたクラス
のためのキーと「」 `struct` または `union` キー [アクセス](#) 。

たとえば、のコードスニペットはじです。

```
struct Vector
{
    int x;
    int y;
    int z;
};
// are equivalent to
class Vector
{
public:
    int x;
    int y;
    int z;
};
```

クラス`をすることで、しいがプログラムにされ、そのクラスのオブジェクトをのようにインスタンスすることができます。

```
Vector my_vector;
```

クラスのメンバーは、ドットをしてアクセスされます。

```
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;
```

アクセス

アクセスとしてする3つの**キーワード**があります。これらは、にくクラスメンバへのアクセスを、のがアクセスレベルをびするまでします。

キーワード	
public	もがアクセスできます
protected	クラス、したクラスとフレンドだけがアクセスできます
private	クラスとのみがアクセスできます

がclassキーワードをしてされている、デフォルトのアクセスはprivateですが、がstructキーワードをしてされている、デフォルトのアクセスはpublicです。

```
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
```

アクセスはにフィールドとメソッドへのアクセスをするためにされます。たとえば、をするのではなく、ゲッターとセッターをにするなど、

```
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;
```



```
};
```

`protected` をすると、ののにクラスのみがアクセスできるようにするのにです。たとえば、のコードでは、メソッド `calculateValue()` は、 `FortyTwo` などのクラス `Plus2Base` からしたクラスにのみアクセスできます。

```
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; }
};
```

`friend` キーワードをすると、されたメンバーとのメンバーにアクセスするためのやにアクセスをできます。

`public`、`protected`、および `private` キーワードは、クラスのサブオブジェクトへのアクセスをまたはするためにもできます。 のをしてください。

クラス/はをつことができます。

クラス/_Bがクラス/_Aからする、これは_Bが_Aつことをします。 _Bは_Aクラス/、 _Aはクラス/です。

```
struct A
{
public:
    int p1;
protected:
    int p2;
private:
    int p3;
};

//Make B inherit publicly (default) from A
struct B : A
{
};
```

クラス/には3つのがあります。

- `public`
- `private`
- `protected`

デフォルトのはメンバーのデフォルトのと同じであることにしてください `struct` キーワードをする は `public` を、 `class` キーワードを `private` にします。

`struct` から `class` させることもですまたはその。この、デフォルトのはによってされているので、`struct` からする `class`、パブリックにデフォルトされます、そして `class` からした `struct`、デフォル

トでは、のをつことになります。

public

```
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

```
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

```
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
```

protectedはprotectedしていますが、のはまれです。アプリケーションでprotectedがどのようにされるかの1つののは、なクラスの、「」とされます。

OOPがしい、はしばしば「IS-A」をモデルするとわれました。つまり、パブリックは、クラスの

インスタンスがクラスのインスタンスであるにのみしいものです。

これはで [Liskov Substitution Principle](#) にされました。パブリックは、クラスのインスタンスがなでクラスのインスタンスにきえられるそしてそれでもがあるにのみしてください。

は、にはになる「HAS-A」ともばれるのからされているとわれています。たとえば、`Stack` クラスは `Vector` クラスからプライベートにできます。プライベートは、パブリックよりもににしています。

されたはほとんどされておらず、どのようながしているかについてなはありません。

をするは、`virtual` キーワードをできます。

```
struct A{};
struct B: public virtual A{};
```

クラス `B` が `A` につき、それは `A` がほとんどのクラスのツリーにすることをします。したがって、クラスのほとんどがそののをします。

```
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`
}
```

`/*A(88)*/` をコメントアウトすると、`C` は `A` しているのでエラーはしません。

また、`object` する `object`、ほとんどのクラスは `C` であるため、`C` は `A` コンストラクタのをし、`A::member` は `5` ではなく `88` であることにしてくださいこれは、タイプ `B` オブジェクトをします。

[ダイヤモンドの](#)をするときにです。

A	A	A
/ \		
B	C	C

\ /
D
virtual inheritance

\ /
D
normal inheritance

BとCのがA、びDからBびCので、の2つのインスタンスがするAでD これは、コンパイラがどのクラスからそのメンバBするものか、Cされるものにアクセスするかをるがないため、AからDメンバにアクセスするときになります。。

はこのをします。はほとんどのオブジェクトにのみするため、DはAインスタンスが1つしかありません。

```
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
    }
};
```

コメントをすると、あいまいさがされます。

のをいて

```
class A {};
class B : public A {};
```

また、のをつこともできます。

```
class A {};
class B {};
class C : public A, public B {};
```

CはAとBからにします。

のclassまたはstructでじがされているは、あいまいさにつながるがあります。してください
におけるあいまい

いくつかのケースではがつかかもしれませんが、をしているときにながすることがあります。

たとえば、2つのクラスには、クラスでオーバーライドされないじのがあり、クラスのオブジェクトをしてそのにアクセスするコードをすると、コンパイラはエラーをします。におけるこのタイプのあいまいさのコードをにします。

```

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( )
}

```

しかし、このは、スコープをして、base1またはbase2のいずれのクラスにどのをするかをすることができます。

```

int main()
{
    obj.base1::function( ); // Function of class base1 is called.
    obj.base2::function( ); // Function of class base2 is called.
}

```

クラスメンバーへのアクセス

クラスのオブジェクトのメンバーとメンバにアクセスするには、.がされます。

```

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();

```

ポインタをしてクラスのメンバにアクセスする、->がにされます。のとして、インスタンスをすることもできます.これはあまりではありませんが、

```

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();

```

クラスメンバーにアクセスするときは、`::`がされますが、クラスのインスタンスではなくクラスののにされます。わりに、スタティックメンバは、インスタンスまたはインスタンスへのポインタからをしてアクセスすることができます。または`->`であり、メンバーにアクセスするのとじです。

```

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();

```

バックグラウンド

メンバーアクセスであるため、`->`がです。`*`よりもされます。

つは、`*pa`であろう`p` オブジェクトへのをじる`p`とそのメンバーのアクセスにいています。`a`しかし、に`a`、`p`のメンバー`a`にアクセスしてからしようとしします。le `*pa`は`*(pa)`とです。まず、2つののため、のでは、これはコンパイラのエラーになり`p`ポインタで、メンバーはありません。`a2`に、`a`はであるため、することはできません。

こののめったにわれないは、をにすることです `(*p).a`

わりに、`->`はほとんどにされます。ポインタをにしてからアクセスするのは、いです。 `!e (*p).a` は `p->a` とまったく同じです。

`::`はスコープで、のメンバにアクセスするのと同じでされます。これは、クラスメンバーがそのクラスのスコープにあるとなされますが、そのクラスのインスタンスのメンバーとはみなされないためです。の.なから、インスタンスメンバではないにもかかわらず、メンバにも`->`がされています。これは、びしがのメンバがであるかであるかをするがないため、テンプレートにコードをするためにされます。

プライベートクラスインタフェースの

プライベートは、クラスのパブリックインターフェイスをするがあるにです。

```
class A {
public:
    int move();
    int turn();
};

class B : private A {
public:
    using A::turn;
};

B b;
b.move(); // compile error
b.turn(); // OK
```

このアプローチは、Aポインタまたはにキャストすることによって、Aパブリックメソッドへのアクセスをにします。

```
B b;
A& a = static_cast<A&>(b); // compile error
```

の、そのようなキャストは、Bをすることをけるためのにもかかわらず、すべてのAメソッドへのアクセスをします。

```
class B : public A {
private:
    int move();
};
```

またはプライベートをして

```
class B : public A {
private:
    using A::move;
};
```

どちらのもです

```
B b;
A& a = static_cast<A&>(b); // OK for public inheritance
a.move(); // OK
```

クラスと

C++ 11

クラスのは、`final` することができます。なクラスをしましょう

```
class A final {
};
```

これをサブクラスしようとするコンパイルエラーがします

```
// Compilation error: cannot derive from final class:
class B : public A {
};
```

なクラスは、クラスのどこにでもされます。

```
class A {
};

// OK.
class B final : public A {
};

// Compilation error: cannot derive from final class B.
class C : public B {
};
```

`friend` キーワードは、クラスのスコープでされていても、のクラスやにそのクラスのプライベートメンバーやされたメンバーへのアクセスをえるためにされます。

```
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

```

ネストされたクラス

`class` または `struct` は、の `class / struct` をむこともできます。これは "れになったクラス" とされます。ここでは、クラスは「クラス」とされます。ネストされたクラスは、むクラスのメンバーであるとなされますが、それのはです。

```

struct Outer {
    struct Inner { };
};

```

むクラスのからは、スコープをしてネストされたクラスにアクセスします。ただし、みクラスのからは、なしでれクラスをできます。

```

struct Outer {
    struct Inner { };

    Inner in;
};

// ...

Outer o;
Outer::Inner i = o.in;

```

ネストされていない `class / struct` に、メンバーとは、ネストされたクラスまたはそのむでできま

す。ただし、ネストされたクラスとはなるクラスとなされるため、それらをむクラスすることはできません。

```
// 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() {}
```

ネストされていないクラスとに、ネストされたクラスは、されるにされていれば、でしてすることができます。

```
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

C++ 11よりでは、ネストされたクラスは、むクラスからの、`static`メンバー、およびにしかアクセスできませんでした。むクラスにされているのすべてのメンバーはをけませんでした。

C++ 11

C++ 11では、ネストされたクラスおよびそのメンバーは、それらがむクラスの`friend`かのように

われ、のアクセスによってすべてのメンバーにアクセスできます。ネストされたクラスのメンバーが、むクラスの1つのメンバーをするをとするは、インスタンスをすがあります。

```
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;
};
```

に、むクラスは、れになったクラスのフレンドとしてわれていないので、にをされることなく、そのプライベートメンバーにアクセスすることはできません。

```
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.
    }
};
```

ネストされたクラスのフレンドは、にそのクラスのフレンドとはみなされません。らがむクラスなのであるがある、これは々にするがあります。に、むクラスはネストされたクラスのフレンドとはみなされないなので、むクラスのフレンドはネストされたクラスのフレンドとはみなされません。

```
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.
}
```

のすべてのクラスメンバーとに、ネストされたクラスは、パブリックアクセスがあるにのみクラスのからをけることができます。ただし、にをけないうり、アクセスになくアクセスできます。

```
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.
```

ネストされたクラスのエイリアスをすることもできます。みクラスにエイリアスがまれている、れとエイリアスはなるアクセスをつことができます。エイリアスがんでいるクラスのにあるは、れになったクラスまたはそのtypedef するがありtypedef。

```
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.
```

のクラスとに、ネストされたクラスは、のクラスからしたり、したりすることができます。

```

struct Base {};

struct Outer {
    struct Inner : Base {};
};

struct Derived : Outer::Inner {};

```

これは、にじてプログラマがネストされたクラスをできるようにすることで、むクラスがのクラスからしているでです。これをtypedefとみわせて、むクラスのネストされたクラスにしたをけることができます

```

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();

```

の、BaseOuterとDerivedOuterメンバーInnerとして、BaseInner_とDerivedInner_それぞれ。これにより、ネストされたをむクラスのインタフェースをさずにさせることができ、ネストされたをにすることができます。

メンバータイプとエイリアス

classまたはstructは、クラスにまれ、そのクラスのメンバとしてわれるエイリアスであるメンバエイリアスをすることもできます。

```
struct IHaveATypedef {
    typedef int MyTypedef;
};

struct IHaveATemplateTypedef {
    template<typename T>
    using MyTemplateTypedef = std::vector<T>;
};
```

メンバーとに、これらのtypedefは、スコープ::をしてアクセスされます。

```
IHaveATypedef::MyTypedef i = 5; // i is an int.

IHaveATemplateTypedef::MyTemplateTypedef<int> v; // v is a std::vector<int>.
```

のエイリアスとに、メンバエイリアスは、そののにされているか、またはにエイリアスされているのをすることができます。に、クラスのにあるtypedefは、クラスのもであれば、クラスのアクセスなtypedefをできます。

```
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;
};

IHaveTypedefs::MyTypedef i; // x_i is an int.
IHaveTypedefs::MyTypedefHelper hi; // x_hi is a Helper<int>.

typedef IHaveTypedefs::MyTypedef TypedefBeFree;
TypedefBeFree ii; // ii is an int.
```

メンバータイプのエイリアスは、のアクセスレベルででき、なアクセスをします。

```
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.
};

```

これはレベルをするためにすることができ、クラスのは、のをそれにするコードをることなくすることができます。

```

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();

```

このようなで、ヘルパークラスがSomeComplexTypeからのにされた、typedefとfriendだけをするがあります。ヘルパークラスがしをするり、でするのではなく、Something::MyHelperとしてするコードは、をえずにそのままします。このようにして、ながされたときにするがあるコードのをにえ、を1でするだけです。

decltypeとみわせることもできます。

```

class SomethingElse {
    AnotherComplexType<bool, int, SomeThirdClass> helper;

public:
    typedef decltype(helper) MyHelper;

private:
    InternalVariable<MyHelper> ivh;

    // ...

public:

```

```
MyHelper& get_helper() const { return helper; }

// ...
};
```

このようなでは、`SomethingElse::helper`をすると、`decltype`のために`typedef`がされ`decltype`。これにより、`helper`をしたいときにながにえられ、ミスのリスクがにえられます。

しかし、すべてとに、これはあまりにもくにられるがあります。タイプがに1または2だけされ、に0されるなどは、をすることはありません。プロジェクトでまたはされている、またはがにいは、ににするのではなく、`typedef`としてするとです。フォワードのとのバランスを、なノイズのとわせなければなりません。

これは、クラスでのテンプレートパラメータへのアクセスをするために、テンプレートクラスでもできます。

```
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);
```

これは、コンテナでされます。コンテナは、のヘルパーをメンバーエイリアスとしてします。たとえば、C++ライブラリのほとんどのコンテナは、なのとともに、の12のヘルパーをします。

```
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;
};

```

C++ 11よりのバージョンでは、これはまだできなかったため、ソートの「テンプレート_{typedef}」をするためによくされていました。これらはエイリアステンプレートのでしになっていますが、いくつかのではまだですそして、のではエイリアステンプレートとみわされています。なの々のコンポーネント、。らは、のタイプエイリアスに_{type}をします。

```

template<typename T>
struct TemplateTypedef {
    typedef T type;
}

TemplateTypedef<int>::type i; // i is an int.

```

これは、1つのパラメータをするエイリアスをするために、のテンプレートパラメータをつでよくされていました。

```

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>.

```

クラスのメンバー

クラスには、またはのいずれかの_{static}メンバーもめることができます。これらはクラスのスコープにあるとなされますが、のメンバーとしてはわれません。それらはなプログラムのめからわりまでしますをち、クラスののインスタンスにびついておらず、クラスで1つのコピーしかしません。

```

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)

```

メンバーは、クラスでされているとはみなされず、されているだけであるため、クラスのにされます。プログラマは、のをすることはできますが、ではありません。メンバをするとき、キーワード `static` はされます。

```

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.

```

このため、は、なとしてでされるり、 `void` いてなになるがあります。

```

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];

```

メンバは、のメンバとに、クラスのまたはですることができます。メンバのとに、メンバをクラ

スのにする、キーワード `static` はされます。

```
// 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; }
```

メンバーが `const` でされていて `volatile` ではなく、またはである、クラスのことすることができます。

```
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

C++ 11では、`LiteralType`のメンバ `constexpr` によってコンパイルにできるも `constexpr` としてできます。そのは、クラスですがあります。

```
struct ExConstexpr {
    constexpr static int ci = 5;           // Good.
    static constexpr double cd = 5;        // Good.
    constexpr static int carr[] = { 1, 1, 2 }; // Good.
    static constexpr ConstexprConstructibleClass c{}; // Good.
    constexpr static int bad_ci;           // Error.
};

constexpr int ExConstexpr::bad_ci = 5;     // Still an error.
```

`const` または `constexpr` メンバが *odr-used* アドレスがられているか、またはにりてられている、に、

クラスののにのがっているがあります。このにはをめることはできません。

```
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.
```

メンバーはのインスタンスにびけられていないため、スコープ::をしてアクセスできます。

```
std::string str = Example::static_str;
```

でないのメンバーであるかのようにアクセスすることもできます。これはになをちますが、メンバーがであるかであるかのをけるために、スコープよりもあまりではありません。

```
Example ex;
std::string rts = ex.static_str;
```

クラスメンバは、クラスメンバののに、スコープをせずにメンバにアクセスできます。

```
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;
```

らはmutableすることはできませんし、らがするがあります。のインスタンスにけられていないため、インスタンスがconstであるかどうかにかかわらず、メンバーにはしません。

```
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.
```

メンバーは、メンバーとにアクセスをします。

```
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.
```

えられたインスタンスにされていないので、メンバーには`this`ポインタはありません。このため、インスタンスをさないり、メンバーにアクセスすることはできません。

```
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.
};
```

`this`ポインタをたないため、そのアドレスはメンバへのポインタにすることはできず、わりののポインタへのポインタにされます。

```
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.
```

`this`ポインタをたないため、`const`や`volatile`でも`const`ませんし、`ref-qualifier`をつこともできませ

ん。らはまた、にすることはできません。

```
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.
};
```

えられたインスタンスにびけられていないので、メンバーはになグローバルとしてわれます。プログラムのにされ、クラスのインスタンスがにするかどうかになく、にされます。が`thread_local` C++ 11としてされているをき、メンバのコピーが1つだけしますこの、スレッドごとに1つのコピーがあります。

メンバーには、クラスにリンクまたはリンクがあるかどうかにかかわらず、クラスとじリンクがあります。ローカルクラスおよびのないクラスは、メンバーをつことはできません。

メンバー

クラスには、クラスの々のインスタンスでするメンバーをたせることができます。

```
class CL {
public:
    void member_function() {}
};
```

これらは、のように、クラスのインスタンスでびされます。

```
CL instance;
instance.member_function();
```

クラスのまたはでできます。でされているは、クラスのスコープにされます。

```
struct ST {
    void defined_inside() {}
    void defined_outside();
};
void ST::defined_outside() {}
```

らはすることができCVおよび/またはREF-、らはらはにびされているインスタンスをしてくださいどのようにをえます。はインスタンスがされたcv-qualifierをつものとなしますする。びされるバージョンは、インスタンスのcvについています。インスタンスとじcv-qualifierをつバージョンがない、であれば、よりくのcvバージョンがびされます。

```

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

メンバのrefは、がrvalueインスタンスでびされることをしているかどうかをし、cv-qualifiersとじをします。

```

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とrefは、にじてみわせることもできます。

```

struct BothCVAndRef {
    void func() const& {} // Called on normal instances. Sees instance as const.
    void func() && {} // Called on temporary instances.
};

```

それらは[でもあります](#)。これはのであり、クラスがのをしながら、クラスとじインターフェースをすることをにします。

```

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().

```

については、[こちら](#)をしてください。

の|クラス

の`struct`がされていますにありません

```
void foo()
{
    struct /* No name */ {
        float x;
        float y;
    } point;

    point.x = 42;
}
```

または

```
struct Circle
{
    struct /* No name */ {
        float x;
        float y;
    } center; // but a member name
    float radius;
};
```

```
Circle circle;
circle.center.x = 42.f;
```

の`struct`ませんのないおよびのないオブジェクト

```
struct InvalidCircle
{
    struct /* No name */ {
        float centerX;
        float centerY;
    }; // No member either.
    float radius;
};
```

のコンパイラでは、の`struct`をとしてできます。

C++ 11

- `decltype`はなのない`struct`とることができます。
- `decltype`はの`struct`のをりすことができます

```
decltype(circle.point) otherPoint;
```

- のない`struct`インスタンスは、テンプレートメソッドのパラメータになります。


```

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";
}

```

オンラインでクラス/をむ <https://riptutorial.com/ja/cplusplus/topic/508/クラス->

55: このポインタ

`this`ポインタは、これをするためになライブラリがないため、C++のキーワードです。 `this`がポインタであることを知らないでくださいそうすることはできません

```
this.someMember();
```

->をして、ポインタからメンバーまたはメンバーにアクセスすると

```
this->someMember();
```

`this`ポインタのをめるためののにつリンク

「`this`」ポインタとはですか

<http://www.geeksforgeeks.org/this-pointer-in-c/>

https://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm

Examples

このポインタ

すべてのメンバには、 `this`のクラスのインスタンスへのポインタであるしパラメータがあります。このパラメータはかにパラメータリストのにされ、コンパイラによってにされます。クラスのメンバがメンバのでアクセスされると、それをしてかにアクセスされ `this`。コンパイラはすべてのインスタンスにしてのメンバをでき、メンバはのメンバをにびすことができます。

```
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; }

```

コンストラクタでは、`this`にクラスですでにされているのフィールド、またはのフィールドにアクセスするまたはにすることができます。に、まだされていないフィールドやクラスのフィールドにはにまたはにではありませんクラスがまだされていないため、フィールドももありません。それをしてメンバをびすこともであり`this`いずれかのクラスのがされないように、コンストラクタにこれはまだされていないクラスに、したがってそのコンストラクタはまだテーブルをしません。

また、コンストラクタでは、オブジェクトのはそのコンストラクタがするです。これは、オブジェクトがとしてされていてもてはまります。えば、のでは、`ctd_good`と`ctd_bad`で`CtorThisBase` `CtorThisBase()`およびタイプ`CtorThisCtorThis()`、それらのであっても、`CtorThisDerived`。よりしたクラスがクラスのりにされると、インスタンスは、されたのにされたインスタンスになるまで、々にクラスをします。

```

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);

```

これらのクラスとメンバをすると

- いコンストラクタでは、 `ctd_good` ために
 - `CtorThisBase` は、 `CtorThis` コンストラクターがされるまでににされます。したがって、 `s` しつつ、 `na` である `i`、したがってアクセスすることができます。
 - `i` が `j(this->i)` にするにされます。したがって、 `i` は `j` をしているは `na` にあり、アクセスすることができます。
 - `j` は `k(j)` にするにされる。したがって、 `j` は `k` をしている `na` にあり、 `tt` アクセスすることができる。
- なコンストラクタでは、 `ctd_bad`
 - `k` は `j(this->k)` にしたにされる。したがって、 `j` をしている、 `k` は `na` にあり、それにアクセスするとのがします。
 - `CtorThisDerived` は、 `CtorThis` がされるまでされません。したがって、 `b` は `na` にあり、 `k` をし、それにアクセスするとのがします。
 - オブジェクト `ctd_bad` まだある `CtorThis` それがるまで `CtorThis()` およびするようにされることはありません `CtorThisDerived` までの `vtable` のを `CtorThisDerived()` したがって、 `virt_func()` はそれをびすか、 `CtorThisDerived::virt_func()` をびすかにかかわらず、 `CtorThis::virt_func()` をびします。

このポインタをしてメンバーデータにアクセスする

このコンテキストでは、 `this` ポインタのはにはではありませんが、されたまたはがクラスのメンバーであることをすことによって、コードをみににします。このの

```

// 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;
}

```

それを[ここの](#)で[ください](#)。

このポインタをしてメンバーデータとパラメータをする

これはメンバーデータをパラメータとするためののなです...このをえてみましょう

```

// 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();
}

```

ここでは、コンストラクタでをします

```
this->name = name;
```

ここでは、クラスDogthis-> nameのプライベートのにパラメータをしていることがわかります。

のコードのをするには <http://cpp.sh/75r7>

このポインタ**CV-**

thisはのポインタとじcvされたものでもあります。ただし、thisパラメータがパラメータリストにリストされていないため、thisためにながです。cvは、パラメータリストの、ののにリストされます。

```

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*
};

```

thisはパラメータなので、this cv-qualifierについてをオーバーロードすることができます。

```

struct CVOverload {
    int func()           { return 3; }
    int func() const     { return 33; }
}

```

```
int func() volatile      { return 333; }
int func() const volatile { return 3333; }
};
```

この`const const volatile`をむの、またはにわらず、はメンバーにきむことができません。のは`mutable`メンバであり、にかかわらずできます。このため、`const`は、なオブジェクトがフードのどのようにえるかをしても、メンバがオブジェクトのオブジェクトがにれるをしないことをするためにされます。

は、オブジェクトがオブザーバーのにするです。それはにびついておらず、にはとしてされていないかもしれません。のがをることができないうり、オブジェクトのすべてのビットをしても、はです。

ビットのともばれるは、オブジェクトがメモリにどのようにされるかです。これは、データをするの1と0のオブジェクトです。オブジェクトは、メモリのがしてしない、にのみです。

C++ことにしてください`const`ではなく、なにネス。

```
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;
}
```

には `this` を `const_cast - const_cast` ですが、ににににのはずですが、わりに `mutable` をうべきです。 `const_cast` は、に `const` であるオブジェクトにしてされるときはのをびすのにし、 `mutable` はにするようにされています。しかし、にコードでこれをするがあります。

こののは、 `const` アクセサーのから `cv` アクセサーをすることです。 `cv` されていないバージョンがびされた、オブジェクトが `const` ないことがされるため、UBのリスクはありません。

```
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));
    }
};
```

これにより、なコードのがされます。

のポインタとに、 `this` が `volatile const volatile` をむの、キャッシュされずにアクセスされるたびにメモリからロードされます。これは、のポインタが `volatile` とするのと同じようににしますので、がです。

インスタンスが `cv` されている、アクセスがされるメンバは、 `this` ポインタがなくともインスタンスの `cv` されたメンバだけであることにしてください。

- `cv` インスタンスは、のメンバにアクセスできます。
- `const` インスタンスは、 `const` および `const volatile` にアクセスできます。
- `volatile` インスタンスは `volatile` および `const volatile` にアクセスできます。
- `const volatile` インスタンスは `const volatile` にアクセスできます。

これは `const` さのの1つです。

```
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.

```

このポインタ

C++ 11

`this` `cv`とに、`ref`を`*this`もできます。REF-はコンパイラがコピーのいずれかをするか、よりであるかにじてセマンティクスをできるように、とセマンティクスのものであるためにされ、そしてするためにされる`*this`わりに`this`。

をする`ref-qualifier`にもかかわらず、`this`はとしてポインタであることにしてください。また、`ref`は`*this`をにしないことにしてください。らのようにて、そのをしてするほうがです。

```

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

```

メンバは`ref-qualifier`のにかかわらずをつことはできません。プログラマはどちらかをするがあります。ありがたいことに、`cv`は`ref`とみわけてすることができ、`const`をすることができます。

```

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&& {}  
};
```

オンラインでこのポインタをむ <https://riptutorial.com/ja/cplusplus/topic/7146/このポインタ>

56: コンパイルとビルド

き

C++でかかれたプログラムは、するにコンパイルするがあります。オペレーティングシステムによっては、さまざまなコンパイラがされています。

ほとんどのオペレーティングシステムはコンパイラなしでされ、でインストールするがあります。なコンパイラのはのとおりです。

- [GCC、GNUコンパイラ・コレクション g++](#)
- [clangLLVM clang++の Cファミリのフロントエンド](#)
- [MSVC、Microsoft Visual C++Visual Studioに visual-c++](#)
- [C++ Builder、Embarcadero C++ Builder RAD Studioに c++ビルダー](#)

C++プログラムのコンパイルについては、なコンパイラマニュアルをしてください。

のコンパイラをののビルドシステムでするのオプションとして、[ビルドシステムに](#)のコンパイラまたはデフォルトのインストールされたものにプロジェクトをさせることができます。

Examples

GCCでコンパイルする

main.cppというののソースファイルをする、と、されていないファイルをコンパイルしてリンクするコマンドはのようになりますなしのコンパイルはのとデバッグにはですが、しいGCCバージョンでは`-Og`がにされます。

```
g++ -o app -Wall main.cpp -O0
```

でするされたファイルをするには、`-O`オプションの1つをします `-O1`、`-O2`、`-O3`、`-Os`、`-Ofast`。

```
g++ -o app -Wall -O2 main.cpp
```

`-O`オプションをすると、をわないことをする`-O0`がデフォルトとしてされます`-O`をしない、`-O1`にされます。

あるいは、`-O`グループのフラグまたはよりくのをしてください。のは、`-O2`レベルと`-O3`レベルの1つのフラグでされてい`-O3`。

```
g++ -o app -Wall -O2 -ftree-partial-pre main.cpp
```

プラットフォームのされたファイルをするには、アーキテクチャのマシンでのため、の
ようにします。

```
g++ -o app -Wall -O2 -march=native main.cpp
```

いずれのも、Windowsでは、`.\app.exe`、Linux、Mac OSでは、`./app`でできるバイナリファイル
がされます。

`-o`フラグもスキップできます。この、GCCはデフォルトのファイル `a.exe` をWindowsにし、`a.out`
をUNIXのシステムにします。ファイルをリンクせずにコンパイルするには、`-c`オプションをし
ます。

```
g++ -o file.o -Wall -c file.cpp
```

これにより、このファイルとリンクしてバイナリをすることができる `file.o` というオブジェ
クトファイルがされます。

```
g++ -o app file.o otherfile.o
```

オプションについては、gcc.gnu.orgをしてください。になのは、`-Og` デバッグをした - の - コ
ンパイル - デバッグサイクルにと `-Ofast` すべての、なをしたものをむです。

`-Wall` フラグは、くのなエラーのをにします。にするがあります。コードのをさせるために、くの
、することもされている `-Wextra` によってになっていないと、のフラグ `-Wall` と `-Wextra`。

コードがのC++をしているは、`-std=`フラグをめること、するをします。サポートされている
は、ISO C++のバージョンのにしています。GCC 6.1.0、`std=`フラグのなは、`c++98` / `c++03`、
`c++11`、`c++14`、および `c++17` / `c++1z` です。スラッシュでられたはです。

```
g++ -std=c++11 <file>
```

GCCには、`-std=`フラグでされたとするとになるコンパイラのがいくつかまれています。すべて
のをにしてコンパイルするには、`gnu++XX` どこをいてもよい `XX` でされるのいずれかである `c++` のを。

されていない、デフォルトのがされます。バージョン6.1.0よりのバージョンのGCCでは、デフォ
ルトは `-std=gnu++03` です。GCC 6.1.0では、デフォルトは `-std=gnu++14` です。

GCCのバグのために、`std::thread` や `std::wait_for` など、C++ 11でされたC++のスレッドをサポ
ートするために、GCCのコンパイルとリンクに `-pthread` フラグがであることにしてください。ス
レディングをしているときにすると、はされず、のプラットフォームではなになることがあり
ます。

ライブラリとのリンク

ライブラリをすには、`-l`オプションをします。

```
g++ main.cpp -lpcr2-8
#pcr2-8 is the PCRE2 library for 8bit code units (UTF-8)
```

ライブラリがライブラリパスにないは、`-L`オプションをけてパスをします。

```
g++ main.cpp -L/my/custom/path/ -lmylib
```

のライブラリをリンクすることができます

```
g++ main.cpp -lmylib1 -lmylib2 -lmylib3
```

1つのライブラリがのライブラリにするは、ライブラリのにライブラリをします。

```
g++ main.cpp -lchild-lib -lbase-lib
```

またはリンカーをして、をさせる`--start-group`と`--end-group` これはなパフォーマンスコストがあります

```
g++ main.cpp -Wl,--start-group -lbase-lib -lchild-lib -Wl,--end-group
```

Visual C ++でのコンパイルコマンドライン

GCCやClangからVisual Studioにるプログラマ、あるいはなコマンドラインのがなプログラマにとっては、Visual C ++コンパイラはコマンドラインからもIDEでもできます。

Visual Studioでコマンドラインからコードをコンパイルするは、まずコマンドラインをするがあります。これは、 [Visual Studio Command Prompt / Developer Command Prompt / x86 Native Tools Command Prompt / x64 Native Tools Command Prompt](#)または Visual Studioのバージョンでされる [のx64 Native Tools Command Prompt](#) つか、コマンドプロンプトでVCコンパイラのインストールディレクトリのサブディレクトリ `\Program Files (x86)\Microsoft Visual Studio x\VC、 x`のようなバージョンある10.0 2010、または14.0 2015とランニング `VCVARSALL` でバッチファイルをコマンドラインパラメータは[ここでします](#)。

Visual Studioでは、GCCとはなり、コンパイラ `cl.exe` をしてリンカ `link.exe` のフロントエンドをするのではなく、コンパイラがにびすのプログラムとしてリンカをしています。 `cl.exe`と`link.exe` は、々のファイルとオプションで々にできます。また、 `cl`は、のタスクがにされたに`link`するファイルとオプションをすようできます。 `cl`された`link`オプションは`link`オプションにされ、 `cl`されないファイルは`link`されます。これはにVisual Studioのコマンドラインでコンパイルするためのなガイドですので、 `link`はではされません。リストがなは、 [こちらをごください](#)。

`cl`へのはとをし、 `link`するはそうではないことにしてください。

[パスをするは、Windowsシエルの "カレントディレクトリ"%cd%することをおめします。このにれ

ていないは、のディレクトリにされます。コマンドラインからは、`cl`をしたときのディレクトリになり、デフォルトでコマンドプロンプトでされますたとえば、コマンドプロンプトが`C:\src>`、`%cd%`は`C:\src\`。]

のフォルダに`main.cpp`というののソースファイルがあるとすると、されていないファイルをコンパイルしてリンクするコマンドのとデバッグにですはのいずれかです

```
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.
```

のソースファイル "`niam.cpp`"がじディレクトリにあるとして、をしてください

```
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".
```

また、のようにワイルドカードをすることもできます。

```
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".
```

ファイルのをまたはするには、のいずれかをします。

```
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".
```

`/o`と`/Fe`ともに、にじてなは`.exe`または`.dll`を`"name"` `o-param`にして、`/OUT:o-param`としてlinkする

パラメータ `o-param` とぶをします。 `/o` と `/Fe` はどちらもがじですが、Visual Studio ではがされます。
`/o` はとマークされており、にGCCやClangにしいプログラマーけにされているようです。

`/o` としてフォルダおよび/またはののスペースはオプションですが、`/Fe` としてフォルダおよび/またはのにはスペースをれることはできません。

に、されたファイルをするには、のようになります。

```
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.
```

に、プラットフォームのされたファイルされたアーキテクチャをつマシンのVCVARSALL するため
VCVARSALL するには、ターゲットプラットフォームのなコマンドプロンプトまたはVCVARSALL パラメ
ータをします。 `link` は、オブジェクトファイルからのプラットフォームをするがあります。そう
でない、 `/MACHINE` オプションをしてターゲットプラットフォームをにします。

```
// If compiling for x64, and LINK doesn't automatically detect target platform:
cl main.cpp /link /machine:X64
```

のいずれのもの、 `/o` または `/Fe` でされたのファイルがされます。どちらもされていないは、コンパイ
ラにされたのソースファイルまたはオブジェクトファイルとじがけられます。

```
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".
```

リンクせずにファイルをコンパイルするには、をします。

```
cl /c main.cpp
// Generates object file "main.obj".
```

これは、`link`をびさずにするように`cl`にし、オブジェクトファイルをします。このファイルはこのファイルとリンクしてバイナリをすることができます。

```
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".
```

にもなコマンドラインパラメータがあります。これはユーザがっておくとにです

```
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).
```

* nixシステムやGCC / Clangにしているなら、`cl`、`link`、およびそのVisual Studioコマンドラインツールは、スラッシュ `/c`などのわりにハイフン `-c`などでしたパラメータをけることができます。さらに、Windowsではスラッシュまたはバックスラッシュがなパシリとしてされるため、* nixのパスもできます。これにより、なコンパイラコマンドラインを`g++`や`clang++`から`cl`ににすることができます。


```
g++ -o app src/main.cpp
cl -o app src/main.cpp
```

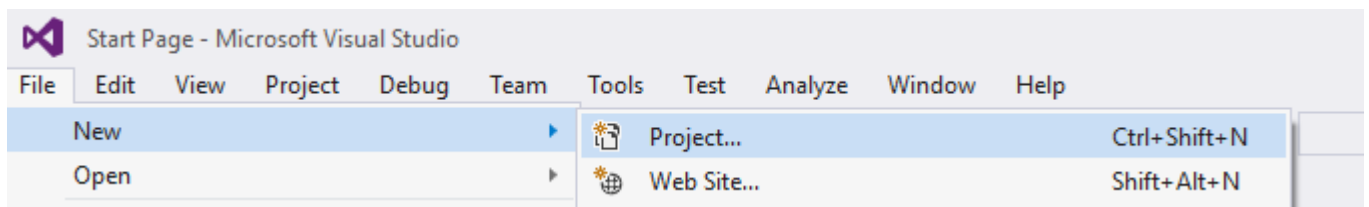
もちろん、よりな`g++`や`clang++`オプションをするコマンドラインをするは、するコンパイラのドキュメントやリソースサイトでのコマンドをするがありますが、これにより、しいコンパイラ。

コードにのがなは、MSVCののリリースがでした。 [Visual C ++ 2015 Update 3](#)では、 `/std` フラグをしてコンパイルするのバージョンをすることができます。なは、 `/std:c++14`および `/std:c++latest` `/std:c++17`がまもなくきますです。

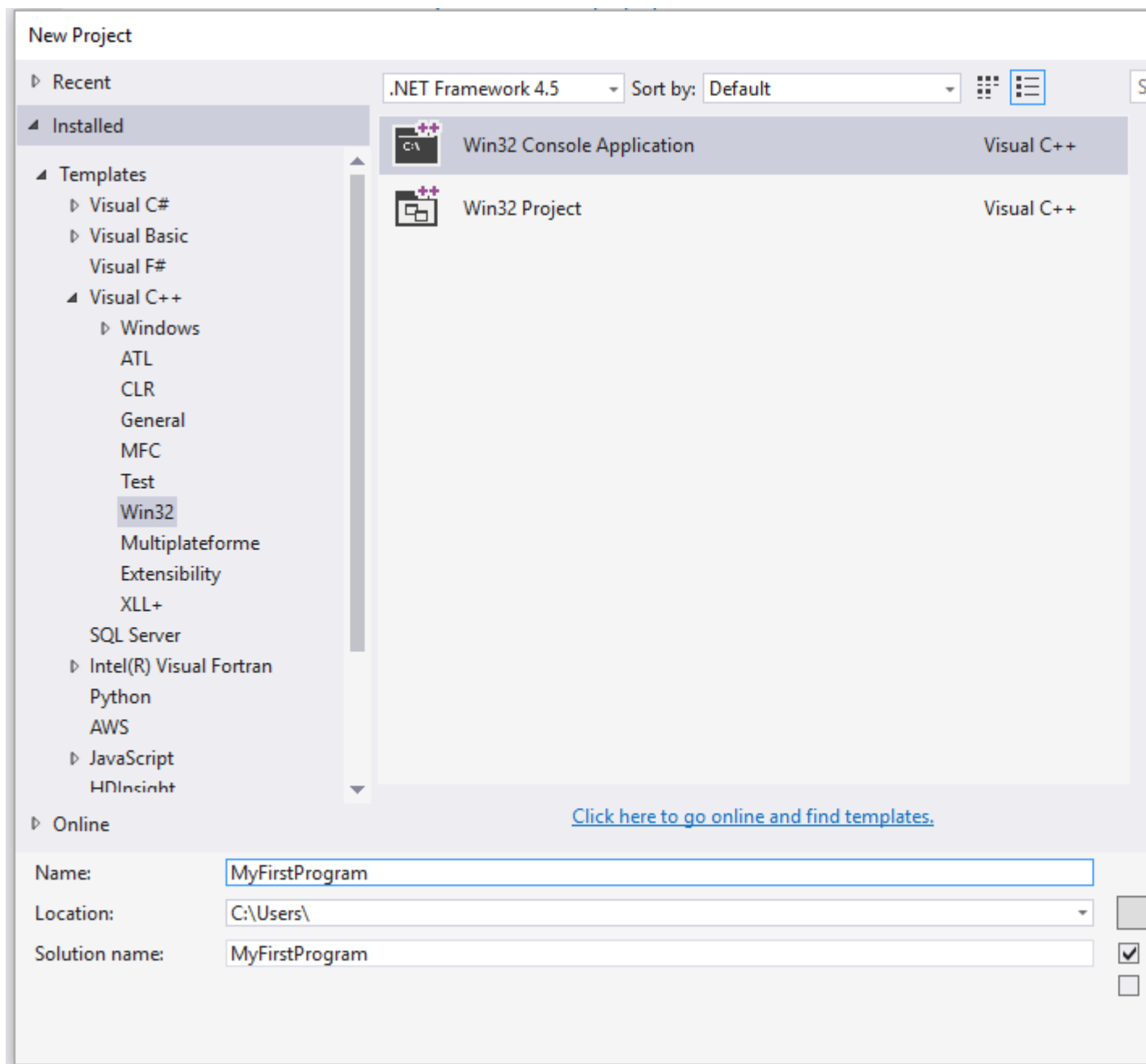
このコンパイラのバージョンでは、のフラグがでしたが、これはにのプレビューにされていました。

Visual Studio グラフィカル インターフェイスでコンパイルする - Hello World

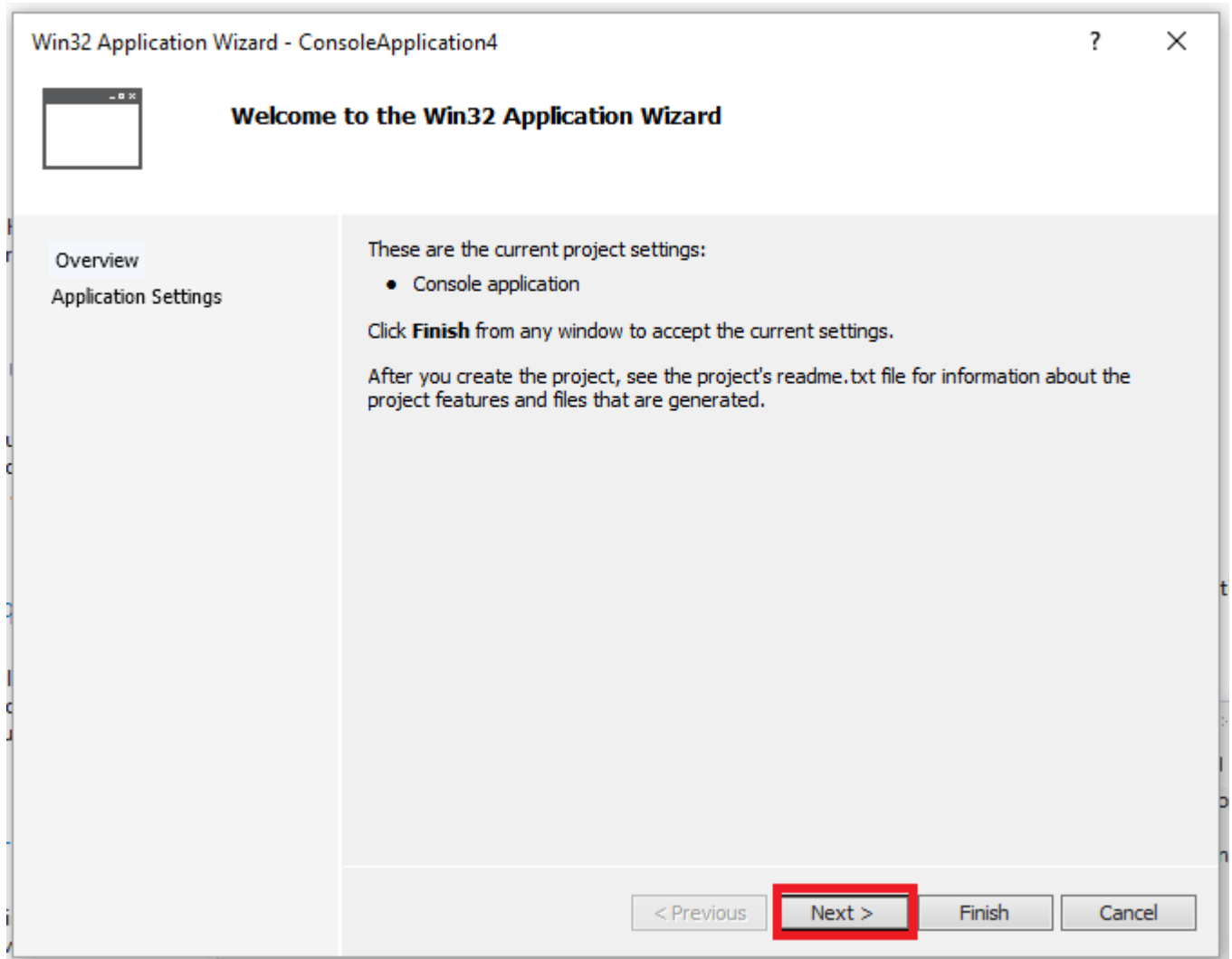
1. [Visual Studio Community 2015](#)をダウンロードしてインストールする
2. Visual Studio コミュニティをく
3. 「ファイル」 -> 「」 -> 「プロジェクト」をクリックします。



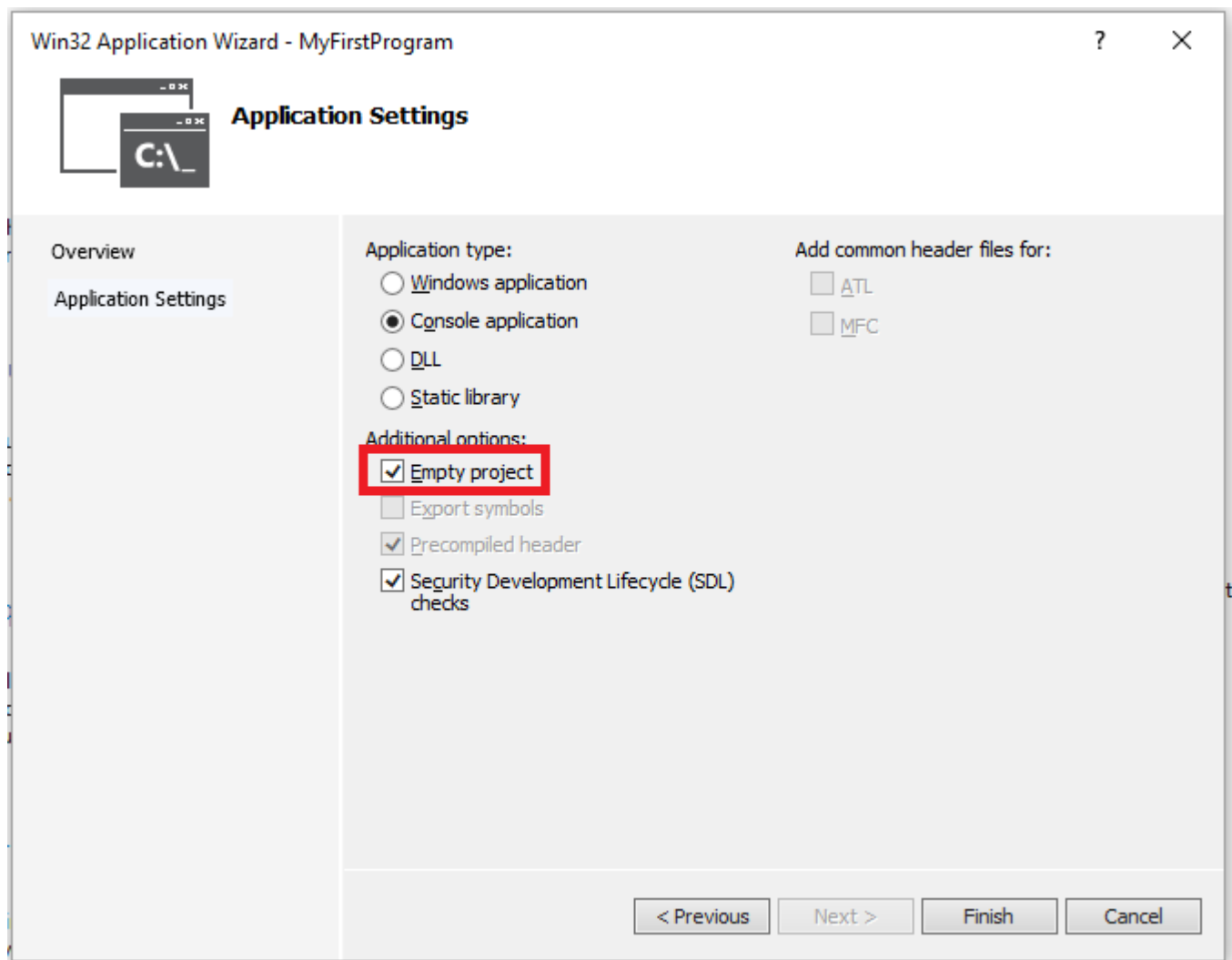
4. [テンプレート] -> [Visual C ++] -> [Win32 コンソールアプリケーション]をクリックし、**MyFirstProgram**というをけます。



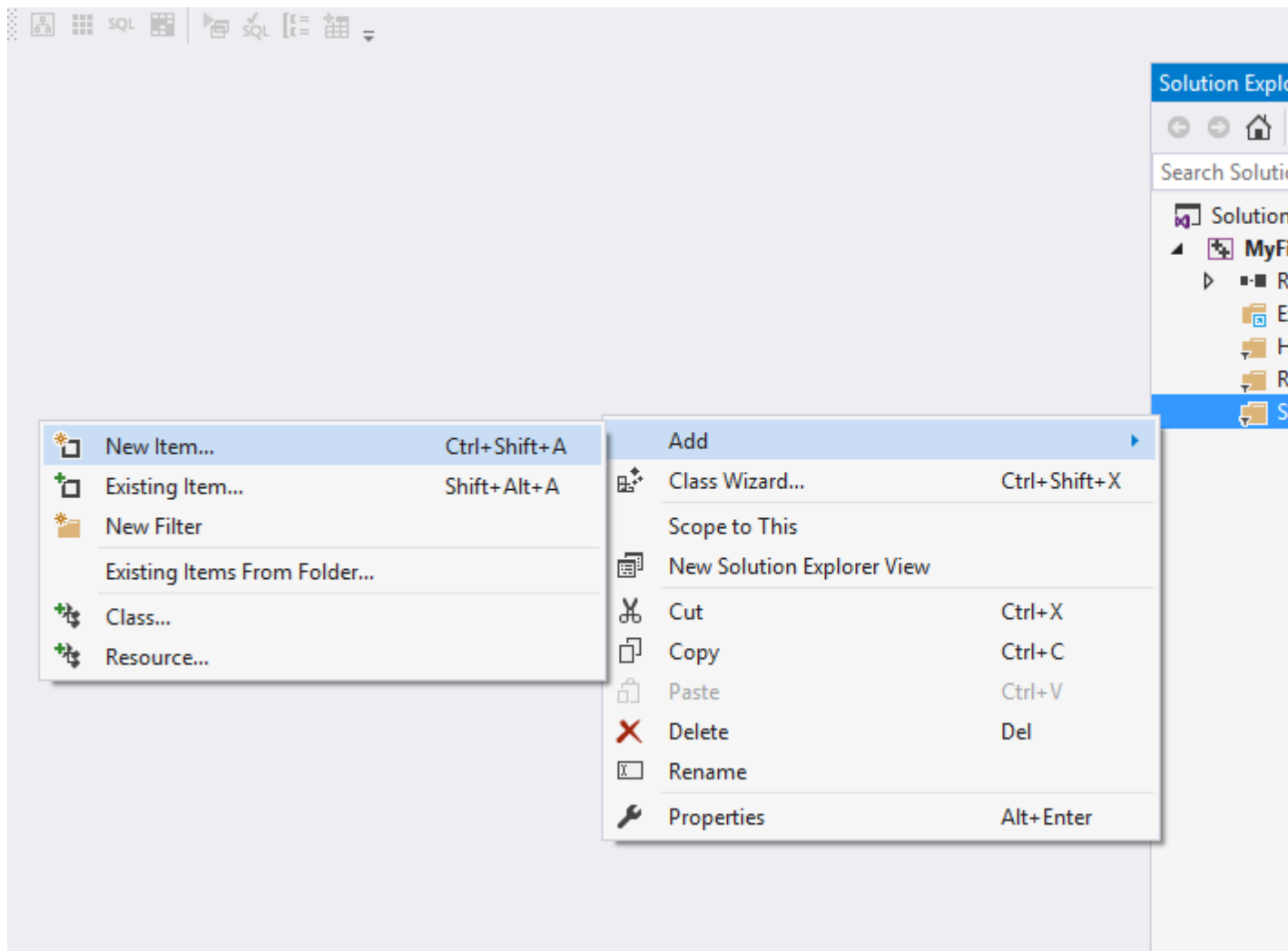
5. OKをクリックします。
6. のウィンドウで[へ]をクリックします。



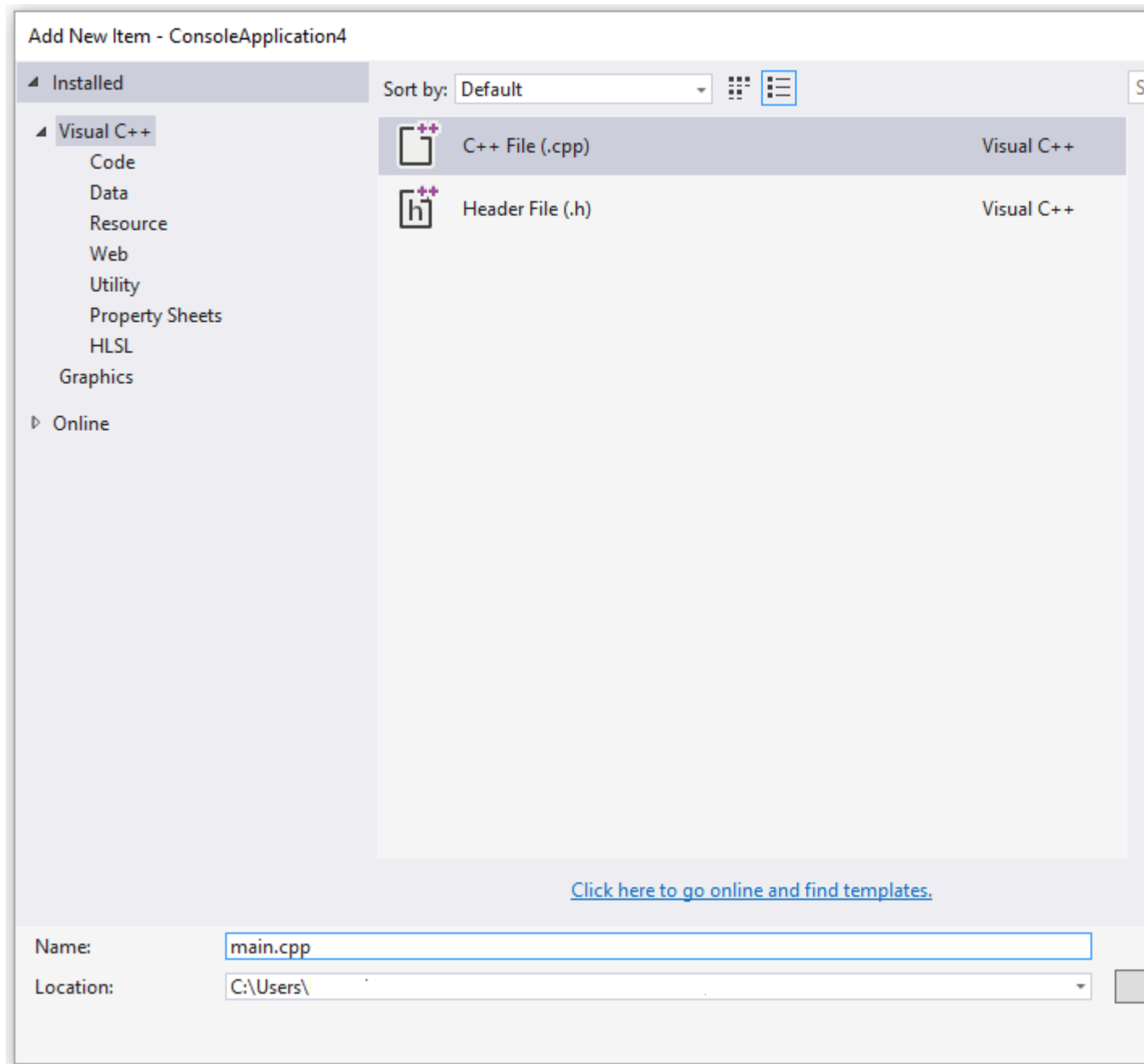
7. [Empty project] チェックボックスをオンにし、[]をクリックします。



8. ソースファイルをクリック -> ->しい



9. C ++ファイルをし、main.cppファイルにをけて、「」をクリックします。

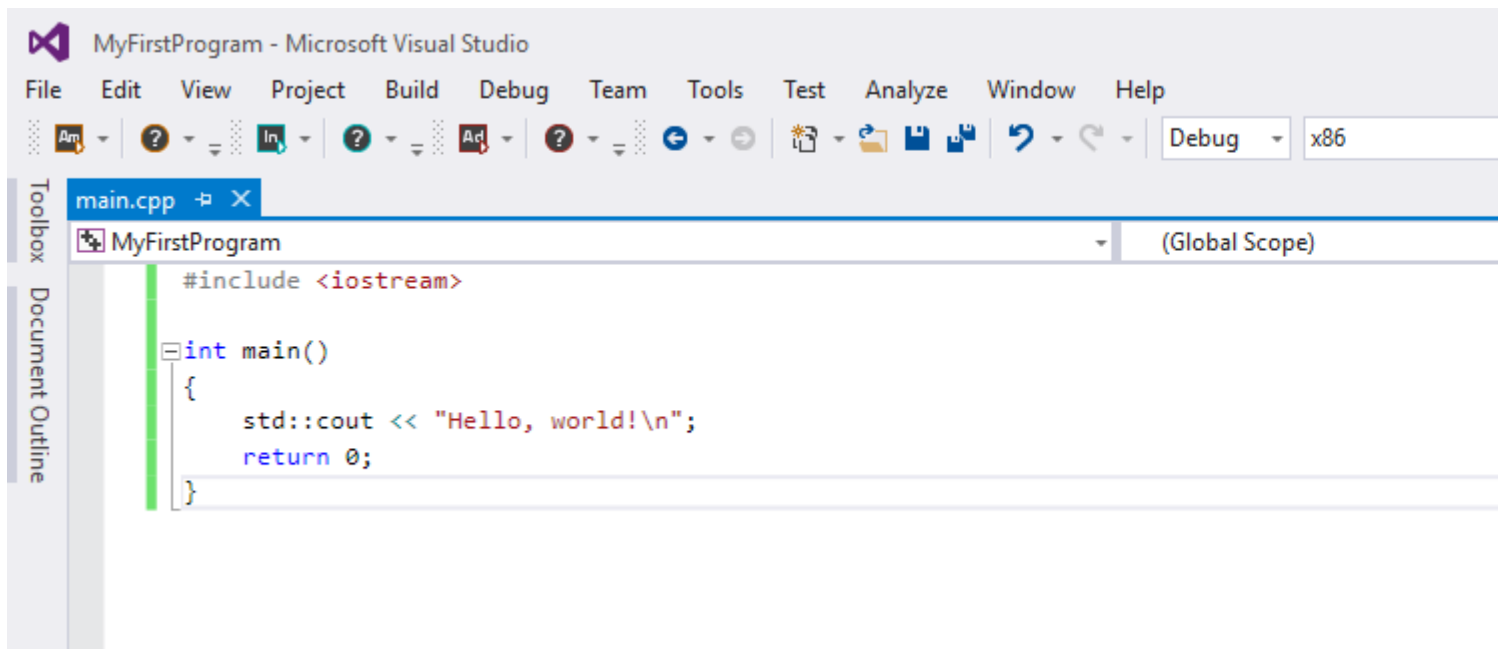


10のコードをコピーして新しいファイルmain.cppに貼り付けます。

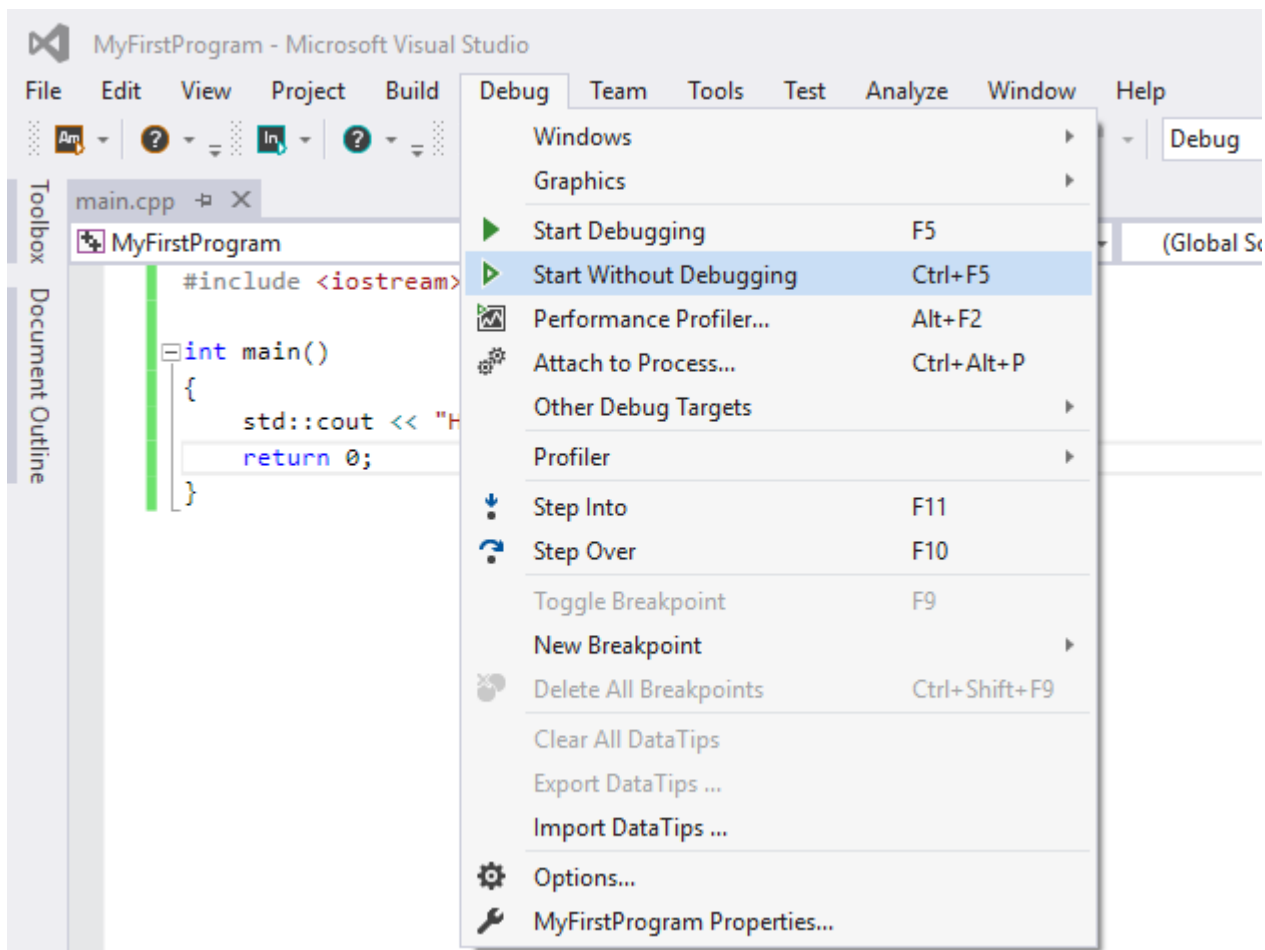
```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

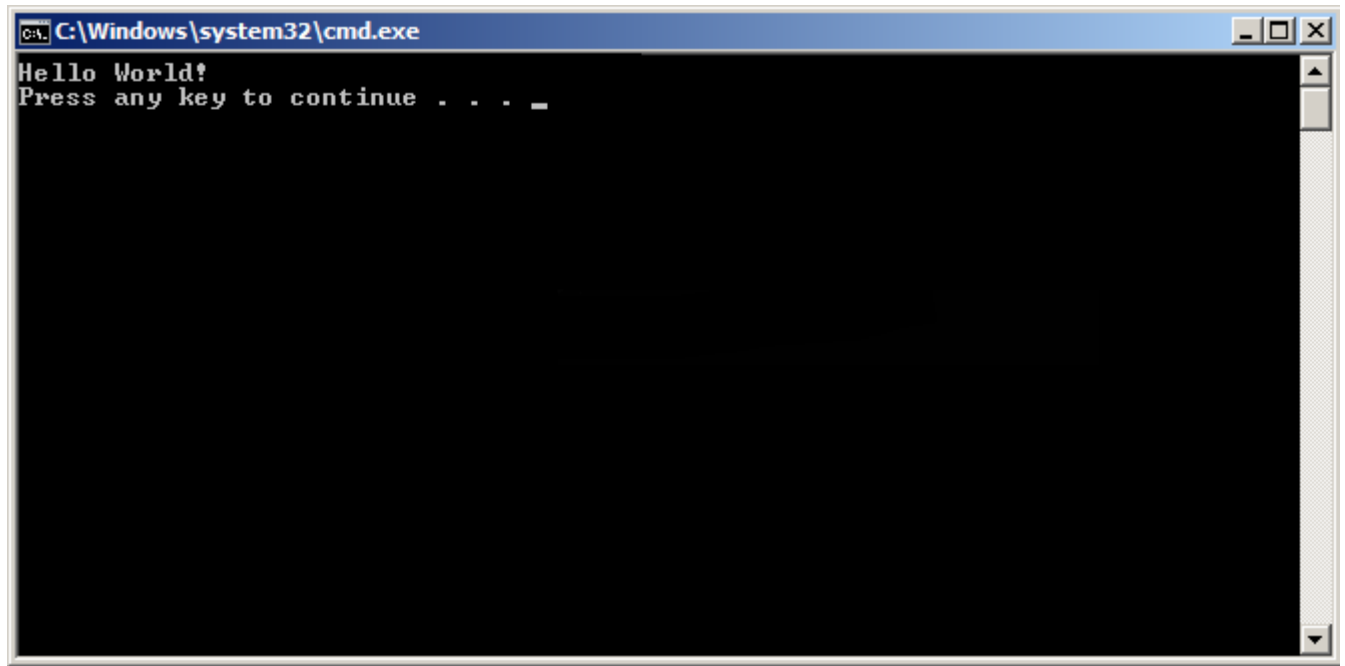
あなたのはのようになります。



11. デバッグ -> デバッグなしでまたは **ctrl + F5** をすををクリックします。



12. しました。のコンソールがです。



Clang とのコンパイル

ClangのフロントエンドはGCCとがあるようにされているので、GCCを使ってコンパイルできるプログラムのほとんどは、ビルドスクリプトで`g++`を`clang++`でスワップするとコンパイルされます。

`-std=version`がされていない、`gnu11`がされます。

MSVCにれているWindowsユーザーは、`cl.exe`を`clang-cl.exe`とれえることができます。デフォルトでは、clangはインストールされているMSVCのバージョンとのをようとします。

Visual Studioをしてコンパイルするは、プロジェクトプロパティのPlatform toolsetをしてclang-clをできます。

どちらのも、clangはフロントエンドでのみがありますが、バイナリオブジェクトファイルもしようとします。clang-clのユーザーは、MSVCとのがまだではないことにするがあります。

clangやclang-clをうには、のLinuxディストリビューションやIDEにバンドルされているものMacのXCodeなどでデフォルトインストールをうことができます。このコンパイラのものバージョンまたはこれがインストールされていないプラットフォームでは、のダウンロードページからダウンロードできます。

CMakeを使ってコードをビルドするは、`cc`や`cxx`をのようにしてコンパイラをりえることができます

```
mkdir build
cd build
CC=clang CXX=clang++ cmake ..
cmake --build .
```

Cmakeのもしてください。

オンラインコンパイラ

さまざまなWebサイトがC++コンパイラへのオンラインアクセスをしています。オンラインコンパイラのセットはサイトによって異なりますが、はのことがです。

- ブラウザのコードをWebフォームにりけます。
- いくつかのコンパイラオプションをし、コードをコンパイルします。
- コンパイラおよび/またはプログラムをする。

コンパイラをしてサーバーでのコードをできるようにするオンラインコンパイラのWebサイトのは、はにされていますが、はリモートのコードのはとみなされます。

オンラインコンパイラは、のにちます。

- C++コンパイラがないマシンスマートフォン、タブレットなどからさなコードスニペットをします。
- なるコンパイラでコードをコンパイルし、コンパイルしたコンパイラになくじでコードをするようにしてください。
- C++のをび、える。
- のC++コンパイラがローカルマシンでできない、のC++いC++ 14とC++ 17をんでください。
- のコンパイラのきなセットとしてコンパイラにバグをします。コンパイラのバグがのバージョンでされたかどうかをしてください。これはあなたのマシンではできません。
- オンラインジャッジのをする。

オンラインコンパイラがされるべきでないもの

- C++をしてフルのアプリケーションさなアプリケーションでもをする。、オンラインコンパイラはサードパーティのライブラリとのリンクやビルドアーティファクトのダウンロードをしていません。
- なをします。サーバーのコンピューティングリソースはられているため、ユーザーのプログラムはにされます。されたは、テストやにです。
- コンパイラサーバーまたはネットのサードパーティホストをする。

はのリソースにはしません。ウェブサイトはアルファベットにされます。

- <http://codepad.org/>コードによるオンラインコンパイラ。ソースコードのまたはエラーでコンパイルしたにコードをすることはあまりうまくしません。
- <http://coliru.stacked-crooked.com/>コマンドラインをするオンラインコンパイラ。のためにGCCとClangコンパイラのをします。
- <http://cpp.sh/> - C++ 14をサポートするオンラインコンパイラ。コンパイラのコマンドラインをすることはできませんが、いくつかのオプションはGUIコントロールでできます。
- <https://gcc.godbolt.org/> - コンパイラのバージョン、アーキテクチャ、およびアセンブリのいリストをします。あなたのコードがなるコンパイラによってコンパイルされているものをするがあるときににです。ARM、ARMv8ARM64として、Atmel AVR、MIPS、MIPS64、

MSP430、PowerPCでなこれらのコンパイラは、GCC、Clang、MSVC_{CL}、Intelコンパイラ_{icc}、ELLCC、Zapccがあります。、x86、およびx64アーキテクチャーです。コンパイラのコマンドラインをすることができます。

- <https://ideone.com/> - コードスニペットのをするために、ネットでくされています。のためにGCCとClangのをしますが、コンパイラのコマンドラインをすることはできません。
- <http://melpon.org/wandbox> - のClangとGNU / GCCコンパイラのバージョンをサポートしています。
- <http://onlinegdb.com/> - エディタ、コンパイラgcc、およびデバツガgdbをむのIDEです。
- <http://rextester.com/> - Boostライブラリをできるように、CとC++のためのClang、GCC、およびVisual Studioコンパイラのものコンパイラとにをします。
- http://tutorialspoint.com/compile_cpp11_online.php - GCCをえたフルのUNIXシェルと、いやすいプロジェクトエクスプローラです。
- <http://webcompiler.cloudapp.net/> - RiSE4funのとしてMicrosoftがするオンラインVisual Studio 2015コンパイラ。

C++コンパイルプロセス

C++プログラムをするときは、プログラムをコンパイルしてからしてください。コンパイルとは、C、C++などのがめるでかれたプログラムを、ができるマシンコードにするプロセスです。たとえば、prog.cppというのC++ソースコードファイルがあり、コンパイルコマンドをすると、

```
g++ -Wall -ansi -o prog prog.cpp
```

ソースファイルからファイルをするには、に4つのがあります。

1. C++プリプロセッサは、C++ソースコードファイルを、ヘッダ#include、マクロ#define、そののプリプロセッサディレクティブをします。
2. C++プリプロセッサによってされたC++ソースコードファイルは、プラットフォームのアセンブリにコンパイルされます。
3. コンパイラによってされたアセンブラコードは、プラットフォームのオブジェクトコードにアセンブルされます。
4. アセンブラでされたオブジェクトコードファイルはリンクされています
ライブラリまたはファイルをするためにされるライブラリのオブジェクトコードファイルをします。

プリプロセッサは#includeや#defineのようなプリプロセッサをいます。これはC++のにはです。そのため、してするがあります。

#includeディレクティブをそれぞれのファイルのはのみにきえ、マクロ#defineをきえ、#ifにじてテキストのなるをすることで、に1つのC++ソースファイルでします。 #ifdefおよび#ifndefディレクティブ。

プリプロセッサは、トークンのストリームでします。マクロは、トークンをのトークンにきえることでされます##は、2つのトークンをさせることができます。

この、プリプロセッサは、ののトークンのストリームであるのをします。また、がどこからたのかをコンパイラにえるなマーカーをして、それらをとってかりやすいエラーメッセージをすることができます。

#ifと#errorディレクティブをにして、このでエラーをすることができます。

のコンパイラフラグをすることで、でプロセスをすることができます。

```
g++ -E prog.cpp
```

コンパイル・ステップは、プリプロセッサのにしてされます。コンパイラはなC++ソースコードをしプリプロセッサディレクティブなしで、アセンブリコードにします。に、そのコードをコードにアセンブルして、のバイナリファイルをらかのELF、COFF、a.out、...でする、となるバックエンドツールチェーンのアセンブラをびします。このオブジェクトファイルには、にされているシンボルのコンパイルみコードバイナリがされています。オブジェクトファイルのシンボルはでされます。

オブジェクトファイルは、されていないシンボルをできます。これは、をしているで、をしていないです。コンパイラはこれにをつけず、ソースコードがであるり、オブジェクトファイルをうまくします。

コンパイラは、このでコンパイルをします。これにより、ソースコードファイルをにコンパイルできるため、にです。これがするは、1つのファイルのみをした、すべてをコンパイルするがないことです。

されたオブジェクトファイルは、でするために、ライブラリとばれるなアーカイブにくことができます。

このでは、エラーやしたオーバーロードエラーなどの「の」コンパイラエラーがされます。

コンパイル・ステップのでプロセスをするために、-Sオプションをできます。

```
g++ -Wall -ansi -S prog.cpp
```

みて

アセンブラはオブジェクトコードをします。UNIXシステムでは、オブジェクトコードファイルをすために、が.oMSDOSでは.OBJのファイルがされることがあります。このフェーズでは、アセンブラはこれらのオブジェクトファイルをアセンブリコードからマシンレベルのにし、されたファイルはなオブジェクトコードです。したがって、コンパイルフェーズではリロケータブルオブジェクトプログラムがされ、このプログラムはののでしてコンパイルするはありません。

アセンブリステップにプロセスをするには、-cオプションをできます。

```
g++ -Wall -ansi -c prog.cpp
```

リンクする

リンカは、アセンブラがしたオブジェクトファイルからなコンパイルをするものです。これは、またはライブラリがいても、のライブラリとほとんどじではありませんまたはファイルのいずれかです。

のシンボルへのをしいアドレスにきえることによって、すべてのオブジェクトファイルをリンクします。これらのシンボルのそれぞれは、のオブジェクトファイルまたはライブラリですることができます。それらがライブラリのライブラリにされている、それらについてリンカにえるがあります。

このでもなエラーはがしているか、がしています。は、がしないすなわち、きまれているか、またはそれらがするオブジェクトファイルまたはライブラリがリンカーにえられていないことをします。はらかです。じシンボルが2つのなるオブジェクトファイルまたはライブラリでされています。

Code :: BlocksGraphical interfaceでコンパイルする

1. Code :: Blocksをダウンロードしてインストールして**ください**。Windowsのは、にmingwがまれているファイルをするようにしてください。のファイルはコンパイラをインストールしません。
2. Code :: Blocksをき、"Create a new project"をクリックしてください

Start here - Code::Blocks 16.01

File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plug



Management x

Projects Symbols

Workspace

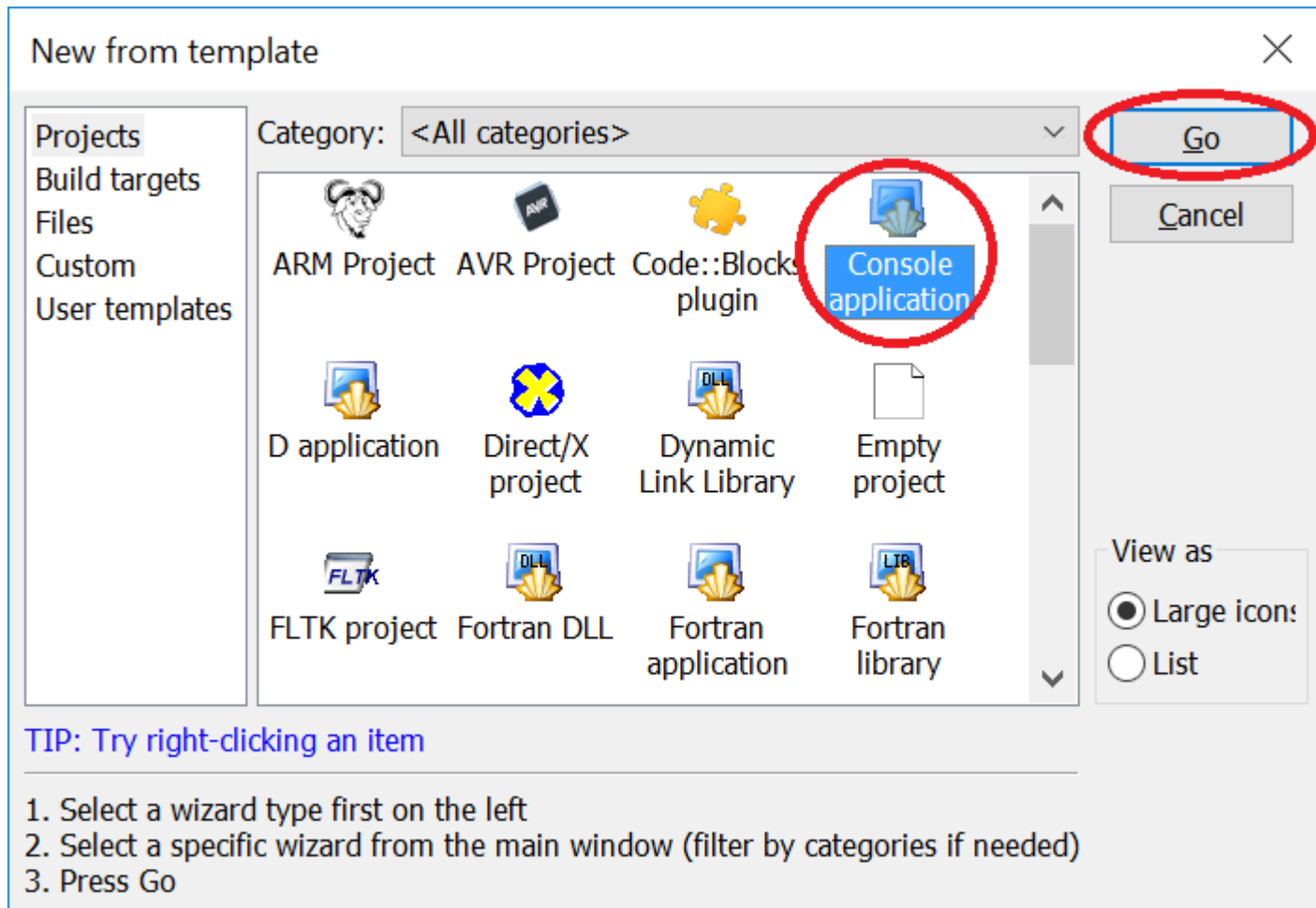
Start here x

Logs & others

Code::Blocks x Search results x Cccc x Build

Start here

3. 「コンソールアプリケーション」をし、「」をクリックします。



4. "へ"をクリックして "へ"をクリックし、プロジェクトのをしてするフォルダをし、 "へ"をクリックして ""をクリックします。

クリックして ""をクリックします。

5. これでコードをしてコンパイルできます。「Hello world」をするデフォルトコード。コンソールにはにそこにあります。プログラムをコンパイルおよび/またはするには、ツールバーの3つのコンパイル/ボタンの1つをします。

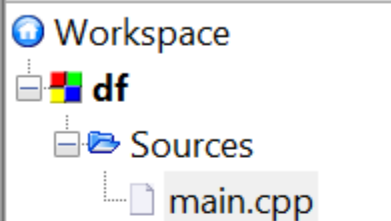
main.cpp [df] - Code::Blocks 16.01

File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plug



Management

Projects Symbols



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

せずにコンパイルするには、、コンパイルせずにするには、 コンパイルしてからするには、。

するには、。

デフォルトの "Hello world" をコンパイルしてします。コードはのをします。



```
Hello world!  
  
Process returned 0 (0x0)    execution time : 0.047 s  
Press any key to continue.  
_
```

オンラインでコンパイルとビルドをむ <https://riptutorial.com/ja/cplusplus/topic/4708/コンパイルとビルド>

57: サイドバイサイド C++ と C++ 11 と C++ 14 と C++ 17 でされたな C++ のの

Examples

コンテナをループする

C++ では、のようにインデックスをしてシーケンスコンテナをループすることができます。

```
for(size_t i = 0; i < c.size(); ++i) c[i] = 0;
```

シンプルではあるが、そのようなは、ったやっただけのようなののりがあります。

```
for(size_t i = 0; i <= c.size(); ++j) c[i] = 0;
      ^~~~~~^
```

のをつイテレータをして、すべてのコンテナにしてループをすることもできます。

```
for(iterator it = c.begin(); it != c.end(); ++it) (*it) = 0;
```

C++ 11 では、ループベースのベースおよび `auto` キーワードがされ、コードはのようになりました。

```
for(auto& x : c) x = 0;
```

ここではのパラメータはコンテナ `c` とのをする `x` です。これにより、にされたセマンティクスエラーがされます。

C++ 11 によれば、なはのものとです

```
for(auto begin = c.begin(), end = c.end(); begin != end; ++begin)
{
    // ...
}
```

このようなでは、 `auto begin = c.begin(), end = c.end(); begin` と `end` をじにしますが、 `end` はももされません。したがって、ベースの `for` ループは、ペアイテレータ/イテレータでされたコンテナにしてのみします。C++ 17 では、をのようにしてこのをしています。

```
auto begin = c.begin();
auto end = c.end();
for(; begin != end; ++begin)
{
    // ...
}
```

```
}
```

ここで、`begin`と`end`は、のためにできるり、なるであることがされます。これは、よりくのコンテナ、例えば、/センチネルのペアによってされるコンテナをループすることをにする。

オンラインでサイドバイサイドC++とC++ 11とC++ 14とC++ 17でされたなC++ののをむ
<https://riptutorial.com/ja/cplusplus/topic/7134/サイドバイサイドcplusplusとcplusplus-11とcplusplus-14とcplusplus-17でされたなcplusplusのの>

58: シングルトンデザインパターン

シングルトンは、クラスがインスタンスを1つだけし、それへのアクセスのグローバルポイントをするようにされています。1つのインスタンスまたはなグローバル・アクセス・ポイントのみをとし、のアクセス・ポイントをとしないは、シングルトンにりえるにのオプションをしてください。

グローバルは、コードをするのをよりにするがあります。たとえば、びしの1つがシングルトンからけったデータにしていなは、にシングルトンのデータをにえるものをするがあります。

シングルトンはまたをします。これはされたコードの2つのコンポーネントをするためにされるで、コンポーネントはののをらします。

シングルトンはにります。クラスにグローバル・アクセス・ポイントがある、すべてのスレッドはそれにアクセスでき、デッドロックやにるがあります。

に、なは、ったにされた、パフォーマンスのをきこすがあります。をすることで、サブクラスをなどのシングルトンのいをするともできます。

Robert Nystromの ゲームプログラミングパターン

Examples

レイジー

このはここのQ & Aセクションからされました <http://stackoverflow.com/a/1008289/3807729>

されたシングルトンでされたのなについては、このをしてください。

でもにシングルトンのサンプルをc++でできますか

なけはされ、しくシングルトンをしました。

```
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
};

```

シングルトンをするについては、このをしてくださいではありません
[シングルトンどのようにすべきか](#)

のとは、の2つのをしてください。

[の](#)

[C++のの](#)

ライフタイムをするこのをごください

[C++ののはですか](#)

シングルトンにするいくつかのスレッドのをしているこのをしてください。

[GetInstanceメソッドのとしてされたシングルトンインスタンス](#)

C++でダブルチェックロックがしないをしているこのをしてください。

[C++プログラマがべきののるいはですか](#)

サブクラス

```

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;
}

```

このでは、なコンパイラスイッチは、`API`クラスをなサブクラスにバインドし`API`。このようにして、プラットフォームのコードにすることなく`API`にアクセスできます。

スレッドセーフシントン

C++ 11

C++ 11では、スコープオブジェクトのがしてわれることがされています。これは、[レイジー](#)をしてスレッドセーフなシングルトンをするためにできます。

```

class Foo
{
public:
    static Foo& instance()
    {
        static Foo inst;
        return inst;
    }
private:
    Foo() {}
    Foo(const Foo&) = delete;
    Foo& operator =(const Foo&) = delete;
};

```

なシングルトン。

あなたはシングルトンがそれをとしなくなっシングルトンをするすべてのオブジェクトまでされないことをできるようにするがあり、のオブジェクトとのがあります。

この、`std::shared_ptr`は、プログラムのでデストラクタがびされているときでも、すべてのユーザにしてシングルトンをかしけるためにできます

```

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() {}
};
```

これはこのQAセクションのえとしてされます。

オンラインでシングルトンデザインパターンをむ <https://riptutorial.com/ja/cplusplus/topic/2713/シングルトンデザインパターン>

59: スコープ

Examples

なブロックスコープ

ブロック{ ... }ののスコープは、にまり、ブロックのでします。ネストされたブロックがある、ブロックはブロックでされたのスコープをすことができます。

```
{
    int x = 100;
    //   ^
    //   Scope of `x` begins here
    //
} // <- Scope of `x` ends here
```

ネストされたブロックがブロックでされた、クラスのにあるじをつされたがたにされ、のがされます。

```
{
    int x = 100;

    {
        int x = 200;

        std::cout << x; // <- Output is 200
    }

    std::cout << x; // <- Output is 100
}
```

グローバル

なるソースファイルからアクセスなののインスタンスをするためには、キーワード`extern`してグローバルスコープでをすることができます。このキーワードは、コードのどこかにこののがあるので、どこでもでき、すべてのきみ/みしがメモリの1つのでわれることをコンパイラにします。

```
// 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;
}
```

```
The radius is: 123
```

オンラインでスコープをむ <https://riptutorial.com/ja/cplusplus/topic/3453/スコープ>

60: ストリームマニピュレータ

き

マニピュレータは、`operator >>`または`operator <<`をしてストリームとストリームをするのにつなヘルパーです。

それらは`#include <iomanip>`によってすべてめることができます。

マニピュレータはのことができます。えは

1. `os.width(n); os << std::setw(n);`としくなり`os << std::setw(n);`
`is.width(n); is >> std::setw(n);`としくなり`is >> std::setw(n);`
2. `os.precision(n); os << std::setprecision(n);`としくなり`os << std::setprecision(n);`
`is.precision(n); is >> std::setprecision(n);`としくなり`is >> std::setprecision(n);`
3. `os.setfill(c); os << std::setfill(c);`としくなり`os << std::setfill(c);`
4. `str >> std::setbase(base);`または`str << std::setbase(base);`しい

```
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); os << std::flag;`としくなり`os << std::flag;`
`is.setf(std::ios_base::flag); is >> std::flag;`しくなり`is >> std::flag;`
`os.unsetf(std::ios_base::flag); os << std::no ## flag;`としくなり`os << std::no ## flag;`
`is.unsetf(std::ios_base::flag); is >> std::no ## flag;`としい`is >> std::no ## flag;`
-
のflag **S boolalpha**、**showbase**、**showpoint**、**showpos**、**skipws**、**uppercase**。

6. `std::ios_base::basefield`。
flag Sの **dec**、**hex**、**oct**

- `os.setf(std::ios_base::flag, std::ios_base::basefield); os << std::flag;`としくなり
`os << std::flag;`
`is.setf(std::ios_base::flag, std::ios_base::basefield); is >> std::flag;`しくなり
`is >> std::flag;`

1

- `str.unsetf(std::ios_base::flag, std::ios_base::basefield);`
`str.setf(std::ios_base::fmtflags(0), std::ios_base::basefield);`

2

7. `std::ios_base::adjustfield`。

flag **S**の **left**、**right**、**internal**

- `os.setf(std::ios_base::flag, std::ios_base::adjustfield); os << std::flag;`としくなり
`os << std::flag;`
`is.setf(std::ios_base::flag, std::ios_base::adjustfield); is >> std::flag;`しくなり
`is >> std::flag;`

1

- `str.unsetf(std::ios_base::flag, std::ios_base::adjustfield);`
`str.setf(std::ios_base::fmtflags(0), std::ios_base::adjustfield);`

2

1にされているするフィールドのフラグが`unsetf`によってにされている`unsetf`。

2 `flag`がセットされている。

8. `std::ios_base::floatfield`。

- `os.setf(std::ios_base::flag, std::ios_base::floatfield); os << std::flag;`としくなり
`os << std::flag;`
`is.setf(std::ios_base::flag, std::ios_base::floatfield); is >> std::flag;`しくなり
`is >> std::flag;`
flagのために **fixed**、**scientific**。
- `os.setf(std::ios_base::fmtflags(0), std::ios_base::floatfield); os << std::defaultfloat;`と
しくなり
`os << std::defaultfloat;`
`is.setf(std::ios_base::fmtflags(0), std::ios_base::floatfield); is >> std::defaultfloat;`と
しい
`is >> std::defaultfloat;`

9. `str.setf(std::ios_base::fmtflags(0), std::ios_base::flag); str.unsetf(std::ios_base::flag)`
とじ
`str.unsetf(std::ios_base::flag)`
flag **S**の **basefield**、**adjustfield**、**floatfield**

10. `os.setf(mask)`は、`os << setiosflags(mask);`とじ
`os << setiosflags(mask);`
`is.setf(mask)`はのもの
`is >> setiosflags(mask);`じ
`is >> setiosflags(mask);`

`os.unsetf(mask)` は `os << resetiosflags(mask); os.unsetf(mask)` としくなり `os << resetiosflags(mask);`
`is.unsetf(mask)` はのようになり `is >> resetiosflags(mask);`
ほぼすべての `std::ios_base::fmtflags mask` です。

Examples

ストリームマニピュレータ

`std::boolalpha` と `std::noboolalpha` - ブールのテキストとをりえます。

```
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` と `std::noshowbase` - ベースをすプレフィックスをするかどうかをします。

`std::dec` decimal、 `std::hex` hexadecimal、 `std::oct` octalは、のをするためにされます。

```
#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
```

デフォルトは `std::ios_base::noshowbase` と `std::ios_base::dec` です。

`std::istringstream` については、`<sstream>` ヘッダーをしてください。

`std::uppercase` と `std::nouppercase` - と16のにをするかどうかをします。ストリームにはしません。

```
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

```

デフォルトは `std::nouppercase` です。

`std::setw(n)` - のフィールドのをに `n` ます。

いくつかのがびされると、`width` プロパティ `n` は `0` にリセットされ `0` フルリストは [ここに](#) あります。

```

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"

```

デフォルトは `std::setw(0)` です。

`std::left`、`std::right` および `std::internal` - `std::ios_base::adjustfield` を `std::ios_base::left`、`std::ios_base::right` および `std::ios_base::internal` して、りつぶしのデフォルトをする `std::ios_base::internal` します。 `std::left` と `std::right` はのにされ、`std::internal` - 、およびの。ストリームにはしません。

```

#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

```

デフォルトは `std::left` です。

`std::fixed`、`std::scientific`、`std::hexfloat` [C ++ 11]および`std::defaultfloat` [C ++ 11] - のをします。

`std::fixed`は`std::ios_base::floatfield`を`std::ios_base::fixed`にします。

`std::scientific` - `std::ios_base::scientific`、

`std::hexfloat` - `std::ios_base::fixed` | `std::ios_base::scientific`および

`std::defaultfloat` - `std::ios_base::fmtflags(0)` ます。

`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::stringstream 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
```

デフォルトは `std::ios_base::fmtflags(0)` です。

いくつかのコンパイラにはバグがあります

```
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` と `std::noshowpoint` - がににまれるかどうかをします。ストリームにはしません。

```
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
```

デフォルトは `std::showpoint` です。

`std::showpos` と `std::noshowpos` - のに+のをします。ストリームにはしません。

```
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
```

`std::noshowpos` はデフォルトです。

`std::unitbuf`、`std::nounitbuf` - すべてののにストリームをフラッシュするようにします。ストリームにはしません。`std::unitbuf` はフラッシュをきこします。

`std::setbase(base)` - ストリームのベースをします。

`std::setbase(8)` は `std::ios_base::basefield` を `std::ios_base::oct` にすることとしく、

`std::setbase(16)` - `std::ios_base::hex`、
`std::setbase(10)` - `std::ios_base::dec` ます。

`base`、のある 8、10 は 16 に `std::ios_base::basefield` するためにされて `std::ios_base::fmtflags(0)` これは、とをします。

デフォルトの `std::ios_base::basefield` は `std::ios_base::dec`、デフォルトでは `std::setbase(10)` です。

`std::setprecision(n)` - をします。

```
#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
```

デフォルトは `std::setprecision(6)` です。

`std::setiosflags(mask)` と `std::resetiosflags(mask)` - `std::ios_base::fmtflags` の `mask` でされたフラグをセットしてクリアする。

```
#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`と`std::noskipws` - フォーマットされたによるののスキップをします。ストリームにはしません。

```
#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
```

デフォルトは`std::ios_base::skipws`です。

`std::quoted(s[, delim[, escape]])` [C ++ 14] - スペースをめんだきのをまたはします。

`s` - またはする

`delim` - デリミタとしてする"をします。

`escape` - エスケープとしてする\デフォルトでは。

```
#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]

```

はのリンクをしてください。

ストリームマニピュレータ

`std::ends` - ストリームにヌル、`\0`、をします。よりには、このマニピュレータのはのようになります

```

template <class charT, class traits>
std::basic_ostream<charT, traits>& ends(std::basic_ostream<charT, traits>& os);

```

このマニピュレータは、でされているときに `os.put(charT())` びしてをします

```
os << std::ends;
```

`std::endl` と `std::flush` ののフラッシュストリーム `out` びすことによって、 `out.flush()` にがされます。しかし、 `std::endl` は、フラッシングのに、`\n` に、`\n`、します。

```

std::cout << "First line." << std::endl << "Second line. " << std::flush
          << "Still second line.";
// Output: First line.
// Second line. Still second line.

```

`std::setfill(c)` - りつぶしを `c` し `c`。 `std::setw` よくされます。

```

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]です。 `out << std::put_money(mon, intl)` では、 `std::money_put` されているロケールの `std::money_put` ファセットでされている `mon long double` または `std::basic_string` をそのに `std::money_put` します `out`。 `intl` が `true` はをし、それのはをします。

```

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 py6 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] - されたフォーマット `fmt` につて、`t` をフォーマットして `std::tm` にします。

`tmb` - `localtime()` または `gmtime()` からしたカレンダー `const std::tm*` へのポインタ。
`fmt` - ヌルであるへのポインタ `const CharT*` のをします。

```

#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

```

はこのリンクをしてください。

ストリームマニピュレータ

`std::ws` - ストリームののをいます。これは `std::skipws` ます。

```

#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]。 `in >> std::get_money(mon, intl)` でされ、などのを

`std::money_get` にきまロケールの `in`、 `びのをする` `mon` の `long double` または `std::basic_string` マニピュレータは、 `intl` が `true` はなをし、それのはオプションのがです。

```
#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] - されたフォーマット `fmt tmb` にされている/のをします。

`tmb` - がされる `const std::tm*` オブジェクトへのなポインタ。

`fmt` - ヌルでするへのポインタ `const CharT*` フォーマットをします。

```
#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
```

はこのリンクをしてください。

オンラインでストリームマニピュレータをむ <https://riptutorial.com/ja/cplusplus/topic/10699/ストリームマニピュレータ>

61: スマートポインタ

- `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, ...); // C++ 14`
- `std::weak_ptr<ClassType> variableName (new ClassType(arg1, arg2, ...));`

C++はメモリではありません。にりてられたメモリ `new` オブジェクトでされたオブジェクトは、にりてされていない `delete`、"リーク"されます。そのオブジェクトへのポインタをするに、にりてられたメモリがされることをするのはプログラマのです。

スマートポインタをして、にりてられたメモリのをにすることができますつまり、のポインタがになったときにされます。

スマートポインタは、ほとんどの、「の」ポインタよりもされます。にりてられたメモリのセマンティクスを、オブジェクトがまたはにされることをしているかどうか、そのですることによってにします。

スマートポインタをできるようにするには、`#include <memory>`をします。

Examples

の `std::shared_ptr`

クラステンプレート `std::shared_ptr` は、オブジェクトのをのポインタとできるポインタをします。これは、をす `std::unique_ptr` とはです。

は、オブジェクトをすポインタのがそのオブジェクトのにされるカウントとばれるによってされます。の `std::shared_ptr` インスタンスのまたはりてのいずれかによってこのがゼロになると、オブジェクトはにされます。

```
// Creation: 'firstShared' is a shared pointer for a new instance of 'Foo'
std::shared_ptr<Foo> firstShared = std::make_shared<Foo>(*args*);
```

じオブジェクトをするのスマートポインタをするには、のポインタのであるの `shared_ptr` をするがあります。それをうには2つのがあります

```
std::shared_ptr<Foo> secondShared(firstShared); // 1st way: Copy constructing
std::shared_ptr<Foo> secondShared;
secondShared = firstShared; // 2nd way: Assigning
```

のいずれかのでは `secondShared` ののインスタンスのをし、ポインタ `Foo` と `firstShared`。

スマートポインタはポインタとじようにします。つまり、`*`をしてすることができます。regular

->もにします

```
secondShared->test(); // Calls Foo::test()
```

にエイリアスされた `shared_ptr` がスコープかられると、`Foo` インスタンスのデストラクタがびされます。

セマンティクスのなデータをりてあるがある、`shared_ptr` をすると `bad_alloc` がスローされることがあります。コンストラクタにのポインタがされた、それはしされているオブジェクトをするものとみなされ、がスローされたにはディレクタをびします。つまり、`shared_ptr<T>(new T(args))` は `shared_ptr<T>` りてがしたに `T` オブジェクトをリークしません。ただし、`make_shared<T>(args)` または `allocate_shared<T>(alloc, args)` をすることをおめします。これにより、メモリりてがされます。

`shared_ptr` をして [] をりて

C++ 11 C++ 17

ながら、`make_shared<>` をしてをりてるなはありません。

`new` と `std::default_delete` をして `shared_ptr<>` をすることはです。

たとえば、10ののをりてるには、のようにコードをくことができます。

```
shared_ptr<int> sh(new int[10], std::default_delete<int[]>());
```

りてられたメモリが `delete[]` をしてしくクリーンアップされるように、`std::default_delete` することはです。

コンパイルにサイズをっていれば、のようにすることができます

```
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>{}(); }
```

`make_shared_array<int[10]>` は、デフォルトでされた10のintをす `shared_ptr<int>` します。

C++ 17

C++ 17では、`shared_ptr` はをにサポートしていました。 `array-deleter`をにするはなく、 `[]` インデックスをしてポインタをすることができます。

```
std::shared_ptr<int[]> sh(new int[10]);
sh[0] = 42;
```

ポインタは、がするオブジェクトのサブオブジェクトをすることができます。

```
struct Foo { int x; };
std::shared_ptr<Foo> p1 = std::make_shared<Foo>();
std::shared_ptr<int> p2(p1, &p1->x);
```

`p2`と`p1`が`Foo`のオブジェクトをしますが、`p2`はその`int`メンバ`x`しています。つまり、`p1`がになりりてされたりすると、の`Foo`オブジェクトはききになり、`p2`がぶらがないことがされます。

`shared_ptr`は、エイリアスコンストラクタでされた`shared_ptr`とそれのすべての`shared_ptr`のみをします。じ`Foo`インスタンスへのでされたのすべての`shared_ptr`をむのポインタについてはわかりません

```
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!!
```

`shared_ptr`の

デフォルトでは、`shared_ptr`はカウントをインクリメントし、をしません。ただし、`std::move`をしてをすることができます。

```
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
```

をつ`std::weak_ptr`

`std::weak_ptr`インスタンスは、`std::shared_ptr`インスタンスがするオブジェクトをすることができますが、なになることができます。これは、ウィークポインタがオブジェクトのカウントをしないため、オブジェクトのポインタのすべてがりてまたはされた、オブジェクトのをしないことをします。

のでは、 `std::weak_ptr` インスタンスがされているため、ツリーオブジェクトのはされていません。

```
#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();
}
```

ノードがルートノードのノードにされると、その `std::weak_ptr` メンバーの `parent` ノードがルートノードにされます。メンバーの `parent` は、ルートポインタのカウントがインクリメントされないように、ポインタとはにウィークポインタとしてされます。ルートノードが `main()` でリセットされると、ルートはされます。ノードへのりの `std::shared_ptr` のみがルートのコレクションの `children` にまわっていたため、すべてのノードもでされます。

ブロックののにより、 `shared_ptr` カウンタと `weak_ptr` カウンタのがゼロにするまで、 `shared_ptr` りてメモリはされないことがあります。

```
#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)
}
```

`std::weak_ptr` をじてされるオブジェクトきて、データ・アクセスをしていない `std::weak_ptr` できま

せん。わりに、されるオブジェクトへの`std::shared_ptr`をみる`lock()`メンバをします

```
#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());
    }
}
```

の`std::unique_ptr`

C++ 11

`std::unique_ptr`は、にされたオブジェクトのをするクラステンプレートです。 `std::shared_ptr`とはなり、オブジェクトはいつでも`std::unique_ptr` 1つのインスタンスによってのみされ、

```
// Creates a dynamic int with value of 20 owned by a unique pointer
std::unique_ptr<int> ptr = std::make_unique<int>(20);
```

`std::unique_ptr`はC++ 11、そして`std::make_unique`はC++ 14です。

`ptr`だけがにりてられた`int`へのポインタをします。オブジェクトをするのポインタがになると、されているオブジェクトがされます。つまり、オブジェクトがクラスで、そのオブジェクトのメモリがされた、そのデストラクターがびされます。

で`std::unique_ptr`と`std::make_unique`をするには、のをします。

```
// 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);
```

これらはオーバーロードされているので、ポインタとに `std::unique_ptr` アクセスできます。

スマートポインタのものをポインタに `std::move`、`std::move` をします。これにより、のスマートポインタが `nullptr` をすようになります。

```
// 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)
```

`unique_ptr` をパラメータとしてにす

```
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))
```

から `unique_ptr` をします。これは、りののをにえるので、ファクトリをくのにましいC++11のです。びしは `unique_ptr` をし、そのをいます。

```
std::unique_ptr<int> foo()
{
    std::unique_ptr<int> ptr = std::make_unique<int>(59);
    return ptr;
}

std::unique_ptr<int> ptr = foo();
```

これとする

```
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

クラステンプレート `make_unique` は、C++ 14でされています。でC++ 11コードにでするのはです

```
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

ダムスマートポインタ `std::auto_ptr` とはなり、`unique_ptr` はベクトルりて `std::vector` ではなくでインスタンスすることもできます。これまでののは、スカラーりてのでした。10ののためのにりてられたをするように、えは、のようにし `int[]` テンプレートタイプだけではないように `int`。

```
std::unique_ptr<int[]> arr_ptr = std::make_unique<int[]>(10);
```

どちらをできるか

```
auto arr_ptr = std::make_unique<int[]>(10);
```

さて、`arr_ptr` をのようします

```
arr_ptr[2] = 10; // Modify third element
```

りてについてするはありません。このテンプレートバージョンは、コンストラクタとデストラクタをにびします。ベクトルされたバージョンの `unique_ptr` または `vector` をすることはなです。

C++ 11よりのバージョンでは、`std::auto_ptr` がでした。`unique_ptr` とはなり、`auto_ptr` をコピーすることができます。このとき、ソース `ptr` はまれているポインタのをい、ターゲットはそれをけります。

カスタム・リターをしてCインターフェースへのラッパーをする

[SDL2](#)などのくのCインタフェースにはのがあります。つまり、スマートポインタをすることはできません。

```
std::unique_ptr<SDL_Surface> a; // won't work, UNSAFE!
```

わりに、のデリゲーターをするがあります。ここのでは、[SDL_Surface](#) をしています。これは [SDL_FreeSurface\(\)](#) をしてするがありますが、のくのCインタフェースにもです。

`deleter` はポインタでびしでなければならないため、えはなポインタにすることができます

```
std::unique_ptr<SDL_Surface, void(*) (SDL_Surface*)> a(pointer, SDL_FreeSurface);
```

のびしなオブジェクトもしますたとえば、 `operator()` つクラスなど `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
```

これにより、でゼロのオーバーヘッド `unique_ptr` をするのメモリがされるだけでなく、もらえます。

Deleterは `unique_ptr` ののであり、は `unique_ptr` をしてのカスタムdeleterのサイズのをけることができます。したがって、 `std::unique_ptr<SDL_Surface, SurfaceDeleter>` と `std::unique_ptr<SDL_Surface, void(*) (SDL_Surface*)>` はじをじようにしますが、のはポインタのサイズだけです。のは2つのポインタをするがあります `SDL_Surface*` とポインタフリーのカスタム・リターダをするは、のをラップすることをおめします。

カウントがなは、 `unique_ptr` わりに `shared_ptr` をできます。 `shared_ptr` にDeleterがされています。これにより、APIでなDeleterのがされます。 `unique_ptr` に `shared_ptr` をすることのは、deleterをするためのメモリコストがくなり、カウントをするためのパフォーマンスコストがかかることです。

```
// 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

template auto、カスタムレーターをにラップすることができます

```
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>>;
```

のはにのようになります。

```
unique_ptr_deleter<SDL_Surface, SDL_FreeSurface> c(pointer);
```

ここで`auto`のは`void` えば`SDL_FreeSurface` をしてもしなくても `fclose`、すべてのきをすることです。

セマンティクスのないの`auto_ptr`

C++ 11

`std::auto_ptr`はC++ 11ではされ、C++ではされます17。C++ 03のバージョンをすることをなくされ、をつけたいにのみ、これをしてください。 `std::auto_ptr`をきえるために、`std::move`とみわけて`unique_ptr`にすることをおめします。

たちが`std::unique_ptr`をつに、セマンティクスをさせるに、`std::auto_ptr`をつていました。`std::auto_ptr`はのをしますが、コピーにをします。

すべてのスマートポインタとに、`std::auto_ptr`にリソースをクリーンアップします [RAII](#)を。

```
{
    std::auto_ptr<int> p(new int(42));
    std::cout << *p;
} // p is deleted here, no memory leaked
```

は1だけです

```
std::auto_ptr<X> px = ...;
std::auto_ptr<X> py = px;
// px is now empty
```

これにより、`std::auto_ptr`をしてをかつにつことができます。

```
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
```

のは"コピー"コンストラクタでした。 `auto_ptr`のコピーコンストラクタとコピーは、オペランドを`const`でりしてできるようにします。はのとおりです。

```
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 ... */
};

```

これはコピーセマンティクスをします。コピーセマンティクスは、オブジェクトをコピーすると2つのバージョンをすがあります。どのようなコピーなタイプの`T`、はのようにくことができます

```

T a = ...;
T b(a);
assert(b == a);

```

しかし、`auto_ptr`のはそうではありません。その、`auto_ptr`をコンテナにれることはではありません。

これをして**shared_ptr**をする

`enable_shared_from_this`は、`this`にしてな**shared_ptr**インスタンスをできるようにします。

クラステンプレート**enable_shared_from_this**からクラスをさ**shared_from_this**ことで、**shared_ptr**インスタンスを**this**す**shared_from_this**メソッドをします。

オブジェクトはに**shared_ptr**としてするがあることにしてください。

```

#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

```

2コンストラクタで**enable_shared_from_this**をびすことはできません。

```
#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();
    ...
}
```

`shared_ptr`がしていないオブジェクト `shared_from_this()` ローカルオブジェクトやグローバルオブジェクトなど `shared_from_this()` で `shared_from_this()` をすると、そのはです。C++ 17、わりに `std::bad_alloc` スローされます。

コンストラクタからの `shared_from_this()` は、 `shared_ptr` がしていないオブジェクトでのとじです。これは、コンストラクタがったに `shared_ptr` がオブジェクトをしているためです。

キャスト `std::shared_ptr` ポインタ

することはできません `static_cast`、 `const_cast`、 `dynamic_cast` し、 `reinterpret_cast` `std::shared_ptr` としてされるポインタとポインタののをするために。わりに、 `std::static_pointer_cast`、 `std::const_pointer_cast`、 `std::dynamic_pointer_cast` および `std::reinterpret_pointer_cast` `std::const_pointer_cast` するがあります。

```
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));
```

`std::reinterpret_pointer_cast` [reinterpret_pointer_cast](#) はC++ 11およびC++ 14ではできません [.N3920](#) によってのみされ、 [20142](#) に Library Fundamentals TS にされたためです。ただし、のよう
にできます。

```
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())); }
```

スマートポインタのvalue_ptr

value_ptrは、のようにするスマートポインタです。コピーすると、そのがコピーされます。され
ると、そのがされます。

```
// 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)...) };
}

```

このvalue_ptrは、それをempty_ptr_tする、またはそこからするにのみです。それはそれがunique_ptrであるというをするので、explicit operator bool() constがそれにする。get()はリファレンスをすようにされていますほとんどではないため、get_pointer()はわりにポインタをします。

このスマートポインタは、のをめているpImplにはですが、ファイルのでpImplのをしたくないもあ

ります。

デフォルトではないCopierでは、したインスタンスをしてにするを持っているクラスもできます。

オンラインでスマートポインタをむ <https://riptutorial.com/ja/cplusplus/topic/509/スマートポインタ>

62: スレッディング

-
- スレッドスレッド&&その
- なスレッド&& func、 Args && ... args

パラメーター

パラメータ	
other	other をきぎ、 other はもうスレッドをしていない
func	のスレッドでびす
args	func

いくつかのメモ

- 2つの `std::thread` オブジェクトは、じスレッドをすことはできません。
- `std::thread` オブジェクトは、スレッドをしていない、 `join` びしなどになることがあります。

Examples

スレッド

スレッドをすると、スレッドがするまでされます。

くの、をしたいので、スレッドがするのをつがありますおそらく、スレッドがすでにしているがあります。

```
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 sepearte thread
std::cout << n << '\n';
```

スレッドを `detach` 、スレッドをにさせることもできます。

```
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
```

スレッドへののけし

`std::thread`はそれらをコピー/するので、または`const`をスレッドにすることはできません。わりに、`std::reference_wrapper`し`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
```

std :: threadの

C++では、スレッドは`std :: thread`クラスをしてされます。スレッドとはのフローです。のタスクをにしているにヘルパーに1つのタスクをさせることにしています。スレッドのすべてのコードがされると、します。スレッドをするときには、するものをすがあります。あなたがスレッドにすることができいくつかの

- フリー
- メンバー
- Functorオブジェクト
- ラムダ

フリーの - のスレッドでをする

```
#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;
}

```

メンバーの - のスレッドでメンバーをする

```

#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;
}

```

Functorオブジェクトの

```

#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;
}

```

ラムダの

```

#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;
}

```

のスレッドでの

`std::this_thread` は、`std` 名前空間からのスレッドでいことをするをつ `std::this_thread` namespace です。

<code>get_id</code>	スレッドのIDをします。
<code>sleep_for</code>	されたただける
<code>sleep_until</code>	のまです
<code>yield</code>	のスレッドをスケジュールし、のスレッドにをえる

`std::this_thread::get_id` をつてのスレッドIDをする

```
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
```

`std::this_thread::sleep_for` **をって3** `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";
```

3まで `std::this_thread::sleep_until` **をして** `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";
```

のスレッドに `std::this_thread::yield` **をってさ** `std::this_thread::yield`

```
void foo(int a)
{
    for (int i = 0; i < a; ++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();
```

std :: threadのわりに**std :: async**をう

`std::async`はスレッドをすることもできます。 `std::thread`とすると、あまりではありませんが、

にをしたいときにはいやすくなります。

にをびす

```
#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
}
```

とし

- `std::async`は、によってされるりをする `std::future` をします。その `future` がされると、スレッドがするまでって、コードをにシングルスレッドにします。これは、りがなときにとされます。

```
std::async(std::launch::async, square, 5);
//thread already completed at this point, because the returning future got destroyed
```

- `std::async`はポリシーなしですので、`std::async(square, 5);`コンパイルする。そうすると、システムはスレッドをするかどうかをします。システムがにできるよりもくのスレッドをしていないり、システムはスレッドをすることをしました。あいにく、はにそのようなでスレッドをしないことをするだけです。そのため、`std::launch::async`をしてそのをオーバーライドして、システムにスレッドをにさせるがあります。
- にしてください。

とにするの

スレッドがにしていることをする

`std::thread` デストラクタがびされると、`join()` または `detach()` いずれかのびしがわれていなければなりません。スレッドが `std::terminate` されたりデタッチされていない、デフォルトでは `std::terminate` がびされます。RAIIをすると、これはにのようになにできます。

```
class thread_joiner
{
public:

    thread_joiner(std::thread t)
        : t_(std::move(t))
    { }
```

```

~thread_joiner()
{
    if(t_.joinable()) {
        t_.join();
    }
}

private:

    std::thread t_;
}

```

これはのようにされます

```

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

```

これはもします。スレッドをにし、のをする`t()`われたがをスローした、`join()`はしてスレッドに
びされず、プロセスはします。

スレッドオブジェクトのりて

たちはのスレッドオブジェクトをし、でそれらにをりてることができます。

スレッドオブジェクトをのアクティブな`joinable`スレッドにりてると、スレッドがきえられるに
`std::terminate`がにびされます。

```

#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);
}

```


スレッドは、のプリミティブのでも、ミューテックスをしてできます。ライブラリにはいくつかのmutexがありますが、もなのはstd::mutexです。ミューテックスをロックするには、ロックをします。もなロックタイプはstd::lock_guardです

```
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
```

std::lock_guardすると、mutexはロックオブジェクトのライフタイムにわたってロックされます。リージョンをでロックするがあるは、わりにstd::unique_lockします。

```
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
}
```

よりくのスレッド

の

は、スレッドのをするためにミューテックスとみわけてされるプリミティブです。これをするためのまたはもなではありませんが、パターンにしているにとってはもなです。

1つはstd::unique_lock<std::mutex> std::condition_variableをちます。これにより、をするかどうかをするに、コードがをにべることができます。

は、std::thread、std::condition_variable、std::mutexなどをしてプロデューサ/コンシューマのスケッチをするです。

```
#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;
}

```

なスレッドプールをする

C++ 11スレッディングプリミティブはレベルです。それらは、スレッドプールのようなよりいレベルのをくのうことができます

C++ 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; })は、std::future<std::string>します tasks.queue([]{ return "hello world"s; })これは、タスクオブジェクトがするまでにhello worldます。

tasks.start(10) これは10のスレッドをします tasks.start(10) してスレッドをします。

`packaged_task<void()>`のは、のみのをするされた`std::function`がないためです。カスタムのものをくことはおそらく、`packaged_task<void()>`をうよりもくなるでしょう。

。

C++ 11

C++ 11では、`result_of_t<blah>`を`typename result_of<blah>::type`にきえます。

ミューテックスの。

スレッドローカルストレージ

スレッドローカルストレージは、`thread_local` **キーワード**をしてできます。`thread_local`でされたは、スレッドをつとわれます。

- プログラムのスレッドには、それぞれのスレッドローカルののコピーがあります。
- ローカルスコープのスレッドローカルは、のコントロールがそのをするときにされます。そのようは、`extern`されていないり、にです。
- またはクラスローカルスコープをつスレッドローカルは、スレッドののとしてされます。
- スレッドローカルは、スレッドにされます。
- クラスのメンバーはなにのみスレッドローカルにすることができます。そのため、スレッド、インスタンスのペアごとに1つのコピーではなく、スレッドごとにそののコピーが1つします。

```
void debug_counter() {  
    thread_local int count = 0;  
    Logger::log("This function has been called %d times by this thread", ++count);  
}
```

オンラインでスレッディングをむ <https://riptutorial.com/ja/cplusplus/topic/699/スレッディング>

63: スレッド

き

スレッドは、スレッドがする、いくつかのがになるがあります。このトピックでは、これらのをするためにライブラリによってされるさまざまなをつけることができます。

Examples

std :: shared_lock

shared_lockは、のロックとみわけてして、のリーダーとなライターをすることができます。

```
#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はするスレッドによってのをにします。タスクをできないは、std::system_errorをスローします。

std::once_flagとみわけてします。

```
#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;
}
```

なアクセスのためのオブジェクトロック。

オブジェクトにしてのをしているにオブジェクトをロックしたいことがよくあります。たとえば、イテレータをしてオブジェクトをべたりしたりするがあるです。のメンバをびすがあるときは、に々のメンバではなくオブジェクトをロックするがです。

えば

```
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
};
```

チェックサムをするとき、オブジェクトはみみのためにロックされ、オブジェクトからにみみたいのスレッドがにうことができます。

```
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;
}

```

オブジェクトをするとデータがされるため、ロックをしてをうがあります。

```

void clear(text_buffer& buf)
{
    auto lock = buf.lock_for_updates(); // exclusive lock
    std::fill(std::begin(buf), std::end(buf), '\0');
}

```

のロックをするは、にすべてのスレッドにしてじでロックをするようするがあります。

```

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));
}

```

これは**std :: deferred :: lock**と**std :: lock**をとてうのがです

std :: condition_variable_any、std :: cv_status

std::condition_variable、std::condition_variable_anyのは、どのタイプのBasicLockableでもします。

のりとしてのstd::cv_statusは、2つのなりコードがあります。

- std :: cv_status :: no_timeoutタイムアウトがなく、にされました
- std :: cv_status :: no_timeoutのタイムアウト

オンラインでスレッドをむ <https://riptutorial.com/ja/cplusplus/topic/9794/スレッド>

64: セマフォ

き

セマフォはC++ではできませんが、mutexとをってにできます。

このはからされました

C++ 0xにセマフォはありませんかスレッドをさせるには

Examples

セマフォ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;
};
```

セマフォークラスの

のは、4つのスレッドをします。3つのスレッドはセマフォをし、セマフォは1のカウントにされます。のスレッドはnotify_one()びし、のスレッドの1つをさせます。

その、s1 すぐにをし、セマフォのcountは1にまります。のスレッドは、notifyがびされるまでをします。

```
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();

    ...
}
```

オンラインでセマフォをむ <https://riptutorial.com/ja/cplusplus/topic/9785/セマフォ>

65: セマンティクスをする

Examples

セマンティクスをする

セマンティクスは、C++でオブジェクトをのオブジェクトにするです。このために、いオブジェクトをにして、それをすべてしいオブジェクトにします。

このためには、がであるかをするがあります。の `T&&` `T`はオブジェクトは、の `T&`、はのとはれま
すとあまりわりません。しかし、それらは2つのなるとしてします。したがって、セマンティクス
をうにとなる、あるタイプまたはのタイプをるコンストラクタまたはをできます。

2つのなるタイプがなは、2つのなるをすることです。のコンストラクタはコピーにし、のコンス
トラクタはにしています。

オブジェクトをするには、`std::move(obj)`をします。これは、そのオブジェクトのしいへのデー
タをむことができるように、オブジェクトへののをします。これについては、でするいくつかのが
あります。

なのは、`std::move`をすると、のみ`std::move`されることです。いえれば、`std::move(obj)`ステート
メントは`obj`のをしませんが、`auto obj2 = std::move(obj)` おそらくはいます。

コンストラクタの

このコードスニペットがあるとします。

```
class A {  
public:  
    int a;  
    int b;  
  
    A(const A &other) {  
        this->a = other.a;  
        this->b = other.b;  
    }  
};
```

コピーコンストラクタをする、つまりオブジェクトをコピーしてしいオブジェクトをするをする
には、はのをします。AのオブジェクトへののをるAのコンストラクタがあります。このメソッド
でオブジェクトをでコピーします。

あるいは、`A(const A &) = default;`くこともでき`A(const A &) = default;`コピーコンストラクタを
して、にすべてのメンバーをコピーします。

ただし、コンストラクタをするには、のわりにをします。

```

class Wallet {
public:
    int nrOfDollars;

    Wallet() = default; //default ctor

    Wallet(Wallet &&other) {
        this->nrOfDollars = other.nrOfDollars;
        other.nrOfDollars = 0;
    }
};

```

いをzeroしていることにしてください。デフォルトのコンストラクタ `Wallet(Wallet&&) = default;` は、`nrOfDollars`のをPODとしてコピーします。

セマンティクスはのインスタンスから「スチール」をするようにされているため、このスチールにのインスタンスがどのようにえるかをすることがです。この、をゼロにしないと、ドルを2にすることになります。

```

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

```

したがって、私たちはいものからオブジェクトをしました。

のはなですが、ムーブコンストラクタのをしています。リソースがしているなど、よりなにはよりになります。

```

// 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);
}
};

```

りての

をつオブジェクトにをりてるとに、オブジェクトをコピーすることなく、オブジェクトをのにすることもできます。々はこのりてとぶ。あるオブジェクトからののオブジェクトにをします。

このためには、`operator =`をオーバーロードするがあります。これは、コピーのようにのをるのではなく、のをとるようにします。

```

class A {
    int a;
    A& operator= (A&& other) {
        this->a = other.a;
        other.a = 0;
        return *this;
    }
};

```

これはりてをするなです。 `operator =`をオーバーロードして、それに`r`をえ、それをのオブジェクトにりてることができる。

```

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

```

したがって、オブジェクトをのオブジェクトにりてることができます。

On²からOnへのさをらすためにstd::moveをする

C++ 11では、オブジェクトのになコアとライブラリのサポートがされました。アイデアは、オブジェクト₀がと1のときということであるOににまだし、コピーし、そのようになりてられたバッファとしてだけでビルファ—O「sのリソースへの、その、コピーをんでいます。

コアのサポートはに

- ビルダー_{&&}、例えば、std::string&&へのでstd::string、そのオブジェクトとぶことをしているが、そのリソースだけくすねすることができますなものであるすなわち
- これらのリソースをにコピーするのではなく、されたのオブジェクトからリソースをにするとえられるコンストラクタ T(T&&)なサポート
- auto operator=(T&&) -> T&にサポートしています。これもソースからするはずです。

ライブラリのサポートは、に<utility>ヘッダのstd::moveテンプレートです。これは、されたオブジェクトへのrvalueをします。これは、であるかのように、であることをします。

コンテナの、のコピーはにOnのさです。ここで、nはコンテナのアイテムのです。はO1、です。そして、そのにコピーするコンテナnアルゴリズムのために、これははではOn²にちょうどリニアONからのさをすることができます。

Andrew Koenigは、[20139のDr. Dobbs Journalの"Containers That Never Changes"](#)では、にかなプログラミングスタイルをする、アルゴリズムののいをしました。このスタイルでは、ループはにをしてされます。また、Collatzシーケンスのなどのアルゴリズムでは、にコンテナをにコピーするがあります。

```
// 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';
}

```

```
42 21 64 32 16 8 4 2
```

ベクトルのコピーによるアイテムコピーのは、 $1 + 2 + 3 + \dots n$ のであるため、ここではおおよそ n^2 です。

には、g++およびVisual C++コンパイラでは、の`collatz(42)`びしによって、ベクトルコピーコンストラクティブしでCollatzシーケンスが8のアイテムと36のアイテムコピー $8 * \text{collatz}(42) = 28$ になりました。

これらのアイテムのコピーはすべて、にがなベクトルをするだけでできます。これをうには、ベクトルの`const`とをし、ベクトルをしするがあります。のりはすでににされています。ベクトルがされ、でこれされないびしでは、にそれらをコピーするのではなく、`std::move`をして`std::move`させてください

```

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>() );
}

```

ここで、g++およびVisual C++コンパイラでは、ベクトルコピーコンストラクティブしによるコピーのは、に0でした。

アルゴリズムは、まだずしもこのCollatzシーケンスのさはONであり、これはになである $ON^2 \rightarrow ON$ 。

いくつかのサポートでは、おそらくをしても、そののとののをし、します。その、そののはエラーになります。Alas、C++ 14、C++ではこれがサポートされていません。ループフリーのコードでは、のno useは、`struct result;`とに、な`struct`としてするのによってにできます`struct result;`これはいものであり、のプログラマーにはされそうにない。もかなりをくがあります。

すると、C++とライブラリのサポートにより、アルゴリズムのさはにされますが、サポートののために、コードののとコードのさをにして`const`がすることができます。

をつために、コピーコンストラクタのびしによるアイテムコピーのをするためにされるされたベクトルクラス

```

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_ ) )
    {}
};

```

コンテナでのセマンティクスの

コンテナをコピーするわりにすることができます。

```

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 assignment.
    // 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);
}

```

したオブジェクトをする

したオブジェクトをすることができます

```
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));
}
```

オンラインでセマンティクスをするをむ <https://riptutorial.com/ja/cplusplus/topic/2129/セマンティクスをする>

66: タイプキーワード

Examples

クラス

1. クラスのをします。

```
class foo {
    int x;
public:
    int get_x();
    void set_x(int new_x);
};
```

2. のがクラスであることをするなをしました。クラスがすでにされているは、のでされていてもつけられます。クラスがすでにされていないは、されます。

```
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. テンプレートのにパラメータをします。

```
template <class T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

4. テンプレートテンプレートパラメータのでは、キーワード `class` がパラメータのにします。テンプレートテンプレートパラメータのはクラステンプレートにすぎないため、ここでの `class` のはです。しかし、C++のにはそれがです。

```
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. センス2とセンス3をじでみわせることができます。えば

```
template <class T>
class foo {
};

foo<class bar> x; // <- bar does not have to have previously appeared.
```

C++ 11

6. enumのまたはでは、enumをスコープきenumとしてします。

```
enum class Format {
    TEXT,
    PDF,
    OTHER,
};
Format f = F::TEXT;
```

のいをいて、classとがあります

- キーワードstructをしてクラス・タイプがされている、ベースおよびメンバーのデフォルトのアクセシビリティは、privateはなくpublicです。
- structは、テンプレートパラメータまたはテンプレートテンプレートパラメータをするためにすることはできません。classだけができます。

1. のをします。

```
enum Direction {
    UP,
    LEFT,
    DOWN,
    RIGHT
};
Direction d = UP;
```

C++ 11

C++ 11では、enum スコープきenumをするために、classまたはstructにオプションでenumことができます。また、がスコープとののは、そのとなるタイプがですることができる: T、のTをします。

```
enum class Format : char {
    TEXT,
    PDF,
    OTHER
};
Format f = Format::TEXT;
```

```
enum Language : int {
    ENGLISH,
    FRENCH,
    OTHER
};
```

のenumのenumは、enumがされているスコープにあるとえられますが、スコープのにenumこともできます。

```
Language l1, l2;

l1 = ENGLISH;
l2 = Language::OTHER;
```

2. のがにされたのであることをするなをしました。なは、をするためにはできません。は、のでされていてもこのでをけることができます。

```
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. enumをせずにする、なenumをしました。これは、にされたenumをするか、またはにされていないenumをすることができます。

スコープとしてされたenumは、でunscopedとしてすることはできません。のすべてのは、になるでなければなりません。

スコープのをする、のがわかるまですることはできないため、になるはにするがあります。

```
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
```

1. ユニオンタイプのをします。

```
// Example is from POSIX
union sigval {
    int      sival_int;
    void     *sival_ptr;
};
```

2. のがなのであることをするなをしました。がにされているは、のでされていてもそれをつける

ことができます。がすでにされていないは、されます。

```
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
};
```

オンラインでタイプキーワードをむ <https://riptutorial.com/ja/cplusplus/topic/7838/タイプキーワード>

67: タイプ

201411、C++は、をしておよびブレースされたイニシャライザのタイプのルールをするN3922をしました。これはC++のではありませんが、のコンパイラでされています。

Examples

コンストラクタのテンプレートパラメータ

C++ 17よりでは、テンプレートでコンストラクタのクラスタイプをすることはできません。にするがあります。しかし、これらのはにいく、ラムダのはをけることができないので、

`make_pair()`、`make_tuple()`、`back_inserter()`などのファクトリが`make_pair()`ます。

C++ 17

これはもはやではありません

```
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_iterator<std::vector<int>>
std::lock_guard lk(mtx); // std::lock_guard<decltype(mtx)>
```

コンストラクターはクラステンプレートのパラメータをすとなされますが、これはであり、なガイドをすることができます。

```
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>
```

テンプレートタイプの

テンプレートのな

```
template<typename T>
void f(ParamType param);

f(expr);
```

ケース1 `ParamType`はまたはポインタですが、ユニバーサルまたはフォワードはありません。この、タイプはこうにします。コンパイラは、が`expr`にする、それをします。コンパイラは`expr`のを`ParamType`とパターンマッチさせて`T`をします。

```
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&

```

ケース2 `ParamType` はまたはです。この、`expr` が `rvalue` の、は `case 1` とじです。 `expr` がの、`T` と `ParamType` がとしてされます。

```

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&&

```

ケース3 `ParamType` はポインタでもでもありません。 `expr` がの、はされます。 `expr` が `const` のもされます。それがであるは、`T` のをするにもされます。

```

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

```

オートタイプの

C++ 11

`auto` キーワードをするタイプのは、テンプレートタイプのとほぼじです。にいくつかのをします。

```

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), cs'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&&

```


はのとおりで。

```
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>
```

でまれたイニシャライザをするとわかるように、`auto`はに`std::initializer_list<T>`のをします。 `T`できない、コードはされます。

のりのとして`auto`がされているは、そのにのりのがあることをします。

```
auto f() -> int {
    return 42;
}
```

C ++ 14

C ++ 14では、C ++ 11でのにえて、のことがです。

1. のりのをたないのりのとしてする、のりのを、ののりがあればそれからすることをします。

```
// f returns int:
auto f() { return 42; }
// g returns void:
auto g() { std::cout << "hello, world!\n"; }
```

2. ラムダのパラメータです、ラムダをジェネリックラムダとします。

```
auto triple = [](auto x) { return 3*x; };
const auto x = triple(42); // x is a const int with value 126
```

な`decltype(auto)`は、`auto`ものではなく`decltype`ルールをしてをします。

```
int* p = new int(42);
auto x = *p;           // x has type int
decltype(auto) y = *p; // y is a reference to *p
```

C ++ 03では、`auto`キーワードはCからされたクラスとはくなるをちました。

オンラインでタイプをむ <https://riptutorial.com/ja/cplusplus/topic/7863/タイプ>

68: タイプ

き

とは、になるをクライアントからしながら、さまざまなとのしたインタフェースをできるをするためののセットです。C++では、さまざまなタイプのびしなオブジェクトをするをつ

`std::function<R(A...)>`、おそらくもよくられたのです。

Examples

なみ

のは、オブジェクトがのクラスからしていなくても、オブジェクトをするコードからオブジェクトのをすです。そうすることで、ポリモーフィズムのテンプレート;では、コンパイルになをるがありますが、にインタフェースにするようするはありませんとおよび;では、コンパイルになをるはなく、にインタフェースにするようにするがあります。

のコードは、タイプのなメカニズムをしています。

```
#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;
};
```

サイトでは、をつクラスとに、ののみをするがあります。えば

```
#include <iostream>
```

```
void print_value(Printable const &p)
{
    p.print(std::cout);
}
```

これはテンプレートではなく、ヘッダーファイルですらで、ファイルでできるものであることにしてくださいがでなテンプレートとはなりません。

なので、`Printable`についてるはありません。テンプレートとに、インタフェースにするがあります。

```
struct MyType { int i; };
ostream& operator << (ostream &os, MyType const &mc)
{
    return os << "MyType {" << mc.i << "}";
}
```

このクラスのオブジェクトをでしたにすことができます

```
MyType foo = { 42 };
print_value(foo);
```

vtableでレギュラータイプにする

C++は、RegularまたはなくともPseudo-Regularとしてられています。

レギュラータイプは、コピーやをして、りて、りてがなタイプであり、することができ、しいとすることができます。なしですることでもあります。に、さまざまなstdアルゴリズムやコンテナでなくいくつかのサポートしています。

これは[ルート・ペーパー](#)ですが、C++ 11ではstd::hashサポートをしたいとえています。

ここでは、でvtableでをします。

```
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);
        }
    };
}
}

```

ライブの。

このようなタイプはのためのキーとしてすることができます `std::map` か `std::unordered_map` のように、キーのためのなかをけけます。

```
std::map<regular_type, std::any>
```

には、アニメのレギュラーからコピーなものまでのマップです。

`any`とはなり、の`regular_type`はさなオブジェクトのも、のデータのをサポートしていません。のタイプをにすことはしくありません。

なオブジェクトのでは、されたストレージバッファを`regular_type`にし、`ptr`をくしてオブジェクトをし、しないようにするがあります。

は`make_dtor_unique_ptr`で`make_dtor_unique_ptr`、にバッファにデータをするをえて、バッファにきがなければヒープにれます。それでもしれません。

のみの`std::function`

`std::function`はいくつかのをします。なものの1つは、されたがコピーであることです。

これは、の`ptrs`をする`lambda`のようないくつかのコンテキストでをきこします。スレッドにタスクをディスパッチするスレッドプールのように、コピーがにならないコンテキストで`std::function`をしている、このによってオーバーヘッドがするがあります。

に、`std::packaged_task<Sig>`は、なびしオブジェクトです。あなたは`std::packaged_task<void(Args...)>`に`std::packaged_task<R(Args...)>`をすることができますが、それはかなりく、あまりしないびしクラス。

したがって、`task`。これは、な`std::function`のをしています。コピーコンストラクタをしましたこれには、`clone`メソッドを`details::task_pimpl<...>`するがあります。

```
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());
}
};

```

このライブラリをにするには、さなバッファのをして、ヒープのすべてのびしコードをしないようにします。

SBOをするには、デフォルトの`task(task&&)`、クラスのいくつかの`std::aligned_storage_t`、`destroy-only`メモリをヒープにさないにできる`deleter`をつ`m_pImpl` `unique_ptr`、`emplace_move_to( void* ) = 0`を`task_pimpl`ます。

のコードの SBOなし。

Tのしたバッファまでする

すべてののに、りて、の、またはポインタさえもまれます。

タイプタイプのをうのは、のビヘイビアをし、そのビヘイビアをサポートするのタイプをり、それをラップすることです。そののにまれないすべてののは、「れられた」または「された」。

`array_view`はまたはコンテナをり、それが**T**したバッファであるというをいてすべてをします。

```

// 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は、.data()をサポートするのコンテナを、Tおよび.size()メソッドへのポイントまたはarray_viewり、それをしたTsのランダムアクセスになるまでarray_viewます。

これは `std::vector<T>`、`std::string<T>`、`std::array<T, N>` `T[37]`、イニシャライザリスト `{}` づくものをむ、またはそののものをすることができます `T* x.data()` と `size_t x.size()` `T* x.data()` サポートしています。

この、たちがしているものからすることができるデータは、「ビュー」とともに、メモリをりてるもなく、カスタムをくありません。

。

ADLがなコンテキストでメンバ `data` とメンバ `size` をすることが `size` れます。

タイプをするタイプ **std::any**

このでは、C++ 14 と `boost::any` ます。C++ 17 では `std::any` わりに `std::any` スワップできます。

なはのとおりで。

```
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);
```

これはほぼです。

このは、[@dyp](#) と [@cpplearner](#) ののについています。

まず、をすタグをします。

```
template<class T>struct tag_t{constexpr tag_t(){};};
template<class T>constexpr tag_t<T> tag{};
```

このクラスは、`any_method` されたシグネチャをします。

これは、ポインタとそのポインタのファクトリをし、`any_method` え `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`は、インスタンスとにするポインタのです。

`any_method_function::operator()`は`tag_t<T>`をとり、`tag_t<T> any& T`となることをとした
`any_method_function::type`カスタムインスタンスをきみます。

々は、にのメソッドをタイプイレースできるようにしたい。そのため、タプルでそれらをね、ごと
 にストレージにタプルをりけ、それらへのポインタをするヘルパーラッパをします。

```

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 );
    }
};

```

`vtable`がさいた例えば1にはこのをし、そのようなはのためにクラスにされたポインタをすることが
 できます。

は`super_any`をし`super_any`。は`super_any_t`をとって、`super_any`のをにしています。

```

template<class...methods>
struct super_any_t;

```

これは、スーパーがSFINAEとよりいいエラーメッセージをサポートするメソッドをします。

```
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> ) )
{};
```

に、any_methodをします。any_methodはメソッドポインタです。々はそれをグローバルにし、のようなをしてconstをします。

```
const auto print=make_any_method( [](auto&&self, auto&&os){ os << self; } );
```

またはC++で17

```
const any_method print=[](auto&&self, auto&&os){ os << self; };
```

ラムダをすると、ルックアップステップのためにをするので、をくすることができることにしてください。これはすることができますが、このはにそれよりもくなります。だから、にラムダから、またはラムダでパラメータされたから、のメソッドをします。

```
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)...);
}
};

```

C++でとされないファクトリメソッド17はじています

```

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)};
}

```

これはされた`any`です。これは、で`any`、そしてそれがまれているはいつでもポインタののりにぶ`any`います。

```

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;
}
};

```

any_method を const オブジェクトとしてするので、これにより super_any がしになります。

```

template<class...Ts>
using super_any = super_any_t< std::remove_cv_t<Ts>... >;

```

テストコード

```

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);
}

```

ライブの。

もともとは、SOのとそして、にったの々にされました。

オンラインでタイプをむ <https://riptutorial.com/ja/cplusplus/topic/2872/タイプ>

69: タイプ

は、コンパイルにさまざまなプロパティをしてテストするためにされるテンプレートされたです。コンパイルにのコードのをまたはできるきロジックをするためにできます。ライブラリーは、いくつかのなるをするC++11でもたらされました。のテンプレートをすることもできます。

Examples

C++ 11

`type_traits`ヘッダーには、コンパイルにのプロパティをおよびチェックするためののテンプレートクラスとヘルパーがまれています。

これらは、ユーザーエラーをチェックし、プログラミングをサポートし、をにするためにテンプレートでされます。

ほとんどののは、がいくつかのをたすかどうかをするためにされます。これらのはのとおりです。

```
template <class T> struct is_foo;
```

テンプレートクラスがいくつかの`foo`をたすでインスタンスされている、`is_foo<T>`は`std::integral_constant<bool, true> std::true_type`からし、そうでなければ`std::integral_constant<bool, false> std::false_type`。これにより、にはのメンバーがえられます

```
static constexpr bool value
```

`T`が`foo`をたすは`true`、そうでないは`false`

```
operator bool
```

```
!value
```

C++ 14

```
bool operator()
```

```
!value
```

タイプ

```
value_type  bool
```

type	std::integral_constant<bool,value>

これは、`static_assert`や`std::enable_if`などのでできます。 `std::is_pointer`の

```
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
}
```

`std::add_pointer`や`std::underlying_type`など、をするさまざまなもあります。これらは、に、されたをむの`type`メンバをする。たとえば、`std::add_pointer<int>::type`は`int*`です。

std :: is_same とのの

C++ 11

`std::is_same<T, T>`のは、2つのをするためにされます。boolとしてされます。がじはtrue、そうでないはfalseです。

えは

```
// 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";
```

`std::is_same`のは、`typedef`になくします。これは`int == int32_t`したのでにされていますが、これはにはらかではありません。

えは

```
// Prints true on all compilers.
typedef int MyType
std::cout << std::is_same<int, MyType>::value << "\n";
```


テンプレートされたクラスまたはをにしたときにするために `std::is_same` をする

アサートとみわせると、 `std::is_same` テンプレートは、テンプレートされたクラスとのなをする
でなツールになります。

`int` からのと2つののみをします。

```
#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;
}
```

C++ 11

よりなタイプをするいくつかのなるタイプのがあります。

インテグラル

`int`、 `char`、 `long`、 `unsigned int` などのすべてのので `true` とされます。

```
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.
```

すべてののにしてとします。 `float`、 `double`、 `long double` など

```
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.
```

ですか

`enum class` をむすべてのので `true` とされます。

```
enum fruit {apple, pear, 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.
```

ポインタです

すべてのポインタでとします。

```
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.
```

クラスです

enum class いて、すべてのクラスとでtrueとされます。

```
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.
```

タイプのプロパティ

C++ 11

のプロパティは、なるにできるをします。これらののはずしもらかではない。

のでは、されていないコンパイラのみをしています。なではなく、なです。

4のさでります。

```
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);
}
```

これは、がのときにtrueとされます。

```
std::cout << std::is_const<const int>::value << "\n"; // Prints true.
std::cout << std::is_const<int>::value << "\n"; // Prints false.
```

ですか

タイプがvolatileのときはtrueとされます。

```
std::cout << std::is_volatile<static volatile int>::value << "\n"; // Prints true.
std::cout << std::is_const<const int>::value << "\n"; // Prints false.
```

されている

これは、すべてのされたにしてとされます。

```
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.
```

されていない

すべてのなしタイプでとされます。

```
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.
```

オンラインでタイプをむ <https://riptutorial.com/ja/cplusplus/topic/4750/タイプ>

70: テンプレート

き

クラス、およびC++ 14のはテンプレートすることができます。テンプレートは、すべてのパラメータがされたときになクラス、またはになるいくつかのきパラメータをつコードです。パラメータには、またはテンプレートをできます。よく知られているテンプレートは`std::vector`であり、のがされたときになコンテナになります。例えば、`std::vector<int>`です。

- `template < template-parameter-list >`
- テンプレートをエクスポートする`< template-parameter-list > /* C++ 11まで */`
- テンプレート`<>`
- テンプレート
- `extern`テンプレート `/* C++ 11 */`
- テンプレート`< template-parameter-list >`クラス... `opt opt`
- テンプレート`< template-parameter-list >`クラス `opt = id-expression`
- テンプレート`< template-parameter-list >` `typename ... opt opt /* C++17 */`
- テンプレート`< template-parameter-list >` `opt = id-expression /* C++17 */`
- `postfix-expression`。テンプレート`ID-expression`
- `postfix-expression ->`テンプレート`id-expression`
- ネストされたの `template simple-template-id ::`

`template`は、コンテキストにじて、C++で5つのなるをつキーワードです。

1. `<>`でまれたテンプレートパラメータのリストのには、[クラステンプレート](#)、[テンプレート](#)、またはのテンプレートの[な](#)などのテンプレートがされます。

```
template <class T>
void increment(T& x) { ++x; }
```

2. そののにの `<>`があると、[なな](#)がされます。

```
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. そのに`<>`ないがくと、[なインスタンス](#)のまたはをします。

```
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. テンプレートパラメータリストには、テンプレートテンプレートパラメータがされています。

```
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. スコープ::およびクラスメンバーアクセスの.および->、のがテンプレートであることをします。

```
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
    }
};
```

C++ 11のに、テンプレートをしてすることができ`export` キーワード エクスポートテンプレートにそれをります、。エクスポートされたテンプレートのは、テンプレートがインスタンスされるすべてののにするはありません。たとえば、のようにするはです。

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
}
```

のしさから、`export` キーワードはほとんどのコンパイラでサポートされていませんでした。これはC++ 11でされました。、`export` キーワードをまったくすることはです。そのわりに、はヘッダーにテンプレートをするがあります、ヘッダーにされていないテンプレートのとはなります。[ヘッダーファイルにテンプレートのみをできるを](#)してください。

Examples

テンプレート

テンプレートは、じをつよりなとににもできます。

```
// '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;
}
```

これはテンプレートとじでできます。

```
printSum<int>(4, 5);
printSum<float>(4.5f, 8.9f);
```

どちらのも、テンプレートのはパラメータのをきえるためにされます。はのC++と同じようにしますパラメータがコンパイラがのをするテンプレートのとしない。

テンプレートのもう1つのテンプレートクラスとはなりは、にされるパラメータについてテンプレートパラメータをできるということです。

```
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.
```

このにより、テンプレートとをみわせるときにコードをすることができます。ライブラリには、ヘルパー `make_X()` をって `template structure X` をるためののパターンがあります。

```
// 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};
}
```

これはどのようにちますか

```
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 paramters.
                                           // In this code both val1 and val2 are the same
                                           // type.
```

これはコードをするためのものではありません。これは、コードをよりにするためにされています。のではなく、のでコードをすることで、をすることができます。

Templateは、をしてとののをけることができます。

```
template <typename T>
void f(T &&t);
```

この、のタイプの t はコンテキストにじてされます。

```
struct X { };

X x;
f(x); // calls f<X&>(x)
f(X()); // calls f<X>(x)
```

のケースでは、タイプ T は $X \ X\&$ へのとしてされ、 t のタイプは X へのであり、2のケースでは T のタイプが X としてされ、 $t \ X \ X\&\&$ 。

のケースでは、`decltype(t)`は T とじであるが、2のケースではないことにすることはです。

t をのにするには、でもでも、`std::forward` をうがあります。

```
template <typename T>
void f(T &&t) {
    g(std::forward<T>(t));
}
```

```
}
```

フォワーディングリファレンスはバリデーションナルテンプレートでできます

```
template <typename... Args>
void f(Args&&... args) {
    g(std::forward<Args>(args)...);
}
```

はテンプレートパラメータにしてのみできます。たとえば、のコードでは`v`ではなく `rvalue`です。

```
#include <vector>

template <typename T>
void f(std::vector<T> &&v);
```

クラステンプレート

クラステンプレートのなえは、テンプレートパラメータがコンパイルにできえられることです。その、じクラスをのにしてすることができます。ユーザーは、クラスのがされたときにされるをします。これの3つのが `main()` されています

```
#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
}

```

テンプレートの

テンプレートクラス/メソッドののインスタンスのをできます。

たとえば、のような

```

template <typename T>
T sqrt(T t) { /* Some generic implementation */ }

```

あなたはのようによくことができます

```

template<>
int sqrt<int>(int i) { /* Highly optimized integer implementation */ }

```

`sqrt(4.0)` をくユーザはジェネリックなをますが、`sqrt(4)` はなをます。

テンプレートの

なテンプレートのとはに、なテンプレートのにより、のテンプレートののいくつかをしたテンプレートをすることができます。なテンプレートのは、テンプレートクラス/にしてのみできます。

```

// 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 }
};

```

のように、テンプレートのは、データとのメンバーのになるセットをすることがあります。

にされたテンプレートがインスタンスされると、もしたがされます。たとえば、テンプレートと2つのなをしましょう。

```

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";
    }
};
```

すぐのびし

```
S<std::string, int, double>::foo();
S<int, float, std::string>::foo();
S<int, double, std::string>::foo();
```

する

```
General case
T = int
T = int, U = double
```

テンプレートはにされているだけです

```
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;
}
```

デフォルトのテンプレートパラメータ

の他に、テンプレートパラメータはデフォルトをつことができます。デフォルトをつすべてのテ

テンプレートパラメータは、テンプレートパラメータリストのにするがあります。なえは、テンプレートのインスタンスにデフォルトをつテンプレートパラメータをできるということです。

デフォルトのテンプレートパラメータの

```
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;
}
```

エイリアステンプレート

C++ 11

な

```
template<typename T> using pointer = T*;
```

このは、`pointer<T>`を`T*`エイリアスにし`T*`。えは

```
pointer<int> p = new int; // equivalent to: int* p = new int;
```

テンプレートはできません。ただし、のネストされたをすることによって、にそのをできます。

```
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;
```

テンプレートテンプレートパラメータ

によっては、をせずにテンプレートにテンプレートをしたいがあります。これはテンプレートテンプレートパラメータがされるためのものです。になテンプレートテンプレートのパラメータの

```

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;
    }
}

```

テンプレートを**auto**でする

C++ 17では、テンプレートのパラメータをするときは、そのをにしなければなりません。だから、のパターンはのようなものになりました

```

template <class T, T N>
struct integral_constant {
    using type = T;
    static constexpr T value = N;
};

using five = integral_constant<int, 5>;

```

しかし、なの、このようなものをうには `decltype(expr), expr` テンプレートをインスタンスするときに `decltype(expr), expr` をくがあります。は、このイディオムをし、に `auto` することです

C++ 17

```

template <auto N>
struct integral_constant {
    using type = decltype(N);
    static constexpr type value = N;
};

```

```
};

using five = integral_constant<5>;
```

unique_ptrののカスタムデリター

なのでのカスタムと、のベースのをしようとしてからることができるunique_ptr。なるC APIのDeleterはりのがなりますが、はありません。らかのできるようにしたいだけです。

```
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>>;
```

そして、りのにわらず、_Tをテンプレートののないパラメータとしてることができるポインタをにうことができ、そこからサイズのオーバーヘッドのunique_ptrりすことができます

```
unique_ptr_deleter<std::FILE, std::fclose> p;
```

タイプのテンプレートパラメータ

テンプレートパラメータのとはに、のいずれかのをたすのをすることができます。

- または、
- オブジェクトへのポインタまたはへのポインタ、
- lvalueへのオブジェクトまたは、
- メンバーへのポインタ、
- std::nullptr_t。

すべてのテンプレートパラメータとに、テンプレートパラメータは、にするか、デフォルトにするか、またはテンプレートのきとしをしてにできます。

タイプのテンプレートパラメータの

```
#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: " << sizeof(anArrayOfChar) << "\n";

int anArrayOfData[] = {1,2,3,4,5,6,7,8,9};
std::cout << "anArrayOfData: " << sizeof(anArrayOfData) << "\n";
}

```

タイプとタイプのテンプレートパラメータをにする

```

#include <array>
int main ()
{
    std::array<int, 5> foo; // int is a type parameter, 5 is non-type
}

```

タイプのテンプレートパラメータは、テンプレートのををするの1つであり、[メタプログラミング](#)をにします。

なテンプレートデータ

C++ 14

コンパイルにされるとのデータメンバーをつクラスまたはををするとなことがよくあります。なは `std::tuple` ですが、のカスタムをするがあることもあります。のは、`std::tuple` ようなではなく、をしてををするです。なのでまります。これは、ののきののベースケースとしてもします。

```

template<typename ... T>
struct DataStructure {};

```

これにより、の、`DataStructure<> data` をすることはすでにですが、まだそれほどではありません。

に、なケースのがわれます。

```

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;
};

```

これで、`DataStructure<int, float, std::string> data(1, 2.1, "hello")` のようなデータをするだけです。

どうしたのに、これは、なくとも1つのなテンプレートパラメータすなわち、の`T` がし、バック`Rest` のをにしないというをするであることにされたい。`T` がすることをすることで、`first` データメ

ンバーのがになります。りのデータは、`DataSetructure<Rest ... > rest`としてにパッケージされます。コンストラクターは、`rest`メンバーへのコンストラクターびしをむ、これらのメンバーのをします。

これをよりよくするために、をとってすることができます`DataSetructure<int, float> data`というがあらとします。はにとし、`int first`と`DataSetructure<float> rest`データメンバーをつをします。`rest`は、このとびし、の`float first`と`DataSetructure<> rest`メンバーをします。に、このの`rest`は、ベースケースのとし、のをします。

これをのようになすることができます

```
DataSetructure<int, float>
-> int first
-> DataSetructure<float> rest
    -> float first
    -> DataSetructure<> rest
        -> (empty)
```

はデータをとていますが、々のデータににアクセスすることはできません`DataSetructure<int, float, std::string> data`ののメンバーにアクセス`data.rest.rest.first`、にはユーザーフレンドリーではありません。だから々は、`get` ベースケースのはするデータがないとしてのみになことにメソッドを`get`

```
template<typename T, typename ... Rest>
struct DataSetructure<T, Rest ...>
{
    ...
    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataSetructure<T, Rest...>>::get(*this);
    }
    ...
};
```

このように、`get`メンバはテンプレートされていますこのは、なメンバのインデックスにあります。したがって、`data.get<1>()`は`std::tuple`た`data.get<1>()`ようなものになります。のは、ヘルパークラス`GetHelper`によってわれま。 `DataSetructure`の`get`なをすることができないのは、`idx`をすることがあるからですただし、すぐにわかるように、`idx`クラスをせずにテンプレートメンバーをすることはできませんテンプレート。ここでC++ 14スタイルの`auto`うと、りののがかなりになるので、たちののははるかにになります。

それでヘルパークラスに。のはのフォワードと2つのスペシャライゼーションがになります。に

```
template<size_t idx, typename T>
struct GetHelper;
```

、ベースケース `idx==0`とき。この、`first`メンバをすだけです。

```
template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};
```

なケースでは、`idx`をらし、`rest`メンバーの`GetHelper`をびします。

```
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);
    }
};
```

を`data.get<1>()`は`DataStructure<int, float> data`があり、`data.get<1>()`がであるとします。これは`GetHelper<1, DataStructure<int, float>>::get(data)` `GetHelper<0, DataStructure<float>>::get(data.rest)`びす`GetHelper<1, DataStructure<int, float>>::get(data)` 2のをびします。の`idx`ような1のによって`data.rest.first`。

そうです `main`でのをいくつかげてみましょう

```
#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;
}

```

なインスタンス

なインスタンスのは、テンプレートをせずに、なクラス、またはをテンプレートからおよびします。なインスタンスは、のからすることができます。これは、のセットでのみインスタンスされる、ヘッダーファイルでのテンプレートのをけるためにできます。えは

```

// 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*);

```

`print_string<char>`と`print_string<wchar_t>`は`print_string.cpp`でにインスタンスされているため、`print_string`テンプレートがヘッダーにされていなくても、リンカーはそれらをつけることができます。これらのなインスタンスがしない、リンカエラーがするがあります。 [ヘッダーファイルにテンプレートのみをできるを](#)してください。

C++ 11

なインスタンスのに`extern` [キーワード](#)がある、わりになインスタンスになります。されたのためのなインスタンスのは、ののされたのなインスタンスをぎます。わりになインスタンスをきこすそのへののは、またはのTUのなインスタンスをすることができます。

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
}
```

オンラインでテンプレートをむ <https://riptutorial.com/ja/cplusplus/topic/460/テンプレート>

71: パラメータパック

Examples

パラメータパックをむテンプレート

```
template<class ... Types> struct Tuple {};
```

パラメータパックは、0のテンプレートをけるテンプレートパラメータです。テンプレートになくとも1つのパラメータパックがあるは、バリデーショナルテンプレートです。

パラメータパックの

パターン `parameter_pack ...` は、 `parameter_pack` カンマでられたリストとそのパラメータのそれぞれにされます

```
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
}
```

のコードは、 `variadic_printer(1, 2, 3, "hello");` びされました `variadic_printer(1, 2, 3, "hello");` プリント

```
1
2
3
hello
```

オンラインでパラメータパックをむ <https://riptutorial.com/ja/cplusplus/topic/7668/パラメータパック>

72: ビットフィールド

き

ビットフィールドは、CおよびC++をしっかりとパックしてサイズをします。これはしないようにえます。メンバーのビットをし、コンパイラはビットをします。このは、ビットフィールドメンバのアドレスをすることができないことである。これは、ビットフィールドメンバがしてされているからである。 `sizeof()` もされています。

ビットフィールドのコストは、メンバーをまたはするためにメモリをし、ビットのをするがあるため、アクセスがくなります。これらは、なサイズにもされます。

ビットはどれくらいのかかりますかなビットフィールドをします。

```
struct foo {
    unsigned x;
    unsigned y;
}
static struct foo my_var;
```

のコードでは、

```
my_var.y = 5;
```

`sizeof (unsigned) == 4`、`x`はのにされ、`y`は4バイトにされます。されるアセンブリコードはのようになります。

```
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
```

これは、`x`が`y`とされていないため、です。しかし、ビットフィールドをとてをするとしてください

```
struct foo {
    unsigned x : 4; /* Range 0-0x0f, or 0 through 15 */
    unsigned y : 4;
}
```

`x`と`y`に4ビットがりとられ、1バイトをします。したがって、は8ではなく1バイトをします`y`にしたとしてアセンブリを`y`とします。

```
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
```

これらのがないしある、これはトレードオフになるがあり、キャッシュにメモリをしたり、スワッピングをしたり、ファイルをらませてこれらのをさせたり、すべてのこととに、いをしてください。

デバイスドライバのデバイスドライバのなとしてビットフィールドをしないでください。ビットフィールドのレイアウトはコンパイラでずしもしているわけではなく、そのようなはありません。へのリード・モディファイ・ライトは、しないをきこすデバイスがしないをするがあります。

Examples

と

```
struct FileAttributes
{
    unsigned int ReadOnly: 1;
    unsigned int Hidden: 1;
};
```

ここでは、これらの2つのフィールドのそれぞれが1ビットのメモリをします。これは、のに: 1でされます。ビットフィールドのはの8ビットintから64ビットintです。 `unsigned` をすることをおめします。そうでないときがじることがあります。

よりくのビットがなは、「1」をなビットにきえます。えは

```
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
};
```

のはわずか22ビットをしており、のコンパイラでは、このの `sizeof` は4バイトになります。

いはとてもシンプルです。をし、ののようになしてください。

```
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;
```

オンラインでビットフィールドをむ <https://riptutorial.com/ja/cplusplus/topic/2710/ビットフィールド>

73: ビット

`std::bitset`をするには、`<bitset>`ヘッダをインクルードする必要があります。

```
#include <bitset>
```

`std::bitset`すべてのものをオーバーロードし、ビットセットのc-styleとじをにします。

- [Bit Twiddling Hacks](#)

Examples

ビットをする

Cスタイルのビット

ビットは、ビットのOR `|` をしてできます。

```
// Bit x will be set  
number |= 1LL << x;
```

std :: bitsetをする

`set(x)`または`set(x,true)` - xビットを1します。

```
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
```

シクリアする

Cスタイルのビット

ビットは、ビットのAND `&` をしてクリアできます。

```
// Bit x will be cleared  
number &= ~(1LL << x);
```

std :: bitsetをする

reset(x)またはset(x, false) - xビットをクリアします。

```
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
```

ビットをトグルする

Cスタイルのビット

XOR ^ をしてビットをりえることができます。

```
// Bit x will be the opposite value of what it is currently
number ^= 1LL << x;
```

std :: bitsetをする

```
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)
```

しする

Cスタイルのビット

をにxシフトし、にビットでAND & をすると、そのビットのがられます。

```
(number >> x) & 1LL; // 1 if the 'x'th bit of 'number' is set, 0 otherwise
```

シフトは、きシフトまたはなしシフトのいずれかとしてできます。numberにおけるnumber >> xされたタイプとのをする、られたは、です。

そのビットのがインプレースでなは、わりにマスクをシフトしておくことができます。

```
(number & (1LL << x)); // (1 << x) if the 'x'th bit of 'number' is set, 0 otherwise
```

すべてのゼロはとみなされるので、いずれかをとしてできます。

std :: bitsetをする

```
std::bitset<4> num(std::string("0010"));  
bool bit_val = num.test(1); // bit_val value is set to true;
```

nのビットをxにする

Cスタイルのビット

```
// Bit n will be set if x is 1 and cleared if x is 0.  
number ^= (-x ^ number) & (1LL << n);
```

std :: bitsetをする

set(n, val) - ビット_nをvalします。

```
std::bitset<5> num(std::string("00100"));  
num.set(0,true); // num is now 00101  
num.set(2,false); // num is now 00001
```

すべてのビットを

Cスタイルのビット

```
x = -1; // -1 == 1111 1111 ... 1111b
```

なぜこれがし、にはのであるかについては[こちら](#)をしてください。

std :: bitsetをする

```
std::bitset<10> x;  
x.set(); // Sets all bits to '1'
```

のセットビットをする

Cスタイルのビット


```
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);
}
```

- n がゼロの、 $0 \& 0xFF..FF$ は0です
- そうでなければ n は $0bxxxxxx10..00$ とくことができ、 $n - 1$ は $0bxxxxxx011..11$ 、 $n \& (n - 1)$ は $0bxxxxxx000..00$ 。

カウントビットの

ビットストリングのは、やそののアプリケーションでしばしばとされ、このはくされています。

ナイーブなでは、ビットごとに1のがです。

```
unsigned value = 1234;
unsigned bits = 0; // accumulates the total number of bits set in `n`

for (bits = 0; value; value >>= 1)
    bits += value & 1;
```

なトリック [Remove rightmost set bit](#)について

```
unsigned bits = 0; // accumulates the total number of bits set in `n`

for (; value; ++bits)
    value &= value - 1;
```

それは、されたビットとじくらくのりしをるので、`value`がゼロビットをほとんどたない`value`がされるときはいことです。

このはPeter Wegner [CACM 3 / 322-1960](#)がにしたものであり、Brian W. KernighanとDennis M. RitchieのCプログラミングにして、よくられています。

これには12のがで、そのうちの1つはです。

```
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) + ...
}
```

こののではののがありますはハミングウェイトをしてください。

くのCPUにはのx86のpopcntがあり、コンパイラはの ではない みみをすることができます。例えばg++をすると、のようになります。

```
int __builtin_popcount (unsigned x);
```

が2のであるかどうかをチェックする

$n \& (n - 1)$ トリック のセットビットをするをは、が2のであるかどうかをするのにもちます

```
bool power_of_2 = n && !(n & (n - 1));
```

チェックのの $n \&&$ がなければ、0はって2のとみなされます。

ビットアプリケーション

ビットのいくつかのアプリケーションの1つは、マスクとなビットをすることによって、をからに、またはそのにすることです。たとえば、はこの $2_{01(1)00001}$ をち、のは $01(0)00001$ です。それらはのビットだけでなる。この、からへのは、にのビットを1にすることです。そうするために、たちはのことをいいます

```
/******
convert small letter to captial letter.
=====
a: 01100001
mask: 11011111 <-- (0xDF)  11(0)11111
:-----
a&mask: 01000001 <-- A letter
*****/
```

をAにするコードはのとおりで。

```
#include <stdio>

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;
}
```

はのとおりで。

```
$ g++ main.cpp -o test1
```

```
$ ./test1  
a (AND) mask = A  
a    &    0xDF = A
```

オンラインでビットをむ <https://riptutorial.com/ja/cplusplus/topic/3016/ビット>

74: ビット

ビットシフトは、すべてのプロセッサアーキテクチャにわたってではなく、なるプロセッサはなるビットをすることができる。いえれば、あなたがいた

```
int a = ~0;
int b = a << 1;
```

これは、64ビットマシンと32ビットマシン、またはx86ベースプロセッサからPICベースプロセッサにかけてなります。

エンディアンは、ビットのではありません。つまり、シフト `>>` はビットをビットにシフトし、XORはまたはビットをします。エンディアンはデータそのものをするがあります。つまり、エンディアンがアプリケーションにとってされる、ビットのになくになります。

Examples

- ビット AND

```
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;
```

a = 6, b = 10, c = 2

なぜ

ビットでANDがビットレベルでし、のブールをします。

```
TRUE  AND TRUE  = TRUE
TRUE  AND FALSE = FALSE
FALSE AND FALSE = FALSE
```

a 0110 のバイナリと b 1010 のバイナリがANDばれると、0010バイナリがられます。

```
int a = 0 1 1 0
int b = 1 0 1 0 &
      -----
int c = 0 0 1 0
```

ビットのANDは、ビットの&=をすることによりてられていないり、ののをしません。

```
int a = 5;  // 0101b (0x05)
a &= 10;    // a = 0101b & 1010b
```

I - ビットOR

```
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;
```

a = 5, b = 12, c = 13

なぜ

ビットのORは、ビットレベルでし、のブールをします。

```
true OR true = true
true OR false = true
false OR false = false
```

a 0101 のバイナリとb 1100 のバイナリがORされると、バイナリ1101られます。

```
int a = 0 1 0 1
int b = 1 1 0 0 |
          -----
int c = 1 1 0 1
```

ビットの|=をすることによりてられていないり、ビットのORはののをしません。

```
int a = 5;  // 0101b   (0x05)
a |= 12;    // a = 0101b | 1101b
```

^ - ビットごとのXOR

```
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;
```

a = 5, b = 9, c = 12

なぜ

ビットでXOR ORはビットレベルでし、のブールをします。

```
true OR true = false
true OR false = true
false OR false = false
```

XORのtrue OR true = false、のtrue AND/OR true = true、したがってXORのにしてください。

これをして、 a 0101 のバイナリと b 1001 のバイナリがXORされると、 1100バイナリがられます。

```
int a = 0 1 0 1
int b = 1 0 0 1 ^
      -----
int c = 1 1 0 0
```

ビットのXORは、ビットの^=をすることにより行われていない、のをしません。

```
int a = 5; // 0101b (0x05)
a ^= 9;    // a = 0101b ^ 1001b
```

ビットのXORは、くのででき、とのためのビットマスクでよくされます。

のは、しばしばらしいトリックのとしてされています。しかし、のコードではしないでください
std::swap() じをするためのstd::swap() がいがあります。

また、XORをして、2つのをにすることもできます。

```
int a = 42;
int b = 64;

// XOR swap
a ^= b;
b ^= a;
a ^= b;

std::cout << "a = " << a << ", b = " << b << "\n";
```

これをするには、それをできることをするためにをすることがあります。

```
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;
    }
}
```

だから、それはなるらしいテクニックのようにえますが、のコードではにちません。 xorはではなく、のみわせである $a \oplus c = a \oplus (a \oplus c)$

2015+のコンパイラでも、バイナリとして行っていることができます

```
int cn=0b0111;
```

ビットのNOT

```
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;
```

a = 234, b = 21

なぜ

ビットワイズNOTはビットレベルでし、にビットをします。それはだは₁、にわたったの₀それはだは、₀、それがわたったの₁。ビットワイズNOTは、のタイプのにしてをXORするのとじをちます。

```
unsigned char a = 234; // 1110 1010b (0xEA)
unsigned char b = ~a;  // 0001 0101b (0x15)
unsigned char c = a ^ ~0;
```

ビットワイズNOTは、ののをチェックするなです。

```
unsigned int i = ~0;
unsigned char c = ~0;

std::cout << "max uint = " << i << std::endl <<
            "max uchar = " << static_cast<short>(c) << std::endl;
```

ビットワイズNOTは、ののをせず、をたないため、えは $a \sim 10$ を $a \sim 10$ ことはできません。

ビットワイズ NOT ~ は NOT ! としないでください。NOTはそのをとてをいます $(!1) != (\sim 1)$ つまり、 $(!1) != (\sim 1)$

<< - シフト

```
int a = 1; // 0001b
int b = a << 1; // 0010b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

a = 1, b = 2

なぜ

のビットのワイズシフトは、の_aのビットを₁にされただけシフトし、にビットを0でめみ、₅ 2 0000 0101 4えは、 $5 \ll 4$ は80 20101 0000 のをもたらす。をに1シフトすること、に2をけることとじです

```
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;
}

```

しかし、シフトでは、ビットをむすべてのビットがにシフトされます

```

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;

```

な a = 2147483647, b = -2

コンパイラによってはされるがられるものもありますが、ビットがをけるようにきをシフトしたままにした、はになります。シフトするビットのがのであるか、またはのができるビットのよりもきいは、です

```

int a = 1;
int b = a << -1; // undefined behavior
char c = a << 20; // undefined behavior

```

ビットのシフトは、ビットのりて<<=をすることによりてられていないり、ののをしません。

```

int a = 5; // 0101b
a <<= 1; // a = a << 1;

```

>> - シフト

```

int a = 2; // 0010b
int b = a >> 1; // 0001b

std::cout << "a = " << a << ", b = " << b << std::endl;

```

a = 2, b = 1

なぜ

ビットワイズシフトは、の_aのビットを₁にされたにシフトします。シフトのはであるが、きのシフトのビットにはがこるのかはであるため、がされないことにするべきである。

```

int a = -2;
int b = a >> 1; // the value of b will be depend on the compiler

```

シフトするビットがのであるは、これもです


```
int a = 1;
int b = a >> -1; // undefined behavior
```

ビットのシフトは、ビットのりて>>=をすることによりてられていないり、ののをしません。

```
int a = 2; // 0010b
a >>= 1; // a = a >> 1;
```

オンラインでビットをむ <https://riptutorial.com/ja/cplusplus/topic/2572/ビット>

75: ビルドシステム

き

C++は、Cのように、コンパイルのワークフローやビルドプロセスにするがく、々です。、C++には、プログラムをコンパイルするためにされる、さまざまなビルドシステムがあります。ここでは、いくつかのビルドシステムをしてします。

、しているクロスプラットフォームであるC++のためのなまたはなビルドシステムはしません。しかし、なプラットフォーム/プロジェクトにはいくつかのなビルドシステムがしますが、もすべきはGNU/LinuxオペレーティングシステムではGNU Make、Visual C++ / Visual StudioプロジェクトシステムではNMAKEです。

さらに、のIDEには、ネイティブIDEでにされるなビルドシステムもまれています。のビルドシステムジェネレータは、EclipseのCMakeやMicrosoft Visual Studio 2012などのネイティブIDEビルドシステム/プロジェクトフォーマットをできます。

Examples

CMakeを使ったビルドの

CMakeは、ほぼすべてのコンパイラやIDEのビルドをのプロジェクトからします。のは、[クロスプラットフォームの "Hello World" C++コード](#)に CMakeファイルをするをします。

CMakeファイルはに "CMakeLists.txt"というで、すべてのプロジェクトのルートディレクトリにするはずでサブディレクトリもです。なCMakeLists.txtファイルはのようになります

```
cmake_minimum_required(VERSION 2.4)

project(HelloWorld)

add_executable(HelloWorld main.cpp)
```

それは[Coliru](#)にきててください。

このファイルは、CMakeにプロジェクト、するファイルのバージョン、main.cppがな "HelloWorld"というファイルをするためのをえます。

インストールされたコンパイラ/IDEのビルドをコマンドラインからします。

```
> cmake .
```

のようにアプリケーションをビルドします。

```
> cmake --build .
```

これにより、OSおよびインストールされているツールにじて、システムのデフォルトのビルドがされます。「アウトソース」ビルドをして、ビルドからソースコードをきれいに

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

CMakeは、のプラットフォームシェルのコマンドをすることもできます

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

CMakeには、いくつかのビルドツールとIDEのジェネレータがまれています。 [Visual Studioのnmake](#) メークファイルをするには

```
> cmake -G "NMake Makefiles" ..
> nmake
```

GNU makeでコンパイルする

ま

GNU Make styled `make` は、シェルコマンドをするためのプログラムです。GNU Makeは、Makeファミリーにするのプログラムです。`make`は、Linuxカーネル、Mac OS X、およびBSDからしたものをもUnixライクなオペレーティングシステムとPOSIXライクなオペレーティングシステムのでがあります。

GNU Makeは、なGNU / LinuxオペレーティングシステムにしているGNUプロジェクトにされていることでのにする。GNU Makeには、WindowsとMac OS Xの々なでするバージョンもあります。これはなをつにしたバージョンでもがあります。これらのから、GNU MakeはしばしばCやC++とにえられます。

なルール

`make`でコンパイルするには、プロジェクトディレクトリにMakefileをします。あなたのMakefileはのようにシンプルになります

メークファイル

```
# 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)
```

インデントが4つのスペースではなく、タブでにわれるようにしてください。そうしないと、`Makefile:10: *** missing separator. Stop.` がないというエラーがされます
`Makefile:10: *** missing separator. Stop.`

これをコマンドラインからするには、のようになります。

```
$ cd ~/Path/to/project
$ make
$ ls
app main.cpp Makefile

$ ./app
Hello World!

$ make clean
$ ls
main.cpp Makefile
```

インクリメンタルビルド

よりくのファイルをしめると、makeがよりになります。 **b.cpp**ではなく**a.cpp**をするとどうなり

ますか **b.cpp**をコンパイルするとがっかりします。

のディレクトリをします。

```
.
+-- src
|   +-- a.cpp
|   +-- a.hpp
|   +-- b.cpp
|   +-- b.hpp
+-- Makefile
```

これはいMakefileでしょう

メークファイル

```
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
```

もうタブをてください。このしいMakefileをすると、されたファイルのみをコンパイルし、コンパイルをにえることができます。

ドキュメンテーション

makeのについては、[Free Software Foundationのドキュメント](#)、[stackoverflowのドキュメント](#)、[dmckeeのstackoverflowにするなえ](#)をしてください。

SConsでする

Scons - Pythonのソフトウェアツールをして、**クロスプラットフォームの "Hello World" C ++コード**をすることができます。

に、 `SConstruct` という `SConstruct` ファイルをします `SCons` はデフォルトでこの `hello.cpp` のファイルをします。のところで、ファイルは `hello.cpp` ったディレクトリになければなりません。しいファイルにをきむ

```
Program('hello.cpp')
```

、から、 `scons` し `scons` 。あなたはかのようにえるはずです

```
$ cons
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.
```

はおいのオペレーティングシステムとインストールされているコンパイラによってなります。

`Environment` クラスと `Glob` クラスは、するものをさらにするのにちます。えは、 `SConstruct` ファイル

```
env=Environment(CPPPATH='/usr/include/boost/',
    CPPDEFINES=[],
    LIBS=[],
    SCONS_CXX_STANDARD="c++11"
)

env.Program('hello', Glob('src/*.cpp'))
```

`src` すべての `cpp` ファイルをして、な `hello` をビルドします。その `CPPPATH` は `/usr/include/boost` あり、`C ++ 11` をします。

き

ビルドシステムは、プロジェクトウェブサイトで「**スピードにをいたビルドシステム**」としてされています。 `Ninja` は、システムファイルをすることでファイルをするようにされており、`CMake` や `Meson` などのレベルビルドシステムマネージャとはに、システムをするためのレベルのアプローチをしています。

`Ninja` は、に `C ++` と `Python` でかれており、`Chromium` プロジェクトの `SCons` ビルドシステムのとしてされました。

NMAKE **Microsoft** プログラムメンテナンスユーティリティ

NMAKEは、にMicrosoft Visual Studioおよび/またはVisual C ++コマンドラインツールとみわけてするためにMicrosoftによってされたコマンドラインユーティリティです。

NMAKEはビルドシステムのMakeファミリにするビルドシステムですが、WindowsのファイルパスUnixスタイルのファイルパスとはなりませんをサポートするなど、UnixライクなMakeプログラムとはなるいくつかのなるをえています。

オートツールGNU

オートツールは、のソフトウェアパッケージのGNUビルドシステムをするプログラムです。MakefileGNU Makeでされるなど、さまざまなビルドリソースをするためににくのツールです。したがって、Autotoolsはのビルドシステムジェネレータとえることができます。

すべきAutotoolsプログラムには、

- Autoconf
- Automake make としないでください

に、AutotoolsはUnixのスクリプトとMakefileをして、のコマンドがほとんどのパッケージをビルドするだけでなくインストールできるようにしています。

```
./configure && make && make install
```

したがって、Autotoolsは、のパッケージマネージャ、にPOSIXにするオペレーティングシステムにされているパッケージマネージャとのもっています。

オンラインでビルドシステムをむ <https://riptutorial.com/ja/cplusplus/topic/8200/ビルドシステム>

76: ファイル I/O

き

C++ファイルのはストリームでわれます。なはのとおりです。

テキストをむための `std::istream`。

テキストをくための `std::ostream`。

`std::streambuf` を `std::streambuf` でをみきします。

きは `operator>>` し `operator>>`。

フォーマットされたは `operator<<` します。

ストリームは、フォーマットのやエンコーディングとエンコーディングのののために、えは `std::locale` し `std::locale`。

ストリームの [`<iostream>` ライブラリ](#)

Examples

ファイルをく

ファイルをくことは、3つのファイルストリーム `ifstream`、`ofstream`、および `fstream` すべてにしでじでわれます。

コンストラクタでファイルをくことができます

```
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.
```

または、ファイルストリームのメンバ `open()` をすることもできます。

```
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.
```


ファイルがにけられたかどうかきみであってもにするがあります。には、ファイルがしない、ファイルがなアクセスをたない、ファイルがすでにである、ディスクエラーがした、ドライブがされている、などがまれます。

```
// 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");
}
```

ファイルパスにバックスラッシュがまれているWindowsシステムなど、にエスケープするがあります。

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs("c:\\folder\\foo.txt"); // using escaped backslashes
```

C++ 11

のリテラルをする

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs(R"(c:\folder\foo.txt)"); // using raw literal
```

わりにスラッシュをしてください。

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs("c:/folder/foo.txt");
```

C++ 11

、WindowsのパスにASCIIをむファイルをくは、のワイドパスをできます。

```
// Open the file 'пример\foo.txt' on Windows.
std::ifstream ifs(LR"(пример\foo.txt)"); // using wide characters with raw literal
```

ファイルからのみみ

ファイルからデータを見るにはいくつかのがあります。

データのフォーマットがわかっているは、ストリーム >> をできます。のデータをむfoo.txtというのファイルがあるとします。

```
John Doe 25 4 6 1987
Jane Doe 15 5 24 1976
```

に、のコードをしてファイルからそのデータを見ることができます。

```
// 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.
```

ストリーム>>すべてのをし、できないやであればします。

- の、はまたは `\n` で。
- の、はのでします。

つまり、`foo.txt`ファイルののバージョンものコードでにみられます。

```
John
Doe 25
4 6 1987
```

```
Jane
Doe
15 5
24
1976
```

ストリーム>>にえられたストリームをします。したがって、データをしてみるために、のをさせることができます。ただし、ストリームをブールとしてすることもできますのコードの`while`ループにされています。これは、ストリームクラスには`bool`のがあるためです。この`bool()`は、ストリームにエラーがなければ`true`をし`true`。ストリームがエラーになるとえば、それデータをできないなど、`bool()`は`false`をし`false`。したがって、のコードの`while`ループは、ファイルがまでみられたでします。

ファイルをとしてみるは、のコードをします。

```
// 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>());
```

このコードでは、なメモリリテをらすためにstringスペースをしていstring。

ファイルを1getline()みたいは、getline()うことができます

```
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.
}
```

のをみたいは、ストリームメンバread()することができます。

```
std::ifstream is("foo.txt");
char str[4];

// Read 4 characters from the file.
is.read(str, 4);
```

みりコマンドをしたは、エラーフラグfailbitがされているかどうか、がしたかどうかをにするがあります。これはファイルストリームのメンバfail()びすことうことができます

```
is.read(str, 4); // This operation might fail for any reason.

if (is.fail())
    // Failed to read!
```

ファイルへのきみ

ファイルにきむにはいくつかのがあります。もなは、ファイルストリーム ofstream をストリーム << とにすることです。

```
std::ofstream os("foo.txt");
if(os.is_open()){
    os << "Hello World!";
}
```

<<わりに、ファイルストリームのメンバwrite()することもできます。

```
std::ofstream os("foo.txt");
if(os.is_open()){
    char data[] = "Foo";

    // Writes 3 characters from data -> "Foo".
    os.write(data, 3);
}
```

ストリームにきんだ、エラーフラグbadbitがされているかどうかは、がしたかどうかをすため、ににするがあります。これは、ファイルストリームのメンバbad()びすことによてうことができます

す。

```
os << "Hello Badbit!"; // This operation might fail for any reason.
if (os.bad())
    // Failed to write!
```

オープニングモード

ファイルストリームをするときに、くモードをできます。モードは、にストリームがファイルをくをするです。

すべてのモードはstd::iosにあります。

モードは、ファイルストリームのコンストラクタまたはそのopen()メンバの2パラメータとしてすることができます。

```
std::ofstream os("foo.txt", std::ios::out | std::ios::trunc);

std::ifstream is;
is.open("foo.txt", std::ios::in | std::ios::binary);
```

しいデフォルトをとっていても、iostreamメンバーによってにされていないのフラグをするには、ios::inまたはios::outをするがあることにしてください。

モードをししないと、のデフォルトモードがされます。

- ifstream - in
- ofstream out
- fstream - inとout

デザインでできるファイルオープンモードはのとおりです。

モード		にとって	
app	する		ファイルのにデータをします。
binary	バイナリ		とはバイナリでわれます。
in			みむためにファイルをきます。
out			きみのファイルをきます。
trunc	りめる		くときにファイルのをします。
ate	に		くときにファイルのにします。

binaryモードをすると、データをそのままみきすることができます。それをしないと、'\n'をプラットフォームのシーケンスとのですることができます。

たとえば、Windowsでは、のはCRLF `"\r\n"` です。

きみ `"\n" => "\r\n"`

んでください `"\r\n" => "\n"`

ファイルをじる

にファイルをじることは、C++ではほとんどありません。ファイルストリームはデストラクタのするファイルをにじるためです。ただし、にファイルハンドルをいたままにしないように、ファイルストリームオブジェクトのをするようにしてください。たとえば、これは、すべてのファイルをのスコープ `{}` にれることによってできます。

```
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.
```

`close()` にびすことは、でじ `fstream` オブジェクトをしたいが、そのにファイルをいたままにしたいにのみです。

```
// 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();
```

ストリームをフラッシュする

くのタイプのストリームとに、ファイルストリームはデフォルトでバッファリングされます。つまり、ストリームへのきみによって、になるファイルがすぐにされることはありません。すべ

てのバッファリングされたきみがににされるようにするには、ストリームをフラッシュできます。これは、`flush()` メソッドをびすか、`std::flush` ストリームマニピュレータをしてうことができます。

```
std::ofstream os("foo.txt");
os << "Hello World!" << std::flush;

char data[3] = "Foo";
os.write(data, 3);
os.flush();
```

ストリームをフラッシュしてをくことをみわせたストリームマニピュレータ `std::endl` があります

```
// Both following lines do the same thing
os << "Hello World!\n" << std::flush;
os << "Hello world!" << std::endl;
```

バッファリングは、ストリームへのきみパフォーマンスをさせることができます。したがって、くのきみをうアプリケーションでは、なフラッシュをけるがあります。に、I/Oがにわれない、ストリームオブジェクトにデータがしないように、アプリケーションはにフラッシングをするがあります。

ASCII ファイルを `std::string` にみむ

```
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()`
}
```

`rdbuf()` メソッドは、`rdbuf()` `stringstream::operator<<` メンバをして `buffer` にブッシュできる `streambuf` へのポインタをします。

の [Scott Meyers の Effective STL](#) でしているはのとおりです。

```
std::ifstream f("file.txt");

if (f)
{
    std::string str((std::istreambuf_iterator<char>(f)),
                    std::istreambuf_iterator<char>());

    // Operations on `str`...
}
```

これは、コードをほとんどとしないだけでなく、のSTLコンテナにファイルをみむこともですが
きなファイルではなくなるがあるため、いいです。

コンストラクタへのののりのなカッコは、もなをぐためにです。

なくともものではなく

```
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`...
}
```

これはおそらくもいされた3つのででしょう。

ファイルをコンテナにみむ

のでは、`std::string`と`operator>>`を使ってファイルからをみみます。

```
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);
}
```

のでは、`operator>>`を使って、に1つの「アイテム」をみるファイルをにしています。このじは、ス
トリームからに1つの「アイテム」をみむイテレータである`std::istream_iterator`をしてできます
。また、ほとんどのコンテナは2つのイテレータをしてすることができるので、のコードをすると
のようになります。

```
std::ifstream file("file3.txt");

std::vector<std::string> v(std::istream_iterator<std::string>{file},
                          std::istream_iterator<std::string>{});
```

これをして、`std::istream_iterator`へのテンプレートパラメータとしてみみたいオブジェクトを
するだけで、きなオブジェクトをみむことができます。このように、のように、のようなではな
くをみむことができます

```

// 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>{});

```

フォーマットされたテキストファイルから`struct`をみます。

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

```

```

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

```

ファイルのコピー

```

std::ifstream src("source_filename", std::ios::binary);
std::ofstream dst("dest_filename", std::ios::binary);
dst << src.rdbuf();

```

C++ 17

C++ 17では、ファイルをコピーするためのなは、`<filesystem>` ヘッダを`copy_file`をすること
`copy_file`。

```

std::filesystem::copy_file("source_filename", "dest_filename");

```

ファイルシステムライブラリはもともと`boost.filesystem`としてされ、にC++ 17のでISO C++に
マージされ`boost.filesystem`。

ループ、いのでファイルのわりをチェックしていますか

`eof`は、ファイルのわりをんだでのみ`true`します。のみみがストリームのわりになることをすもの
ではありません。

```
while (!f.eof())
{
    // Everything is OK

    f >> buffer;

    // What if *only* now the eof / fail bit is set?

    /* Use `buffer` */
}
```

あなたはしくくことができます

```
while (!f.eof())
{
    f >> buffer >> std::ws;

    if (f.fail())
        break;

    /* Use `buffer` */
}
```

しかし

```
while (f >> buffer)
{
    /* Use `buffer` */
}
```

シンプルでエラーのがない

さらなる

- `std::ws` ストリームからするをする
- `std::basic_ios::fail` するストリームでエラーがしたは `true` し `true`

のロケールでファイルをきむ

なるロケールをしてデフォルトにファイルをきむがあるは、`std::locale`および
`std::basic_ios::imbue()` をしてのファイルストリームにしてうことができます。

のき

- ファイルをくに、にストリームにローカルをするがあります。
- ストリームにがめまれたら、ロケールをししないでください。

ののロケールがにしていないか、ファイルのをしていない、ロケールをむファイルストリームのめみはのをします。

UTF-8ストリームおよびそののストリームはにしません。また、UTF-8ロケールをむファイルス

トリームは、ファイルがかれたときにBOMマーカ―をファイルからみみ、みることがあります。ファイルをくだけでファイルからがみまれ、にはされません。

```
#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 ofs1;
    ofs1.imbue(std::locale(""));
    ofs1.open("file1.txt");
    ofs1 << 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;
}
```

クラシック "C"ロケールににりえることは、プログラムがのデフォルトロケールをし、ファイルのみきにされたをしたいにです。 "C"ロケールをすると、このでは

```
78,123.456
78,123.456
78123.456
```

たとえば、ロケールがドイツで、したがつてなるをする、このでは、

```
78 123,456
78,123.456
78123.456
```

のののカンマにしてください。

オンラインでファイルI/Oをむ <https://riptutorial.com/ja/cplusplus/topic/496/ファイルi---o>

77: プリプロセッサ

き

Cプリプロセッサは、コードのコンパイルにされるテキストパーサ/リプレイサです。CおよびそのC++のバージョンに、するためにされ、のでできます。

- a. `#include` を使ってのファイルをインクルードする
- b. `#define` をしてテキストマクロを `#define`
- c. `#if #ifdef` をしたときコンパイル
- d. プラットフォーム/コンパイラのロジック ときコンパイルのとして

プリプロセッサのは、ソースファイルがコンパイラにされるにされます。にレベルのきロジックがです。プリプロセッサの例えばオブジェクトのようなマクロはののようにけされていないのでコンパイルにステップがする、コンパイラはチェックをすることができないのでくするがあります。

Examples

ガードをめる

ヘッダファイルは、のヘッダファイルによってまれることがあります。したがって、のヘッダーをむソースファイルコンパイルは、にのヘッダーをむことがあります。このようなヘッダファイルにのがまれていると、コンパイラは1つの2003 C++の§3.2などOne Definition Ruleのをし、とコンパイルにします。

ヘッダーガードまたはマクロガードともばれる「インクルードガード」をすると、のインクルードがされます。これらは、プリプロセッサの `#define`、`#ifndef`、`#define #endif` ディレクティブをしてされています。

えば

```
// Foo.h
#ifndef FOO_H_INCLUDED
#define FOO_H_INCLUDED

class Foo    // a class definition
{
};

#endif
```

インクルードガードをするなは、すべてのコンパイラとプリプロセッサですることです。

ただし、プロジェクトでされるすべてのヘッダーでマクロがであることをするがあるため、ガードをインクルードすると、にとってがします。には、2つまたはそのヘッダーが`FOO_H_INCLUDED`をインクルードガードとしてする、コンパイルにまれるのヘッダーは、のヘッダーがインクルードされることをにします。プロジェクトでされるヘッダーファイルをむサードパーティライブラリののをにすると、のがします。

また、インクルードガードでされるマクロがヘッダーファイルでされたのマクロとしないようにするがあります。

ほとんどのC++では、`#pragma once`ディレクティブもサポートされてい`#pragma once`これは、ファイルが1のコンパイルで1だけまれることをします。デファクトスタンダードですが、ISO C++のではありません。え

```
// Foo.h
#pragma once

class Foo
{
};
```

`#pragma once`はインクルードガードにするいくつかのをしますが、でされている`#pragma`はにコンパイラのフックであり、サポートしていないコンパイラによってってされます。`#pragma once`をするプロジェクトは、それをサポートしていないコンパイラにするのがしくなります。

C++のくのコーディングガイドラインとでは、ヘッダーファイルを`#include`するに、ヘッダーにインクルードガードをするに、プリプロセッサのをにしています。

きロジックおよびクロスプラットフォーム

にえ、きロジックは、マクロをしてコンパイルにコードロジックをまたはにすることにするものです。

3つのなユースケースがあります

- ・じアプリのとなるのあるのアプリプロファイル デバッグ、リリース、テスト、などなロギングなど
- ・クロスプラットフォームのコンパイル - のコードベース、のコンパイルプラットフォーム
- ・のなるをえたのアプリケーションバージョン えば、ソフトウェアのBasic、Premium、Proバージョンにのコードベースをする。

aファイルをするためのクロスプラットフォームのアプローチ

```
#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
}

```

`_WIN32`、`__APPLE__`、`_WIN32`、または`__unix__`のようなマクロは、`、`するによってにされています。

b デバッグビルドのログをにする

```

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"
}

```

cのビルドでプレミアムをにするこれはであり、アプリケーションをインストールせずにのロックをすることをめします

```

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
}

```

いくつかのなトリック

びしのシンボルの

プリプロセッサは、されたシンボルでびすことができますオプションのを。たとえば、このコマンド `gcc -E` はプリプロセッサだけをします

```
gcc -E -DOPTIMISE_FOR_OS_X -DTESTING_MODE=1 Sample.cpp
```

`#define OPTIMISE_FOR_OS_X`と`#define TESTING_MODE 1`が`Sample.cpp`のにされたのと同じで`Sample.cpp`をします。

マクロがされていることをする

マクロがされておらず、そのがまたはチェックされている、ほとんどの、プリプロセッサはを0とします。これにはいくつかのがあります。1つのアプローチは、デフォルトが0としてされ、すべてのえ、アプリケーションビルドプロファイルへのがにされるがあるということですデフォルトではENABLE_EXTRA_DEBUGGING = 0、デフォルトでは-DENABLE_EXTRA_DEBUGGING = 1がきされます。もう1つのアプローチは、すべてのとデフォルトをにすることです。これは#ifdefと#errorディレクティブをみわせてできます。

```
#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
```

マクロ

マクロは、オブジェクトのようなマクロとなマクロの2つのなグループにされます。マクロは、コンパイルプロセスのいでトークンのとしてわれま。これは、コードのきなまたはりしをプリプロセッサマクロにできることをします。

```
// 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);
```

Qtライブラリーは、ユーザーがQObjectをするユーザークラスのにQ_OBJECTマクロをすることによって、このテクニックをしてメタオブジェクト・システムをします。

マクロは、すべてのでかれており、のコードとのをにします。これはではありませんが、にくのプログラマがいスタイルとえるだけです。

オブジェクトのようなマクロがかった、そのマクロのがそのにきえられて、なコピーペーストとしてされます。のようなマクロにすると、そのとそのパラメータのがされます。

```
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))
```

このため、のようなマクロパラメータは、の`AREA()`のように、にまれることがよくあります。これは、マクロにするのあるバグ、にのマクロパラメータがのでされるバグをぐためです。

```
#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));
```

また、このなのために、しないをぐために、マクロにされるパラメータにするがあります。にパラメータがされた、マクロでされるたびにパラメータがされますが、これははなものではありません。がかをすのをぐために、マクロがかってパラメータをむでも、これはてはまります。

```
int oops = 5;
double incremental_damage = AREA(oops++);
// Compiler sees:
double incremental_damage = (3.14159265358979*(oops++)*(oops++));
```

さらに、マクロはをしなため、のにするしにくいエラーにつながります。

プログラマがセミコロンでをさせるので、スタンドアロンとしてされるマクロはセミコロンを「みむ」ようにされていることがよくあります。なセミコロンによってしないバグがするのをぎます。

```
#define IF_BREAKER(Func) Func();

if (some_condition)
    // Oops.
    IF_BREAKER(some_func);
else
    std::cout << "I am accidentally an orphan." << std::endl;
```

このでは、ったセミコロンが`if...else`ブロックをし、コンパイラが`else`を`if`マッチさせ`if`ます。これをぐために、セミコロンはマクロからされます。セミコロンは、セミコロンをしたにセミコロンを「りめ」ます。

```
#define IF_FIXER(Func) Func()

if (some_condition)
    IF_FIXER(some_func);
else
```



```
std::cout << "Hooray! I work again!" << std::endl;
```

のセミコロンをしておく、のステートメントをすることなくマクロをすることができます。これはです。

```
#define DO_SOMETHING(Func, Param) Func(Param, 2)

// ...

some_function(DO_SOMETHING(some_func, 3), DO_SOMETHING(some_func, 42));
```

、マクロはでします。ただし、マクロがのをカバーするがあるは、にバックスラッシュをしてこれをすことができます。このバックスラッシュは、ののでなければなりません。これは、プリプロセッサにして、のをのにして1としてうことをします。これはしてできます。

```
#define TEXT "I \
am \
many \
lines."

// ...

std::cout << TEXT << std::endl; // Output: I am many lines.
```

これは、のをカバーするのがある、なのようなマクロでにです。

```
#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'.")
```

よりなのようなマクロの、のとに、なのをしたり、マクロのにオブジェクトをしたりするためののスコープをえることはです。これは`do-while`ブロックにまれている0の、なイディオムです。このブロックにはにセミコロンがいていないため、セミコロンをみむことができます。

```
#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);
```

variadicマクロもあります。に、これらは、のを、その、な「」パラメータのわりに、それらすべてをする__VA_ARGS__。

```
#define VARIADIC(Param, ...) Param(__VA_ARGS__)

VARIADIC(sprintf, "%d", 8);
// Compiler sees:
sprintf("%d", 8);
```

、__VA_ARGS__はのどこにでもでき、しくされることにしてください。

```
#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);
```

がゼロのパラメータの、なるコンパイラがのカンマをなでします。Visual Studioなどのコンパイラの中には、なをわずにカンマをかにみむものがあります。GCCのようなコンパイラでは、__VA_ARGS__に##をくがあります。このため、がされるには、きでマクロをすることがです。

```
// 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 */
```

プリプロセッサエラーメッセージ

プリプロセッサをしてコンパイルエラーをできます。これは、サポートされていないプラットフォームまたはサポートされていないコンパイラにどうかをユーザーにするなど、いくつかのがまれています。

gccのバージョンが3.0.0であればエラーをします。

```
#if __GNUC__ < 3
#error "This code requires gcc > 3.0.0"
#endif
```

アップルコンピュータでコンパイルするは、エラーをします。

```
#ifdef __APPLE__
#error "Apple products are not supported in this release"
#endif
```

みのマクロ

あらかじめされたマクロは、コンパイラによってされたマクロですソースファイルのユーザとはなりません。これらのマクロは、ユーザによってされたり、されたりしてはなりません。

のマクロは、C++によってされています。

- `__LINE__` このマクロはでされるのをみ、によってすることができる`#line`ディレクティブ。
- `__FILE__` は、このマクロがされているファイルのファイルがまれており、`#line`ディレクティブでできます。
- `__DATE__` にをむ"Mmm dd yyyy"へのびしによってられたかのようにうーんがフォーマットされたファイルのコンパイル、の `std::asctime()`
- `__TIME__` には、ファイルコンパイルの "hh:mm:ss"がまれます。
- `__cplusplus` は、するC++コンパイラによってされ、C++ファイルをコンパイルします。そのはすなわち、コンパイラはとにしているバージョンである199711L、C++ 98とC++ 03のため201103L C++ 11とのため201402L C++ 14のために。

C++ 11

- `__STDC_HOSTED__` は、がホストされているは1、フリースタンディングのは0されます。

C++ 17

- `__STDCPP_DEFAULT_NEW_ALIGNMENT__` には`size_t` リテラルがまれてい`size_t` リテラルは、アラインメント。`__STDCPP_DEFAULT_NEW_ALIGNMENT__` `operator new` へのびしにされるアラインメントです。

さらに、のマクロはによってされていて、していなくてもかまいません。

- `__STDC__` はのをち、はなCをすためにファイルをCとしてコンパイルするときのみにされます。あるいは、コンパイラがこのマクロをサポートしないことにしたはしてしない。

C++ 11

- `__STDC_VERSION__` はインプリメンテーションのをち、`__cplusplus` がC++バージョンとじように、そのはCバージョンです。または、コンパイラがこのマクロをサポートしないことをしたでも、されていません。
- `(uintmax_t)'x' != (uintmax_t)L'x'` のように、`__STDC_MB_MIGHT_NEQ_WC__` が1にされている、セットのいエンコーディングのは、
- `__STDC_ISO_10646__` は、`wchar_t` がUnicodeとしてエンコードされているにされ、サポートされているのUnicodeリビジョンをす`yyyymmL`のにされます。
- `__STDCPP_STRICT_POINTER_SAFETY__` は、がなポインタのをっているは1にされます そうでない、ポインタのがされます
- `__STDCPP_THREADS__` は1にされます。プログラムがのスレッドをつことができるに - ホストではにのスレッドをつことができます

`__func__` はマクロではなく、あらかじめされたローカルです。これは、されるのを、のとしてみます。

これらのなされたマクロのに、コンパイラはあらかじめされたマクロをつことができます。それらをぶには、コンパイラのドキュメントをするがあります。えは

- [gcc](#)
- [Microsoft Visual C ++](#)
- [インテル®C ++コンパイラー](#)

いくつかのマクロは、いくつかののサポートをするだけです。

```
#ifdef __cplusplus // if compiled by C++ compiler
extern "C"{ // C code has to be decorated
    // C library header declarations here
}
#endif
```

はデバッグににです

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;
}
```

そしてなバージョンのためののもの

```
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マクロ

コンパイルにりしコードをするための。

Xマクロは、リストとリストのという2つのでされています。

```
#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;
}

```

これはプリプロセッサによってのようになります。

```

Animal dog;
Animal cat;
Animal racoon;

int main() {
    dog.say();
    cat.say();
    racoon.say();

    return 0;
}

```

リストがきくなるにつれ100の、このテクニックはのコピーリけをけるのにちます。

https://en.wikipedia.org/wiki/X_Macro

Xマクロ

LISTをするにぎのないのXしていないは、マクロをとしてすこともできます。

```

#define LIST(MACRO) \
    MACRO(dog) \
    MACRO(cat) \
    MACRO(racoon)

```

ここで、リストをするときにどのマクロをうべきかをにします。

```

#define FORWARD_DECLARE_ANIMAL(name) Animal name;
LIST(FORWARD_DECLARE_ANIMAL)

```

MACROをびすたびにリストにしてのパラメータがなは、バリデージックマクロをできます

```
//a workaround 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__))
```

のはLISTによってされ、りはLISTびしでユーザーによってされます。えは

```
#define FORWARD_DECLARE(name, type, prefix) type prefix##name;
LIST(FORWARD_DECLARE, Animal, anim_)
LIST(FORWARD_DECLARE, Object, obj_)
```

にする

```
Animal anim_dog;
Animal anim_cat;
Animal anim_racoon;
Object obj_dog;
Object obj_cat;
Object obj_racoon;
```

#pragma once

ほとんどのC++では、`#pragma once`ディレクティブがサポートされてい`#pragma once`これは、ファイルが1のコンパイルで1だけまれることをします。ISO C++のではありません。えは

```
// Foo.h
#pragma once

class Foo
{
};
```

`#pragma once` はインクルードガードにするいくつかのをしますが、でされている`#pragma`はにコンパイラのフックであり、サポートしていないコンパイラによってってされます。 `#pragma once` をするプロジェクトは、にするようにするがあります。

いくつかのコンパイラ、にプリコンパイルされたヘッダーをするコンパイラでは、`#pragma once` をするとコンパイルプロセスがにされるがあります。に、いくつかのプリプロセッサは、どのヘッダがされているかをするによってコンパイルのをする。 `#pragma once` と`#pragma once` がされているのは、にし、コンパイルのまたはのいずれかになります。

`#pragma once` とみわせてをめるヘッダファイルのレイアウトウィンドウのMFCベースのアプリケーションを、およびVisual Studioのでされたた`add class`、`add dialog`、`add windows`ウィザードが。したがって、それらをC++ Windowsアプリケーションでみわせることはにです。

プリプロセッサ

##またはstringizingは、マクロパラメータをリテラルにするためにされます。をつマクロでのみできます。

```
// 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");
```

コンパイラは2つのをし、のprintf()はリテラルになり、にがきます。

プリプロセッサは、マクロのまたはのスペースをします。したがって、printのでじがられます。

```
PRINT(   This line will be converted to string by preprocessor );
```

リテラルのパラメータがののようなエスケープシーケンスをとする、プリプロセッサによってにされます。

```
PRINT(This "line" will be converted to "string" by preprocessor);
// Compiler sees
printf("This \"line\" will be converted to \"string\" by preprocessor""\n");
```

##またはトークンペーストは、マクロの2つのパラメータまたはトークンをするためにされます。

```
// 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);
```

なは

```
variableY = 15
```

オンラインでプリプロセッサをむ <https://riptutorial.com/ja/cplusplus/topic/1098/プリプロセッサ>

78: フレンドキーワード

き

うまくされたクラスは、そのをカプセルし、をしながら、きれいにされたインターフェイスをします。これにより、インタフェースがされていないり、またはがになります。

よりなシナリオでは、おいののにするのクラスがになることがあります。フレンドクラスとファンクションは、ドキュメントされたインタフェースのカプセルとをなうことなく、これらのピアがいのにアクセスできるようにします。

Examples

フレンド

クラスまたはは、それがであるをすることができます。がクラスのフレンドである、はされているメンバーとプライベートメンバーのすべてにアクセスできます。

```
// 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;
}
```

アクセスは、セマンティクスをしません。の、、はです。

フレンドのはされません。たとえば、たちが`PrivateHolder`をサブクラスしているとし

`PrivateHolder`。

```
class PrivateHolderDerived : public PrivateHolder {
```



```
public:
    PrivateHolderDerived(int val) : PrivateHolder(val) {}
private:
    int derived_private_value = 0;
};
```

メンバーにアクセスしようとすると、のようになります

```
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;
}
```

PrivateHolderDerivedメンバは、PrivateHolder::private_valueにアクセスすることはできませんが、friendはそれをできます。

の

メソッドは、としてだけでなく、としてすることもできます

```
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;
}
```

フレンドクラス

クラスがとしてされるかもしれません。フレンドクラスは、フレンドののメンバーがクラスのプライベートでされたメンバーにアクセスできることをします。

```
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;
}

```

Friendクラスではありません。クラスにのプライベートアクセスがなは、ともフレンドがです。

```

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;
};

```

オンラインでフレンドキーワードをむ <https://riptutorial.com/ja/cplusplus/topic/3275/フレンドキーワード>

79: フロー

ループについては、[ループのトピック](#)をしてください。

Examples

switchのcaseラベルをします。オペランドは、のスイッチとしていなければなりません。switchがされると、にしいオペランドをつcaseラベルにジャンプします。

```
char c = getchar();
bool confirmed;
switch (c) {
    case 'y':
        confirmed = true;
        break;
    case 'n':
        confirmed = false;
        break;
    default:
        std::cout << "invalid response!\n";
        abort();
}
```

スイッチ

C++によれば、

switchステートメントは、のにじて、いくつかのステートメントの1つにをします。

キーワードswitchは、カッコでまれたとブロックがきdefault。ブロックには、caseラベルとオプションのdefaultラベルがまれているcaseがあります。switchがされると、のとするをつcaseラベルするcaseまたはdefaultラベルするcaseにがされます。

は、またはでなければならず、または、またはまたはへのをつクラスでなければなりません。

```
char c = getchar();
bool confirmed;
switch (c) {
    case 'y':
        confirmed = true;
        break;
    case 'n':
        confirmed = false;
        break;
    default:
        std::cout << "invalid response!\n";
        abort();
}
```

キャッチ

`catch` キーワードは、ハンドラ、つまりのあるのがスローされたときにコントロールがされるブロックをします。 `catch` キーワードのには、でまれたがきます。これは、のパラメータとています。パラメータはでき、のにする...ができます。ハンドラは、そのがのとがあるにのみをします。については、 [のキャッチをしてください](#)。

```
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;
}
```

デフォルト

`switch`では、のが`case`ラベルののいずれともしくないにジャンプするラベルがされます。

```
char c = getchar();
bool confirmed;
switch (c) {
    case 'y':
        confirmed = true;
        break;
    case 'n':
        confirmed = false;
        break;
    default:
        std::cout << "invalid response!\n";
        abort();
}
```

C++ 11

デフォルトのコンストラクタ、コピーコンストラクタ、コンストラクタの、デストラクタ、コピー、またはのデフォルトのをします。

```
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;
};
```

if

`if`ステートメントをします。キーワード `if` は、またはのいずれかのでまれたをけるがあります。がである、ののサブステートメントがされます。

```
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();
}
```

else

ifステートメントのサブステートメントのろにキーワード `else` けることができ `else`。 `else` キーワードのサブステートメントは、が `false` のときつまり、のサブステートメントがされないときにされます。

```
int x;
std::cin >> x;
if (x%2 == 0) {
    std::cout << "The number is even\n";
} else {
    std::cout << "The number is odd\n";
}
```

のにあるラベルきステートメントにジャンプします。

```
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;
}
```

リターン

からびしにをします。

`return` にオペランドがある、オペランドはのりにされ、されたはびしにされます。

```
int f() {
    return 42;
}
int x = f(); // x is 42
int g() {
    return 3.14;
}
int y = g(); // y is 3
```

`return`オペランドがない、は`void`でなければなりません。なケースとして、`void -returning`は、`void`でもをすることができ`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
}
```

`main`がすると、`std::exit`はにりでびされ、そのはにされます。ただし、`main`からのローカルをしますが、`std::exit`びすことはできません。

```
int main(int argc, char** argv) {
    if (argc < 2) {
        std::cout << "Missing argument\n";
        return EXIT_FAILURE; // equivalent to: exit(EXIT_FAILURE);
    }
}
```

スロー

1. オペランドをむで`throw`がすると、そのはオペランドのコピーであるをスローすることになります。

```
void print_asterisks(int count) {
    if (count < 0) {
        throw std::invalid_argument("count cannot be negative!");
    }
    while (count--) { putchar('*'); }
}
```

2. オペランドのないで`throw`がすると、`std::terminate`がびされます。

```
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. で`throw`がすると、がされ、ができるのがリストされます。

```
// 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();
```

のはC++ 11です。

のthrowの2つの、ステートメントではなくをすることにしてください。throwのはvoidです。これにより、ののようにのににすることができます

```
unsigned int predecessor(unsigned int x) {
    return (x > 0) ? (x - 1) : (throw std::invalid_argument("0 has no predecessor"));
}
```

おしください

tryキーワードのろには、ブロック、コンストラクタのリスト、そしてブロック [ここ](#)をがきます。tryブロックのろには、1つのcatchブロックがきます。がtryブロックからした、tryブロックのにするcatchブロックは、がしたにをするがあります。

```
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
}
```

きif、if..else

ifとelse

えられたがtrueかfalseをすかどうかをチェックし、そのようにします

```
if (condition) statement
```

はとされるかをすのなC++です。

```
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
```

は、か、、、またはなどです。

```
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,
```

のをチェックするは、2つのでできます。

バイナリをって

```
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
```

if / ifelse / elseをって

なスイッチのはifまたはelse

```
if (a== "test") {
    //will execute if a is a string "test"
} else {
    // only if the first failed, will execute
}
```

の

```
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
}
```

しかし、あなたのコードがじのをチェックするは、わりに '**switch**'をうべきです

ジャンプブレーク、、ジャンプ、。

ブレーク

breakをうと、がたされなくてもループをすことができます。ループをさせたり、なわりののにさせたりすることができます

はのとおりで。

```
break;
```

たちはしばしば**break in switch case**をします。つまり、case i switchがたされた、そののコードブロックがされます。

```
switch(conditon) {
```



```
case 1: block1;
case 2: block2;
case 3: block3;
default: blockdefault;
}
```

この、ケース1がたされてからブロック1がされる、にはブロック1のみがされますが、ブロック1がりのブロックでされると、ケース1のみがされてもブロック2、ブロック3、。これをけるには、のようにブロックのにbreakをします。

```
switch(condition){
case 1: block1;
        break;
case 2: block2;
        break;
case 3: block3;
        break;
default: blockdefault;
        break;
}
```

1つのブロックのみがされ、はスイッチループかられます。

breakはif、 while、 forなどののきおよびきループでもできます。

```
if(condition1){
    ....
    if(condition2){
        .....
        break;
    }
    ...
}
```

continue

continueは、ブロックのわりにしたかのように、プログラムがのでりのループをスキップし、のりしにジャンプします。

はのとおりで。

```
continue;
```

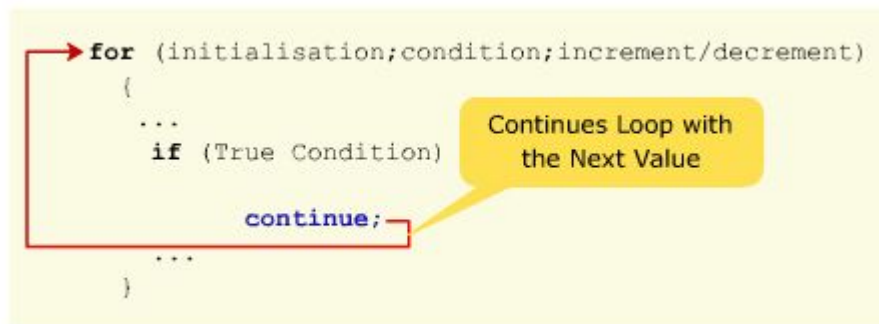
をえてみましょう

```
for(int i=0;i<10;i++){
if(i%2==0)
continue;
cout<<"\n @"<<i;
}
```

をします。

```
@1
@3
@5
@7
@9
```

このコードは、`i%2==0`がたされるたびに`continue`がされ、りのコード`print @とi`をすべてスキップし、ループのインクリメント/デクリメントがされます。



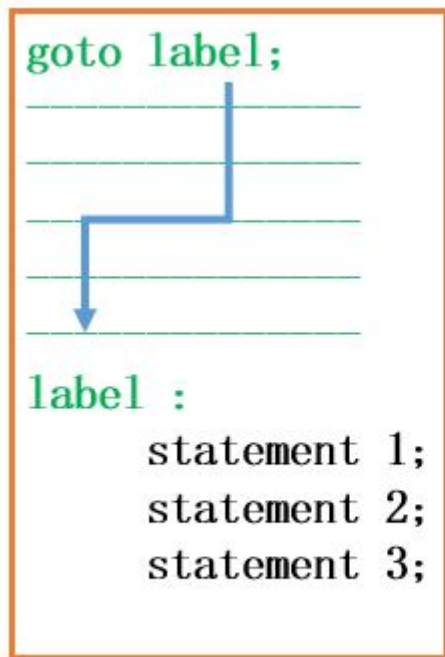
goto

それは、プログラムののポイントへのジャンプをにする。これは、ネストののタイプをしてするため、このをにするがあります。ポイントはラベルによってされ、ラベルは`goto`のとしてされます。ラベルは、なとそれにくコロン;)でされます。

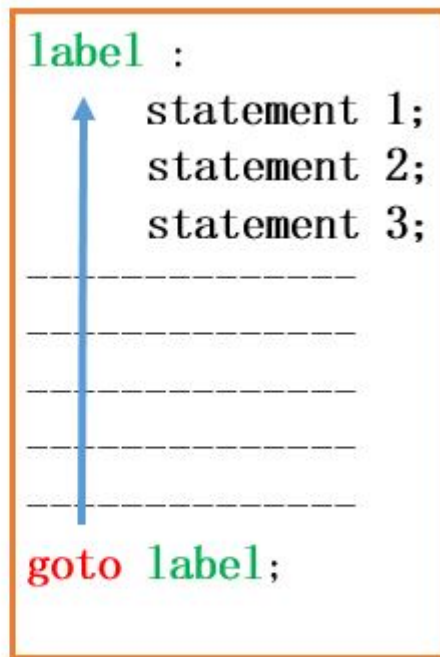
はのとおりで。

```
goto label;
..
.
label: statement;
```

`goto`ステートメントのは、プログラムのフローをトレースするのがしく、プログラムをしにくくしてするのがになるため、することをおめしません。



Forward Reference



Backward Reference

```
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 );
```

```
value of num : 1
value of num : 3
value of num : 5
value of num : 7
value of num : 9
```

`num%2==0`がたされるたびに、`goto`はを `do-while` ループのにります。

`exit`は `cstdlib` されたです。 `exit`、 のコードでのプログラムをすることです。そのプロトタイプは

```
void exit (int exit code);
```

`cstdlib`は、コード `EXIT_SUCCESS` と `EXIT_FAILURE` します。

オンラインでフローをむ <https://riptutorial.com/ja/cplusplus/topic/7837/フロー>

80: プロファイリング

Examples

gccとgprofによるプロファイリング

GNU gprofプロファイラ**gprof**をすると、コードをプロファイルすることができます。これをするには、のをするがあります。

1. プロファイリングをするためのアプリケーションをする
2. されたアプリケーションをしてプロファイリングをする
3. されたプロファイリングをgprofでする

プロファイリングをするためのアプリケーションをビルドするには、`-pg`フラグをします。したがって、たとえば、

```
$ gcc -pg *.cpp -o app
```

または

```
$ gcc -O2 -pg *.cpp -o app
```

々。

`app`というアプリケーションがビルドされたら、どおりします。

```
$ ./app
```

これにより、`gmon.out`というファイルがされ`gmon.out`。

プロファイリングをするには、すぐしてください

```
$ gprof app gmon.out
```

アプリケーションとされたのをすることにしてください。

もちろん、パイプまたはリダイレクトすることもできます

```
$ gprof app gmon.out | less
```

々。

のコマンドのは、がで、びし、、つまり、びしでプロセスへのびしをくをすでなければなりませ

ん。

gperf2dotによるコールグラフダイアグラムの

よりなアプリケーションでは、フラットなプロファイルを用いるのがしいがあります。これが、くの
プロファイリングツールがきのコールグラフのらかのをします。

[gperf2dot](#)は、くのプロファイラLinux perf、callgrind、oprofileなどからのテキストを[コールグラフダイアグラム](#)にします。プロファイラをすることでできます `gprof`

```
# compile with profiling flags
g++ *.cpp -pg
# run to generate profiling data
./main
# translate profiling data to text, create image
gprof ./main | gperf2dot -s | dot -Tpng -o output.png
```


Google Perfツールには、しいインターフェースをえたCPUプロファイラもされています。それをするには

1. Google Perfツールをインストールする
2. いつものようにコードをコンパイルする
3. にライブラリロードパスにlibprofilerプロファイラライブラリをする
4. pprofをして、フラットプロファイルまたはコールグラフダイアグラムをする

えば

```
# 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
```

ここで

- CPUPROFILEは、プロファイリングデータのファイルをします。
- CPUPROFILE_FREQUENCYは、プロファイラのサンプリングをします。

プロファイリングデータをするには、pprofをします。

フラット・コール・プロファイルをテキストとしてすることができます。

```
$ 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

```

...またはPDFでコールグラフをすることができます

```
pprof --pdf ./main main.prof > out.pdf
```

オンラインでプロファイリングをむ <https://riptutorial.com/ja/cplusplus/topic/5347/プロファイリング>

81: ヘッダーファイル

C++では、Cとに、C++コンパイラとコンパイルプロセスでCプリプロセッサがされます。GNU Cプリプロセッサマニュアルでされているように、ヘッダファイルはのようになっています

ヘッダファイルは、のソースファイルでされるCとマクロマクロををむファイルです。
。Cのプリプロセスディレクティブ '#include'をして、ヘッダファイルをプログラムにめることによって、ヘッダファイルのをします。

ヘッダーファイルには2つのがあります。

- システムヘッダファイルは、オペレーティングシステムのへのインタフェースをします。システムコールとライブラリをびすためになとをするために、これらをプログラムにめます。
- のヘッダファイルには、プログラムのソースファイルのインタフェースのがまれています。するとマクロのグループがある、そのすべてまたはがのなるソースファイルでになるたびに、そのためのヘッダーファイルをするをおめします。

ただし、Cプリプロセッサには、ヘッダファイルもソースファイルとじです。

ヘッダー/ソースファイルのスキームは、インターフェイスとのをするために、さまざまなソフトウェアプロジェクトによってされたくされたなです。

C++によってにされるわけではありませんが、ヘッダー/ソースファイルのコンベンションにうことをくおめします。には、すでにしています。

ヘッダーファイルは、のC++ええC++ 20などにまれるとえられるモジュールののによって、プロジェクトファイルのとしてきえられることにしてください。

Examples

な

のは、`// filename`コメントでされるように、いくつかのソースファイルにされるコードブロックをみます。

ソースファイル

```
// 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;
}
```

ヘッダーファイルは、ヘッダーインターフェイスでされたををするのソースファイルにまれますが、そのについてのはありませんコードをらす。のプログラムは、でしたヘッダ`my_function.h`をします。

```
// 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;
}
```

コンパイルプロセス

ヘッダーファイルはしばしばコンパイルプロセスワークフローのであるため、ヘッダー/ソースファイルのをするなコンパイルプロセスでは、のをいます。

ヘッダーファイルとソースコードファイルがすでにディレクトリにあるとすると、プログラマはこのコマンドをします。

```
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
```

あるいは、にmain.cppをオブジェクトファイルにコンパイルしてから、のステップとしてオブジェクトファイルのみをリンクするは、main.cppようにします。

```
g++ -c my_function.cpp
g++ -c main.cpp

g++ main.o my_function.o
```

ヘッダーファイルのテンプレート

テンプレートは、コンパイルにコードをするがあります。たとえば、テンプレートされたがソースコードでされるとパラメータされると、テンプレートされたはのなるににされます。

つまり、テンプレートされたコンストラクトをするコードでは、にのコードをするためにそのにするがとなるため、テンプレート、メンバ、およびクラスを々のソースコードファイルにすることはできません。

したがって、テンプレートされたコードは、ヘッダにかれていれば、そのをもんでいなければなりません。そのをにします。

```
// 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;
}
```

オンラインでヘッダーファイルをむ <https://riptutorial.com/ja/cplusplus/topic/7211/ヘッダーファイル>

82: ポインタ

き

ポインタは、メモリのアドレスです。それらは、データがされたメモリをコピーすることなく、メモリのとをにするためにされます。ポインタは、プリミティブみみまたはユーザーのことができます。

ポインタは、"dereference" *、"address of" & "and" arrow " -> をします。 '*' と ' -> ' は、しされているメモリにアクセスするためにされ、 & は、メモリのアドレスをするためにされます。

- <データ> * <>;
- <データ> * <> =<じデータの>;
- <データ> * <> =<じデータの>;
- int * foo; //をすポインタ
- int * bar =myIntVar;
- ロング*バー[2]。
- い* bar [] = {myLongVar1、 myLongVar2}; //しいlong * bar [2]

じにのポインタをするときのにしてください。

```
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

ポインタの

C ++ 11

のすべてでは、C ++ 11 `nullptr` がとなっています。のバージョンでは、 `nullptr` を `NULL` にきえてください。これは、じょうなをたしていたです。

ポインタの

ポインタは、の*をしてすることができます `int *pointer_to_int;`。
がポインタ `int *` のときは、メモリアドレスだけがされます。メモリアドレスは、になる `int` のデータがされるです。

いは、のサイズをじへのポインタのサイズとするとかです。

```
// 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
*/
```

のアドレスをする

ポインタは、のと同じようにおいにりてることができます。この、ポインタがすのデータではなく、あるポインタからのポインタにコピーされるのはメモリアドレスです。さらに、ヌルメモリーをす `nullptr` をすることができます。 `nullptr` にしいポインタになメモリアドレスがまれているため、なデータをしていません。

えられたののメモリアドレスは、`&` アドレスにをけることによってすることができます。 `&` によってされるは、のメモリアドレスをむになるへのポインタです がスコープかられないりなデータです。

```
// 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
```

とはに

- 2つのポインタをりてても、りてられたポインタがするメモリはきされません。
- ポインタはnullでもかまいません。
- オペレータのアドレスはにされます。

ポインタへのアクセス

アドレスには&があるので、コンテンツにアクセスするにはとして*するがあります。ポインタがされると、それはのになりますにはその。その、constなくてもそれをみてすることができます。

```
(*p_bar0).fool = 5;

// `p_bar0` points to `bar`. This prints 5.
std::cout << "bar.fool = " << bar.fool << 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.fool = " << baz.fool << std::endl;
```

*とのみわせ. ->とされ->

```
std::cout << "bar.fool = " << (*p_bar0).fool << std::endl; // Prints 5
std::cout << "bar.fool = " << p_bar0->fool << std::endl; // Prints 5
```

なポインタをする

ポインタをするときは、なデータをしていることをするがあります。なポインタまたはヌルポインタをすると、メモリアクセスがしたり、ガベージデータをみきしたりするがあります。

```
big_struct *never_do_this() {
    // This is a local variable. Outside `never_do_this` it doesn't exist.
    big_struct retval;
    retval.fool = 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).
}
```

このようなシナリオでは、g++とclang++しくをします。

```
(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]
```

したがって、ポインタなのである、ポインタはnullになるがあるのであります。

```
void naive_code(big_struct *ptr_big_struct) {
    // ... some code which doesn't check if `ptr_big_struct` is valid.
    ptr_big_struct->fool = 12;
}

// Segmentation fault.
naive_code(nullptr);
```

ポインタ

ポインタには2つのがあります。アドレスのメモリアドレスをします。のDereference*でされたアドレスにあるのをします。

```
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)
```

アスタリスク*は、それがポインタであることをすなのためにポインタをするにされます。これをされたアドレスにあるをするためにされとしないでください。それらはじでされたに2つのなるものです。

ポインタ

インクリメント/デクリメント

ポインタはインクリメントまたはデクリメントできますと。ポインターをインクリメントすると、ポインティングされているを1つるのにポインタがめられます。ポインタをさせると、ののにします。

ポインターがすがでない、ポインターはされません。 voidはになです。

```
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.
```

わりのへのポインタがインクリメントされると、ポインタはのをぎた1つのをしします。そのようなポインタはすることはできませんが、デクリメントすることはできます。

の1つのわりのへのポインタをインクリメントするか、またはのへのポインタをらすと、のになります。

オブジェクトへのポインタは、ポインタのために、サイズ1のであるかのようにうことができます。

/

をポインタにすることができます。これらはとしてしますが、1ではなくのをしします。はポインタからすることもでき、ポインタのとしてします。インクリメント/デクリメントとに、ポインタはなをしていなければなりません。

```
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.
```

ポインタの

じへの2つのポインタのをすることができます。2つのポインタはじオブジェクトになければなりません。さもなければのがこる。

じに2つのポインタ p と q がある、 p がの i のであり、 q がの j のである、 $p - q$ は $i - j$ ます。のタイプは`<cstdlib> std::ptrdiff_t` `<cstdlib> std::ptrdiff_t`です。

```
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.
```

オンラインでポインタをむ <https://riptutorial.com/ja/cplusplus/topic/3056/ポインタ>

83: ミューテックス

std :: shared_timed_mutexよりも***std :: shared_mutex***をうほうがいいでしょう。

パフォーマンスのは2です。

RWLockをするは、2つのオプションがあります。

これはstd :: shared_mutexとshared_timed_mutexです。

std :: shared_timed_mutexはちょうどバージョンの 'std :: shared_mutex + time method' だとうか
もしれません。

しかし、はくなります。

のコードは、std :: shared_mutexの**MSVC14.1**です。

```
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));
}

```

std :: shared_mutexが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)にされていることがわかります。

に、std :: shared_timed_mutexのをてみましょう。

のコードは、std :: shared_timed_mutexの**MSVC14.1**です。

```

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;
};

```

std :: shared_timed_mutexがstd :: condition_variableでされていることがわかります。

これはきないです。

そこで、2つのパフォーマンスをしましょう。

```

STLSharedMutex READ :      486647
STLSharedMutex WRITE :      205986
TOTAL READ&WRITE :      692633

STLSharedTimedMutex READ :      140291
STLSharedTimedMutex WRITE :      178849
TOTAL READ&WRITE :      319140

```

これは、1000ミリのみし/きみテストののです。

std :: shared_mutexは**std :: shared_timed_mutex**よりも**2**のみきをしました。

このでは、みきはじですが、みみはのきみよりもになります。
したがって、パフォーマンスのがきくなるがあります。

のコードはこののコードです。

```

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`

`try`ロック、`timed try`ロック、および`recursive lock`のRAIIスタイルにされます。

`std::unique_lock`は`mutex`のをにします。

`std::shared_lock`は、`mutex`のをにします。いくつかのスレッドができる `std::shared_locks`
`std::shared_mutex`。 C ++から14。

`std::lock_guard`は、 `std::unique_lock`と`std::shared_lock`わるの `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;
};

```

ロッククラスのストラテジ **std::try_to_lock**、**std::adopt_lock**、**std::defer_lock**

std::unique_lockのをするは、からするにはつのであるロックがあります **std::try_to_lock**、**std::defer_lock**と**std::adopt_lock**

1. **std::try_to_lock**はブロックせずにロックを**std::try_to_lock**ことができます

```

{
    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**すると、ロックをせずにロックをできます。のミューテックスをロックする、2のびしがにロックをしようとする、デッドロックがするがあります。

```

{
    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;
    //...
}

```

のコードでは、でがきても、ロックはされ、なでされます。

```
{
    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. びしのスレッドがロックをしている、 `std::adopt_lock` は2のロックをみません。

```
{
    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;
    //...
}
```

にめておくべきことは、`std::ado_lock` はな `mutex` のにわるものではないということです。ロックがになると、`mutex` はされます。

`std::mutex`

`std::mutex` は、のスレッドがアクセスするデータをするためにされるシンプルでなです。

```
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_lockC ++ 17`

`std::scoped_lock` は、 `std::lock` されるロックアルゴリズムとみわけて、もう1つの `mutex` をするた

めのRAIIスタイルのセマンティクスをし `std::lock`。 `std::scoped_lock` がされると、`mutex`はしたと
はのでされます。

```
{
    std::scoped_lock lock{_mutex1,_mutex2};
    //do something
}
```

ミューテックスの

C++ 1xでは、`mutex`クラスをできます。

- `std::mutex` - なロックをします。
- `std::timed_mutex` - `try_to_lock`をする
- `std::recursive_mutex` - じスレッドによるなロックをにします。
- `std::shared_mutex`、`std::shared_timed_mutex` - されたユニークなロックをします。

`std::lock`

`std::lock`は、デッドロックアルゴリズムをして1つの`mutex`をロックします。のオブジェクトをロ
ックするびしにがスローされた、`std::lock`はをスローするにロックされたオブジェクトの
`std::lock`をします。

```
std::lock(_mutex1, _mutex2);
```

オンラインでミューテックスをむ <https://riptutorial.com/ja/cplusplus/topic/9895/ミューテックス>

84: メタプログラミング

き

C++でのメタプログラミングとは、マクロやテンプレートをしてコンパイルにコードをすることをします。

に、このロールではマクロがになり、テンプレートはではありませんが、テンプレートがされます。

テンプレートのメタプログラミングは、テンプレートや`constexpr`をしてコンパイルのをしてコードをするというをすることがよくありますが、コンパイルのはメタプログラミングではありません。

メタプログラミングよりには、テンプレートメタプログラミングは、コンパイルにテンプレートをして、またはデータをします。これにより、ではなくコンパイルに1をすることができます。

Examples

ファクタリゼーションの

`Factorial`は、テンプレートメタプログラミングをしてコンパイルにすることができます。

```
#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`はですが、テンプレートメタプログラミングではテンプレートメタとしてわれます。に、テンプレートのメタは、のメンバをチェックすることによってされます。 `::type`をもたらすメ

タを、`type`は`::value`をするメタをします。

のコードでは、すがあるパラメータでテンプレートをインスタンスし、`::value`をしてのをすることによって、`factorial`メタをします。

メタは、じメタをさなでにインスタンスすることになっています。`factorial<0>`は、をす。テンプレートメタプログラミングはプログラミングのをとっているのはな「ループ」です。

テンプレートメタはコンパイルにされるので、そのはコンパイルのをとするコンテキストでできます。えは

```
int my_array[factorial<5>::value];
```

には、コンパイルにされたサイズがです。メタのはコンパイルなので、ここです。

ほとんどのコンパイラでは、のさがえてはいけません。たとえば、デフォルトで`g++`コンパイラは256レベルにをします。`g++`、プログラマは`-ftemplate-depth-X`オプションをしてをします。

C++ 11

C++ 11、こののテンプレートには`std::integral_constant`テンプレートをします。

```
#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"
}
```

さらに、`constexpr`はよりクリーンになります。

```
#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';
}
```

`factorial()`のはのステートメントとしてされています。これは、C++ 11の`constexpr`では、のかなりされたサブセットしかできないためです。

C++ 14

C++ 14、`constexpr`のくのが`constexpr`、よりにけるようになりました。

```
constexpr long long factorial(long long n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

あるいは

```
constexpr long long factorial(int n)
{
    long long result = 1;
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

C++ 17

C++ 17はfold expressionを使ってをすることができます

```
#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;
}
```

パラメータパックの

くの、バリデーションテンプレートのパラメータパックのすべてののにしてをすることができます。これをうにはくがあり、ソリューションはC++でのみきがになります。パックのすべてののをにしたいとします。もなは、することです。

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...);
}

```

わりに、エキスパンダートリックをして、すべてのストリーミングをのどすることができます。これには2のをとしないというがありますが、みさにるというがあります。

C ++ 11

```

template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}

```

このみのについては、[TCのれたえ](#)をご覧ください。

C ++ 17

C ++ 17では、このをするために、アーセナルに2つの新しいツールがされています。1つはfold-expressionです。

```

template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    ((os << args), ...);
}

```

if constexprはif constexpr、のをつのでくことができます

```

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...);
    }
}

```

std :: integer_sequenceでりす

C ++ 14、ではクラステンプレート


```
template <class T, T... Ints>
class integer_sequence;

template <std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

そのためのメタをします。

```
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>;
```

これはC++ 14ではですが、これはC++ 11ツールをしてできます。

このツールをして、`std::tuple`のをつをびすことができます `std::apply`としてC++ 17でされてい
`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<Tuple>(tpl),
        std::make_index_sequence<std::tuple_size<std::decay_t<Tuple>>::value>{});
}

// this will print 3
int f(int, char, double);

auto some_args = std::make_tuple(42, 'x', 3.14);
int r = apply(f, some_args); // calls f(42, 'x', 3.14)
```

タグディスパッチ

コンパイルにをにするは、タグを1つはのとしてするオーバーロードされたのペアにをディスパッチすることです。たとえば、`std::advance()`をするには、`iterator`カテゴリにディスパッチできます。

```
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{} );
}

```

オーバーロードされた `details::advance` の `std::XY_iterator_tag` は、のパラメータです。のはではありません。そののは、コンパイラがタグクラスの `details::advance` がびされるときについて、オーバーロードをできるようにすることです。

ここでは、`advance` する `iterator_traits<T>::iterator_category` メタのいずれかをした `iterator_tag` のタイプにして、クラスを `Iter`。 `iterator_category<Iter>::type` のデフォルトでされたオブジェクトは、コンパイラに `details::advance` なるオーバーロードの1つをさ `details::advance` ます。こののパラメータはの `struct` デフォルトでされたオブジェクトであり、してされないため、にされているがあります。

タグのディスパッチは、`SFINAE` と `enable_if` をしてのコードよりもはるかにみやすいコードをえることができます。

C++ 17では、に `if constexpr` が `advance` のをすることができますが、タグのディスパッチとはなり、オープンなにはしていません。

がかどうかをする

あるタイプにしてオペレータまたはをびすことができるかどうかをすることができます。クラスに `std::hash` オーバーロードがあるかどうかをテストするには、のようになります。

```

#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

C++ 17、`std::void_t`はこのタイプのをするためにできます

```
#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
{};
```

ここで、`std::void_t`はのようにされます。

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

`operator<`などの`operator<`がされているかどうかをするには、はほぼじです。

```
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
{};
```

`T`が`std::hash`オーバーロードをっている、これらは`std::unordered_map<T>`をするためにできますが、`それのは``std::map<T>`をしようします。

```
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>>;
```

C++ 11での

C++では、コンパイルに11のがはるかにになります。たとえば、コンパイルにえられたのをすると、のようになります。

```
template <typename T>
constexpr T calculatePower(T value, unsigned power) {
    return power == 0 ? 1 : value * calculatePower(value, power-1);
}
```

キーワード `constexpr` は、コンパイルにをします。そのためにながすべてたされている `constexpr` キーワード、コンパイルにすべてのをっているがあります。

C++ 11 `constexpr` は1つの `return` からのみするがあります。

これをコンパイルのなとすると、このメソッドはのにもちます。つまり、のがコンパイルにられていないえば、とパワーがユーザーからのとしてえられた、はコンパイルにされるため、コードをすることはできませんC++のいではされます。

えば

```
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

コンパイルにをするもう1つの、のように `fold` をできます。

```
#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;
}
```

のがえられたときののによる **T**

`std::enable_if` `enable_if` をして **SFINAE** をする、ヘルパーテンプレートにアクセスして、の **T** がのにするかどうかをすることがよくあります。

これをけするために、では、 `true` と `false` 2つのタイプのアナログ `std::true_type` と `std::false_type` がすでにされています。

のは、タイプ T がポインタであるかどうかをするをしています `is_pointer` テンプレートは、の `std::is_pointer` ヘルパーのをしています。

```
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> { }
```

のコードには3つのステップがあります2つだけなもあります。

1. `is_pointer_` ののがデフォルトので、 `std::false_type` からします。デフォルトのケースは、`"false condition"` にしているので、に `std::false_type` からするがあります。
2. 2のは、 T がにであるかをにせずにポインタ `T*` `is_pointer_` テンプレートをします。このバージョンは `std::true_type` からしています。
3. 3ののものは、 T からなをするだけですこのは `const` と `volatile` がされます。その、の2つののいずれかにります。

`is_pointer<T>` はクラスなので、そのにアクセスするにはのいずれかがです。

- `::value` する `::value is_pointer<int>::value - value` は、 `std::true_type` または `std::false_type` からした `bool` のクラスメンバーです。
- `is_pointer<int>{}` - これは、 `std::is_pointer` が `std::true_type` または `std::false_type` `constexpr` コンストラクタをつと `std::true_type` と `std::false_type` からデフォルトのコンストラクタをするため、 `std::false_type constexpr` を `bool` っています。

にアクセスできるようにする「ヘルパーヘルパーテンプレート」をすることはいす

```
template <typename T>
constexpr bool is_pointer_v = is_pointer<T>::value;
```

C++ 17

C++ 17では、ほとんどのヘルパーテンプレートはすでに `_v` バージョンをしています。

```
template< class T > constexpr bool is_pointer_v = is_pointer<T>::value;
template< class T > constexpr bool is_reference_v = is_reference<T>::value;
```

If-then-else

C++ 11

ライブラリヘッダー `<type_traits>` のタイプ `std::conditional` は、コンパイルのboolについて、1つのタイプまたはのタイプをできます。

```
template<typename T>
struct ValueOrPointer
{
    typename std::conditional<(sizeof(T) > sizeof(void*)), T*, T>::type vop;
};
```

これは、へのポインタをむ_{TT}ポインタのきさ、またはよりきい_Tがポインタのきさである、それ。したがって、`sizeof(ValueOrPointer)`はに`<= sizeof(void*)`ます。

をつな

C++ 11

テンプレートのメタプログラミングによってさまざまなものをけれるたとえば`min`をくことができます。これは、2つのにして`min`をし、よりくのはにします。

```
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);
```

オンラインでメタプログラミングをむ <https://riptutorial.com/ja/cplusplus/topic/462/メタプログラミング>

85: メモリ

- `:( OPT リスト OPT -ID、 たな OPT`
- `opt new リスト opt タイプID new-initializer opt`
- `:( opt キャストをする`
- `:( opt delete [] キャスト`
- `std :: unique_ptr < type-id > var_name しいタイプ -id opt ; // C ++ 11`
- `std :: shared_ptr < タイプID > var_name しいタイプ -id opt ; // C ++ 11`
- `std :: shared_ptr < タイプID > var_name = std :: make_shared < タイプID > opt ; // C ++ 11`
- `std :: unique_ptr < type-id > var_name = std :: make_unique < タイプID > opt ; // C ++ 14`

leading `::` は、オーバーロードされたクラスのまたはをオーバーライドして、`new` または `delete` をグローバルスコープでさせます。

`new` キーワードのオプションのは、`new` をびすためにされますが、されたプールからメモリをりてることをするタグなど、をアロケータにすためにもできます。

りてられたは、`new Foo` ににされますが、`auto C ++ 11` または `decltype(auto) C ++ 14` としてからすることもできます。

りてられたオブジェクトのは、ローカルのとじによってわれます。に、イニシャライザがされた、オブジェクトはデフォルトでされ、スカラーまたはスカラーのをにりてる、メモリがゼロになるというはありません。

`new-expression` でされたオブジェクトは、`new-expression` が `[]` かれているかどうかにかかわらず、`delete []` をって `delete []` があります。えば

```
using IA = int[4];
int* pIA = new IA;
delete[] pIA; // right
// delete pIA; // wrong
```

Examples

スタック

スタックは、にながされるさなメモリです。スタックにデータをりてることは、すべてのメモリがすでにこののためにりてられているため、ヒーブリてにべてにです。

```
int main() {
    int a = 0; //Stored on the stack
    return a;
}
```

スタックのは、びしのがメモリの々のさなセクションをしてそれぞれのに「スタック」されたな

メモリをつためにかけられます。

```
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();
}
```

スタックにされたデータは、をりてたスコープがまだアクティブであるりです。

```
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;
}
```

フリーストレージヒープ、りて...

「ヒープ」は、スタックがするメモリとはして、をりりおよびりりできるメモリをするなです。

C++、このは、よりなとなされるフリーストアとしてされています。

フリーストアからりてられたメモリは、りてられたのスコープよりくすることがあります。スタックにするにはきすぎるデータは、**Free Store**からりてられることもあります。

メモリは、しいキーワードとするキーワードによってりてられたり、りてされたりすることができます。

```
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
```

サイズのを *new* と *delete* でりることもで、もしなります。のりてはのりてとがなく、2つをさせるとヒープのにつながります。をりてことは、でのですのために、のサイズをするためにメモリをりてます。

```
// Allocates memory for an array of 256 ints
int *foo = new int[256];
// Deletes an array of 256 ints at foo
delete[] foo;
```

new をし、*malloc* と *free* のわりに *delete* をする と、コンストラクタとデストラクタがされますスタックベースのオブジェクトと。これは、*malloc* と *free* よりもしい、*delete* をむです。

```
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;
```

C ++ 11

C ++ 11では、をすために *スマートポインタ* をすることをおめします。

C ++ 14

C ++ 14では、*std::make_unique* をSTLにし、ネイキッド *new* と *delete* をするのではなく、*std::make_unique* または *std::make_shared* するようにしました。

プレースメントしい

メモリをりてるためにFree Storeにしたいくないがあり、*new* メモリをしてカスタムメモリりてをしたいがあります。

このようなでは、*Placement New* をすることができます。ここでは、「*new*」に、にりてられたメモリからメモリをりてるようできます

えば

```
int a4byteInteger;

char *a4byteChar = new (&a4byteInteger) char[4];
```

ここでは、`a4byteChar`がメモリは、`a4byteInteger`して 'stack'にりてられた4バイト `a4byteInteger`。

このメモリりてのは、プログラマがりてをするということです。のでは、`a4byteInteger`はスタックにりてられているので、`'delete a4byteChar'`をにびすはありません。

ダイナミックにりてられたメモリにしてものがられます。えば

```
int *a8byteDynamicInteger = new int[2];

char *a8byteChar = new (a8byteDynamicInteger) char[8];
```

この、によってメモリ・ポインタ `a8byteChar`によってりてられたメモリをする `a8byteDynamicInteger`。しかし、この、`delete a8byteDynamicInteger`をにびしてメモリを `delete a8byteDynamicInteger`があります

C++クラスのの

```
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;
}
```

オンラインでメモリをむ <https://riptutorial.com/ja/cplusplus/topic/2873/メモリ>

86: メンバーへのポインタ

- Classというクラスをします...
 - タイプ `* ptr = Class :: member; //メンバーのみをす`
 - クラス `:: * ptr = Class :: member; //クラスメンバをす`
- クラスメンバへのポインタについては、の2つのがえられます。
 - クラスインスタンス。
 - クラス `* p = インスタンス;`
- クラスメンバへのポインタ
 - `ptr = Class :: i; //すべてのクラスでiをポイントする`
 - インスタンス。 `* ptr = 1; //インスタンスのiにアクセスする`
 - `p -> * ptr = 1; // pにアクセスする`
- クラスメンバへのポインタ
 - `ptr = Class :: F; //すべてのクラスで 'F'をポイント`
 - インスタンス。 `* ptr5; //インスタンスのFをびします`
 - `p->ptr6; // pのFをびします。`

Examples

メンバーへのポインター

`static`メンバーはスコープをいてのC/C++とまったくじです

- `class`にあるので、`class`でられたがです。
- それは、`public`、`protected`または`private`なアクセスをとっています。

したがって、`static`メンバーにアクセスしてそれをしくすると、`class`ののののようにをすことができます

```
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()

```

メンバへのポインタ

クラスのメンバにアクセスするには、インスタンス、またはそのインスタンスへのポインタまたはとして、のインスタンスに「ハンドル」をたせるがあります。クラスインスタンスがえられた、しいをした、メンバーへのポインタをしてメンバーのさまざまなメンバーをすことができます。もちろん、ポインターはあなたがしているものとじであるとされなければなりません...

```

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()

```

メンバへのポインタとはなりののでは、クラスインスタンスとメンバポインタののけは、でられているがあり、`.*`や`->*`がではないかのような

メンバへのポインタ

`class`メンバーにアクセスするには、インスタンス、またはそのインスタンスへのポインターまたはとして、のインスタンスへの「ハンドル」をたせるがあります。 `class`インスタンスがえられた、しいをした、メンバーへのポインタをしてメンバーのさまざまなメンバーをすことができます。もちろん、ポインターはあなたがしているものとじであるとされなければなりません...

```

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()

```

pointer-to-memberのには、いくつかのながです。

- ポインタのをするには、とそれがクラスのにあるというにするがあります `int Class::*ptr;`。
- クラスやをとって、それをポインタへのポインタとともにしたいは、`.*` にてい、をするがあります。
- クラスへのポインタをち、それをポインタへのポインタとともにするは、`->*` にていますをするがあります。

メンバーへのポインタ

`static`メンバは、スコープをいて、のC/C++とまったくじです。

- `class`にあるので、`class`でられたがです。
- それは、`public`、`protected`または`private`なアクセスをとっています。

したがって、`static`メンバーにアクセスしてしくすると、`class`のののようになすことができます。

```

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()
```

オンラインでメンバーへのポインタをむ <https://riptutorial.com/ja/cplusplus/topic/2130/メンバーへのポインタ>

87: ユーザーのリテラル

Examples

い **double** をつ ユーザ リテラル

```
#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;
}
```

このプログラムのはのとおりで。

```
3 km = 3000 m
3 mi = 4828.03 m
```

ののユーザ リテラル

C++ 14

くユーザリテラルは、 `std::literals::chrono_literals` という namespace されます。 `literals` と `chrono_literals` はとも **インラインネームスペース** です。これらのへのアクセスはしてることができる `using namespace std::literals`、 `using namespace std::chrono_literals`、 および `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
}

```

のなユーザーリテラル

C++ 14

のユーザーリテラルは、`literals`と`string_literals`がインラインであるnamespace
`std::literals::string_literals`でされます。これらのへのアクセスはしてることができるusing
`namespace std::literals`、`using namespace std::string_literals`、および`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;
}

```

リテラルに\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

```

のなユーザーリテラル

C++ 14

なユーザーリテラルは、`literals`と`complex_literals` `literals`がインラインネームスペースである

namespace std::literals::complex_literals でされます。これらのへのアクセスはしてることができる
using namespace std::literals、 using namespace std::complex_literals、 および 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
}
```

バイナリのユーザのリテラル

あなたにもかかわらず、C++ 14でバイナリをくことができます

```
int number = 0b0001'0101; // ==21
```

ここでは、バイナリのためののがあります

テンプレートプログラムがコンパイルにされています。

```
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
}
```

オンラインでユーザのリテラルをむ <https://riptutorial.com/ja/cplusplus/topic/2745/ユーザのリ>

テラル

88: ラムダ

- `[ default-capture 、 capture-list ]` リスト なスロー `-> return-type { lambda-body }` //ラムダとの。
- `[ キャプチャリスト ]` リスト `{ lambda-body }` //ラムダ。
- `[=]` リスト `{ lambda-body }` //なすべてのローカルをでします。
- `[]` リスト `{ lambda-body }` //なすべてのローカルをによってします。
- `[ capture-list ] { lambda-body }` //リストとをすることができます。

パラメーター

パラメータ	
デフォルトキャプチャ	リストされていないすべてののキャプチャをします。=によってキャプチャまたは、によってキャプチャ。した、リストされていないはラムダでアクセスできなくなります。デフォルトキャプチャは、キャプチャリストにするがあります。
キャプチャリスト	ラムダでローカルにアクセスするをします。のなはでりまれます。&がいたはによってされます。クラスメソッドでは、thisをして、すべてのメンバーをにすることができます。リストにデフォルトキャプチャがされていないり、リストされていないにはアクセスできません。
リスト	ラムダのをします。
な	オプション、によってりまれるはconstです。mutableをすると、constになります。これらのにするは、びしでされます。
スロー	オプションラムダのスローイングをします。たとえば、noexceptまたはthrow(std::exception)です。
	オプションラムダの。たとえば、lambda-bodyがにをスローした、[[noreturn]]をできます。
-> り	オプションラムダのりのをします。りのをコンパイラができないにです。
ラムダボディ	ラムダのをむコードブロック。

C++ 17のドラフトでは、にコンパイルにできるlambdaというconstexpr lambdaがされています。lambdaはconstexprをたしていればconstexprですが、constexprキーワードをしてすることもできます。

```
//Explicitly define this lambdas as constexpr
[]() constexpr {
    //Do stuff
}
```

Examples

ラムダとはですか

ラムダはなオブジェクトをするなをします。ラムダは、そのオブジェクトがオブジェクトのようになるうクロージャオブジェクトとばれるである。

「ラムダ」はラムダからまります。ラムダは、とについてのをべるために、1930にアロンゾによってされたです。ラムダは、プログラミングLISPのをしました。ラムダとLISPとして、C++のラムダはのなプロパティをし、のコンテキストからをしますが、をしてをすがありません。

ラムダは、びしなオブジェクトをるのとしてよくわれま。これは、としてされたときにのみされるきをするよりもになります。そのような、オブジェクトをインラインでできるので、ラムダがにまれます。

ラムダは、キャプチャリスト[]、オプションのパラメータリスト()、およびボディ{}の3つのでされ、これらはすべてにすることができます。

```
[](){} // An empty lambda, which does and returns nothing
```

キャプチャリスト

[]はキャプチャリストです。デフォルトでは、みスコープのにラムダでアクセスすることはできません。をキャプチャすると、ラムダでコピーまたはとしてアクセスできるようになります。キャプチャされたはラムダのになります。のとはに、ラムダをびすときにすはありません。

```
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.
```

パラメータリスト

()はパラメータリストで、のとはぼじです。ラムダがをらないは、これらのカッコをすることができますラムダmutableをするがあるをく。これらの2つのラムダはです

```
auto call_foo = [x]() { x.foo(); };
auto call_foo2 = [x]{ x.foo(); };
```

C++ 14

パラメータリストでは、ののわりにプレースホルダ`auto`できます。そうすることで、これはテンプレートのテンプレートパラメータのようにします。ラムダにくのは、ジェネリックコードでベクトルをソートするにはです

```
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; };
```

`{}`はで、のとじです。

ラムダのびし

ラムダのオブジェクトはクロージャであり、`operator()`をしてびすことができますのオブジェクトと。

```
int multiplier = 5;
auto timesFive = [multiplier](int a) { return a * multiplier; };
std::out << timesFive(2); // Prints 10

multiplier = 15;
std::out << timesFive(2); // Still prints 2*5 == 10
```

りの

デフォルトでは、ラムダのりのがされます。

```
[](){ return true; };
```

この、り`bool`は`bool`です。

のをして、りのをですることもできます。

```
[]() -> bool { return true; };
```

なラムダ

ラムダのによってされたオブジェクトは、デフォルトではです。これは、されたクロージャオブジェクトの`operator()`がデフォルトで`const`であるためです。

```
auto func = [c = 0]() { ++c; std::cout << c; }; // fails to compile because ++c
// tries to mutate the state of
// the lambda.
```

`mutable`というキーワードをすると、よりいオブジェクトの`operator()` `const`することができます。

```
auto func = [c = 0]() mutable { ++c; std::cout << c; };
```

りのとともにすると、そのにmutableがます。

```
auto func = [c = 0]() mutable -> int {++c; std::cout << c; return c;};
```

ラムダのをす

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));
```

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; });
```

りのの

のreturnをつラムダ、またはじのをつのreturnの、コンパイラはりのをできます。

```
// Returns bool, because "value > 10" is a comparison which yields a Boolean result
auto l = [](int value) {
    return value > 10;
};
```

なるののreturnをつlambdaの、コンパイラはりをできません

```
// error: return types must match if lambda has unspecified return type
auto l = [](int value) {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

この、りのをにするがあります。

```
// The return type is specified explicitly as 'double'
auto l = [](int value) -> double {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

このルールは、`auto`タイプのルールとします。にされたりのをたないLambdaはをすことはない
ので、をむはにするがあります

```
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
```

でりむ

キャプチャリストにのをすると、ラムダはでキャプチャします。これは、ラムダにされたクロー
ジャタイプがのコピーをすることをします。これはまた、のをコピーコンストラクタブルにする
があります。

```
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();
};
```

C++ 14では、そのでをすることができます。これにより、のみのタイプをラムダにりむことがで
きます。

C++ 14

```
auto p = std::make_unique<T>(...);
```

```
[p = std::move(p)]() {
    return p->createWidget();
};
```

ラムダはでするときにをでりみますが、デフォルトではそのをラムダすることはできません。これは、closureがラムダを`operator() const`にれるためです。

`const`は、クロージャのメンバへのアクセスにされ、クロージャのメンバであるキャプチャされたのすべてののにされます。

```
int a = 0;

[a]() {
    a = 2;          // Illegal, 'a' is accessed via `const`

    decltype(a) a1 = 1;
    a1 = 2; // valid: variable 'a1' is not const
};
```

`const`をするには、ラムダにキーワード`mutable`をするがあります

```
int a = 0;

[a]() mutable {
    a = 2;          // OK, 'a' can be modified
    return a;
};
```

`a`はによってされたため`a`ラムダをびすことによってわれたは`aa`ません。`a`のはにラムダにコピーされたので、ラムダの`a`のコピーは`a`とはです。

```
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"
```

されたキャプチャ

C++ 14

Lambdaはだけでなく、をりむことができます。これにより、lambdaはのをできます。

```
auto p = std::make_unique<T>(...);

auto lamb = [p = std::move(p)]() //Overrides capture-by-value of `p`.
{
    p->SomeFunc();
};
```


これは、`lamb`ラムダキャプチャにを、`lamb`ともばれ`p`。 `lamb`make_uniqueによってりてられたメモリをしています。クロージャリーにはコピーながまれているので、`lamb`はコピーできません。しかし、それはすることができます

```
auto lamb_copy = lamb; //Illegal
auto lamb_move = std::move(lamb); //legal.
```

`lamb_move`はメモリをしています。

`std::function<>`は、されたをコピーにするがあることにしてください。の[move-only-std::function](#)とする`std::function`をくこともできますし、ラムダを`shared_ptr`ラッパーにれるだけでもかまいません。

```
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));
```

ののみのラムダをり、コピーされ、そのにすることができるラムダり、ポインタにそのをめむ`std::function`または。

されたキャプチャは、のにして`auto`をします。デフォルトでは、これらのキャプチャはとしてされますが、としてもできます。

```
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.
```

キャプチャでは、をキャプチャするはありません。のをりむことができます

```
auto lamb = [p = std::make_unique<T>(...)]()
{
    p->SomeFunc();
}
```

これは、`lambda`ににすることなく、`lambda`にしてなのをえるのにです。もちろん、ラムダがをしたにこれらのにアクセスするつもりがないにのみです。

リファレンスによるキャプチャ

ローカルののに`&`つけると、はによってりまれます。には、これは、ラムダのクロージャタイプが、ラムダのスコープのからのするへのとしてされたをつことをします。ラムダボディでをすると、のがされます。

```
// Declare variable 'a'
int a = 0;

// Declare a lambda which captures 'a' by reference
auto set = [&a]() {
    a = 1;
};

set();
assert(a == 1);
```

`mutable`というキーワードはではありません。なぜなら`a`は`const`はないからです。

もちろん、によってすることは、ラムダがキャプチャするのをエスケープしてはならないことをします。をびすをびすことはできますが、のをえてラムダをするをびさないでください。そしてラムダをさないでください。

デフォルトキャプチャ

デフォルトでは、キャプチャリストににされていないローカルには、ラムダからアクセスすることはできません。しかし、ラムダボディによってされたをにりむことはです

```
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
```

なキャプチャはなデフォルトのキャプチャとしてできます。なキャプチャは、デフォルトのキャプチャよりもされます。

```
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
};
```

なラムダ

C++ 14

ラムダは、ののをることができます。これによりラムダをよりにすることができます

```
auto twice = [](auto x){ return x+x; };

int i = twice(2); // i == 4
std::string s = twice("hello"); // s == "hellohello"
```

これは、クローザーの`operator()`テンプレートにオーバーロードさせることによってC++でされています。のは、ラムダクロージャとじをします。

```
struct _unique_lambda_type
{
    template<typename T>
    auto operator() (T x) const {return x + x;}
};
```

ジェネリックラムダのすべてのパラメータがであるはありません。

```
[](auto x, int y) {return x + y;}
```

ここで、`x`はののについてかれ、`y`はに`int`ます。

なラムダは、`auto`と`&`のをして、によってもをすることができます。ジェネリックパラメータが`auto&&`とられた、これはされたへのであり、**rvalue**ではありません。

```
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&`.
```

ラムダはであり、をにすることができます

```
auto lam = [](auto&&... args){return f(std::forward<decltype(args)>(args)...);};
```

または

```
auto lam = [](auto&&... args){return f(decltype(args)(args)...);};
```

`auto&&`タイプので「しく」します。

ジェネリック`lambda`をするいは、をするためです。

```
boost::variant<int, double> value;
apply_visitor(value, [&](auto&& e){
    std::cout << e;
});
```

ここでは、なでしています。のでは、しているのはくない

```
mutex_wrapped<std::ostream&> os = std::cout;
```

```
os.write([&](auto&& os){
    os << "hello world\n";
});
```

`std::ostream` タイプをりすところにノイズがあります。あなたがそれをうたびにのをわなければならないようなものです。ここではビジターをしています、なものはしていません。 `auto` あなたがするがありますのでされている `auto` で `for(;;)` ループ。

ポインタへの

ラムダのキャプチャリストが、ラムダは、じをとり、じりのをすポインタへのなをいます。

```
auto sorter = [](int lhs, int rhs) -> bool {return lhs < rhs;};

using func_ptr = bool (*)(int, int);
func_ptr sorter_func = sorter; // implicit conversion
```

このようなは、プラスをしてすることもできます。

```
func_ptr sorter_func2 = +sorter; // enforce implicit conversion
```

このポインタをびすと、ラムダで `operator()` をびすのとまったくじようにします。このポインタは、ソースラムダクロージャのにしてしません。したがって、ラムダよりもくするがあります。

このはに、C++オブジェクトではなくポインタをうAPIで `lambda` をするにです。

C++ 14

ポインタへのは、のキャプチャリストをつラムダにしてもです。にじて、なをするためにテンプレートのがされます。

```
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;
```

クラスラムダとこのキャプチャ

クラスのメンバでされるラムダはのうちにそのクラスのです

```
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);
}
};

```

そのようなラムダは、そのクラスのであるだけでなく、クラスでされているクラスとじアクセスをします。

Lambdasは、がびされたオブジェクトインスタンスをす `this` ポインタをできます。これは、 `this` をキャプチャリストにすることによってわれます。

```

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);
    }
};

```

`this` がされると、ラムダはそのクラスのメンバーを、そのクラスにあるかのようにすることができ。したがっての `this->` がそのようなメンバーにされます。

`this` はによってされますが、のではされないことにしてください。これはポインタである `this` のによってされます。そのため、ラムダがしていない `this`。ラムダアウトがそれをしたオブジェクトのにわたってする、ラムダはになるがあります。

これは、ラムダが `mutable` とする `this` なく `this` をできることもします。それはオブジェクトではなく、 `const` であるポインタです。つまり、メンバが `const` でないりです。

また、デフォルトのキャプチャ `[=]` と `[&]` も `this` にすることにしてください。そして、ともポインタのでキャプチャします。には、デフォルトがえられたときに `this` をキャプチャリストにするのはりです。

ラムダは、ラムダがされたでされた`this`オブジェクトのコピーをできます。これは`*this`をキャプチャリストにすることです

```
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
    }
};
```

ファンクタをしてラムダを**C++ 03**にする

C++のラムダは、**ツール**をくためのになをするなです。したがって、ラムダをファンクタにすることによって、のをC++ 03でることが出来ますはるかにです。

```
// 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;
}

```

ラムダが`mutable`あれば、ファンクタの`call-operator`を`const`、すなわち

```

R operator()(int arg1, int arg2) /*non-const*/ {
    /* lambda-body */
    return R();
}

```

ラムダ

ユークリッドの`gcd()`をラムダとしてくとしましょう。として、のようになります。

```

int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a%b);
}

```

しかし、ラムダはにすることはできません。ラムダをびすはありません。ラムダにはがありません。ラムダので`this`をすると、`this`がキャプチャされたものをし`this`ラムダがメンバのてされているとします。そうでないはエラーです。では、このをどうやってするのでしょうか

`std::function`

ラムダはまだされていない`std::function`へのをすることができ`std::function`

```

std::function<int(int, int)> gcd = [&](int a, int b){
    return b == 0 ? a : gcd(b, a%b);
};

```

これはしますが、えめにするがあります。それはいすびしのわりにタイプをしています、れやすい`gcd`コピーするか`gcd`をすことはラムダがのオブジェクトをするのでします、ラムダではしません。

2つのスマートポインタの

```

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);
    });
};

```

これはくのオーバーヘッドをしますが、コピー/すことができ、すべてのコピーはをします。それはあなたがラムダをすことをにし、そうでなければのよりではありません。

Y-コンビネータをする

いユーティリティのけをりて、これらのをすべてすることができます

```

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)

```

たちは`gcd`をのようになすことができます

```

auto gcd = make_y_combinator(
    [](auto&& gcd, int a, int b){
        return b == 0 ? a : gcd(b, a%b);
    }
);

```

`y_combinator`は、ラムダのコンセプトです。するまで、のけなくともがです。これはまさにラムダのです。

あなたはのとして "recurse"をるラムダをします。あなたがしたいとき、あなたはのをします。

`y_combinator`は、そのをそのでびすが、な "recurse"オブジェクトつまり、`y_combinator`をのとしてつオブジェクトをします。それは`y_combinator`とばれるりのをラムダにもします。

するに

```

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)
});
```

なやなオーバーヘッドがなくラムダでがあります。

インラインパラメータパックのアンパックに**lambdas**をする

C++ 14

パラメータパックをするには、に、するたびにヘルパーをするがあります。

このおもちゃのでは

```
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`は、インデックスのパラメータパックをし、するがあります。これをうには、ヘルパーをびさなければなりません。したパラメータパックをするたびに、カスタムヘルパをするがあります。

これはラムダでけることができます。

のように、パラメータパックをラムダのびしセットにすることができます。

```
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

foldでは、`index_over()` をのようにできます。

```
template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
    return [] (auto&& f){
        ((void) (f(index<Is>)), ...);
    };
}
```

これをしたら、これをして、パラメータパックをでアンパックするわりに、のコードで2のオーバーロードをして、パラメータパックを "インライン"でできます。

```
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` でラムダにさ `index_over` がある `std::integral_constant<std::size_t, ???>`。これはっている `constexpr` への `std::size_t` のにしない `this`、たちは、このようなたちは、それをすなどのコンパイラ、としてそれをすることが出来ます `std::get<i>` の。

のおもちやのにるには、のようにきします。

```
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
    });
}
```

これははるかにく、それをするコードにロジックをします。

ライブでしてみましょう。

オンラインでラムダをむ <https://riptutorial.com/ja/cplusplus/topic/572/ラムダ>

89: リテラル

き

に、リテラルは、とがそのスペルかららかなをすです。えは、`42`はリテラルですが、`x`はそのをり、そのをるためにののコードをむためにをなければならないので、`x`はありません。

しかし、C++ 11では、[ユーザーのリテラルもされています](#)。これはのではリテラルではありませんが、びしのとてできます。

Examples

`bool`の2つののいずれかをす[キーワード](#)。

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

`bool`の2つののいずれかをす[キーワード](#)。

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

nullptr

C++ 11

NULLポインタをす[キーワード](#)。のポインタまたはメンバーへのポインタにして、ののNULLポインタをできます。

```
Widget* p = new Widget();
delete p;
p = nullptr; // set the pointer to null after deletion
```

`nullptr`がポインタではないことにしてください。 `nullptr`のは、`std::nullptr_t`としてられる `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
}
```

この

クラスのメンバでは、**キーワード** `this` は、がびされたクラスのインスタンスへのポインタです。
`this` メンバですることはできません。

```
struct S {
    int x;
    S& operator=(const S& other) {
        x = other.x;
        // return a reference to the object being assigned to
        return *this;
    }
};
```

タイプ `this` メンバの CV-にする `X::f` ある `const`、そのの `thisf` ある `const X*` ので、`this` からデータメンバをするためにすることができません `const` メンバー。に、`this` は、それがれるからの `volatile` をします。

C++ 11

`this` は、でないデータメンバのブレースまたはイコールイニシャライザでもできます。

```
struct S;
struct T {
    T(const S* s);
    // ...
};
struct S {
    // ...
    T t{this};
};
```

`this` はなので、することはできません。

リテラル

リテラルは、フォームのなです

- 10リテラル

0の101,2,3,4,5,6,7,8,9、0の100,1,2,3,4,5,6,7,8,9 7,8,9

```
int d = 42;
```

- 8リテラル

0または1の80、1、2、3、4、5、6、7

```
int o = 052
```

- 16リテラル

シーケンス0xまたはシーケンス0Xのに1つの160,1,2,3,4,5,6,7,8,9、a、A、b、B、c、Cがきます。
。、d、D、e、E、f、F

```
int x = 0x2a; int X = 0X2A;
```

- バイナリリテラルC ++ 14

シーケンス0bまたはシーケンス0Bのに1つの20,1がきます。

```
int b = 0b101010; // C++14
```

- は、されているは、のいずれかまたはをむことができますがされているは、

- unsigned-suffixuまたはU

```
unsigned int u_1 = 42u;
```

- long-suffixlまたはLまたはlong-long-suffixシーケンスllまたはシーケンスLLC ++ 11

のもじにされます。

```
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
```

ノート

リテラルのはとをしません。0xDeAdBaBeUと0XdeadBABEUはじをします1つのlong-long-suffixで、llまたはLL、してllまたはLLです

のリテラルはありません。-1などの、なをうリテラルがすにマイナスをします。

C99よりのCただしC ++ではないでは、long intにまらないの10はunsigned long intをつことができます。

#ifまたは#elifのすると、すべてのきはstd :: intmax_tをつようにし、すべてのなしはstd :: uintmax_tのようになります。

オンラインでリテラルをむ <https://riptutorial.com/ja/cplusplus/topic/7836/リテラル>

90: リファクタリング

き

リファクタリングとは、のコードをにすることをします。リファクタリングは、コードをしてをしたり、バグをしたりしてわれることがいですが、をしたり、バグをすることなくコードをすることをにしています。

Examples

リファクタリング

ここでは、リファクタリングのをけるのあるプログラムをします。これは、C++ 11をするなプログラムで、1から100までのすべてのをしてすることとしており、レビューのために [CodeReview](#)にされたプログラムについています。

```
#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";
}
```

このプログラムのはのようになります。

3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 53 61 67 71 73 79 83 89 97

にくのは、プログラムがである₂をできないということです。プログラムのりのをすることなくそ

のをにコードをすることで、プログラムをリファクタリングして2つのにするほうがかもしれません - リストをするものと、。それがどのようにえるかはのとおりです。

```
#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';
}
```

このバージョンをしてみると、にしくしていることがわかります。

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

のステップは、2のifがにでないことにづくことです。ループのロジックは、えられたのをそののまでします。これは、のがする、そのうちのなくとも1つがそののでなければならぬためです。そのだけをするのプログラマはしままで、このがられます

```
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;
}
```

```

    }
}
if (isprime) {
    primes.push_back(no);
}
}
return primes;
}

```

をもっとわかりやすいようにすることができます。たとえば、`primecount` にはのカウンタではありません。わりに、それはのベクトルへのインデックスです。また、`no` は "number" のとしてわれることがあります、`break` なくし、`n` をうのがです。また、`break` なくし、`n` をするにいくつかのをえることもできます。

```

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;
}

```

私たちはまた、「レンジ・フォー」をして `main` リファクタリングをうこともできます。

```

int main()
{
    std::vector<int> primes = prime_list(100);
    for (auto p : primes) {
        std::cout << p << ' ';
    }
    std::cout << '\n';
}

```

これは、リファクタリングがわれるかもしれない1つのです。はさまざまなをするかもしれない。しかし、リファクタリングのはわりません。これは、をすることなく、コードのとパフォーマンスをさせることです。

ゴト クリーンアップ

CであったC++コードベースでは、`goto cleanup` のパターンをつけることができます。 `goto` コマンドは、のワークフローをしにくくするので、これはしばしばされます。くの、リターン、ループ、にきえることができます。しかし、`goto cleanup` では、クリーンアップロジックをりくがあります。

```

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;
}

```

C++では、このをするためにRAIIをうことができます

```

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;
}

```

このから、のコードをリファクタリングしけることができます。たとえば、VectorRAIIをstd::unique_ptrまたはstd::vectorきえます。

オンラインでリファクタリングをむ <https://riptutorial.com/ja/cplusplus/topic/7600/リファクタリング>

91: リンケージ

き

リンケージは、Cなどののでかれたとにをリンクできるようにをコンパイルするようにコンパイラにします。

- extern リテラル { -seq opt }
- extern リテラル

ここでは、C++がCとをつようにするためには、すべてのコンパイラがextern "C"をサポートするがあり、extern "C++"は、みリンケージをオーバーライドしてデフォルトをするためにされます。サポートされているのリンケージはです。

Examples

Unixライクなオペレーティングシステムシグナルハンドラ

シグナルハンドラはCびしをしてカーネルによってびされるので、をコンパイルするときにCびしをするようにコンパイラにするがあります。

```
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;
        }
        // ...
    }
}
```

CライブラリヘッダーをC++とにする

ほとんどののはCとC++なので、ACライブラリヘッダーはC++プログラムにみむことができます。たとえば、のfoo.hえてfoo.h。

```
typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);
```

`make_foo`のはにコンパイルされ、ヘッダとともにオブジェクトでされます。

C++プログラムは`#include <foo.h>`することができますが、コンパイラは`make_foo`がCシンボルとしてされていることを`make_foo`せず、おそらくがされたでしょうとします。それがのをつけることができたとしても`make_foo`ライブラリではなく、すべてのプラットフォームは、びしにびしC++をするCおよびC++、およびC++コンパイラのじびしをし`make_foo`セグメンテーションフォールトがするがある、`make_foo` Cのびしでびされることをしています。

このをするは、ヘッダーのほとんどすべてのを`extern "C"`ブロックで`extern "C"`です。

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);

#ifdef __cplusplus
} /* end of "extern C" block */
#endif
```

とき`foo.h` Cプログラムからまれている、それだけでのとしてされますが、とき`foo.h` C++のプログラムからまれている、`make_foo`になる`extern "C"`ブロックとコンパイラをすためになっているだろうunmangledなとCのびしをします。

オンラインでリンクージをむ <https://riptutorial.com/ja/cplusplus/topic/9268/>リンクージ

92: ループ

き

ループは、がたされるまでりしのグループをします。C++には、for、while、do ... whileの3のプリミティブループがあります。

- while ステートメント。
- while while ステートメント
- for *init-statement* ; ; *expression* ステートメント ;
- for *range-for-range-declaration* *for-range-initializer* ステートメント。
- 。
- する ;

にきループよりもalgorithmびしがましい。

アルゴリズムがにかあるいはにているをむならば、アルゴリズムびしはよりになり、しばしばよりでエラーのこりにくいものになります。

にシンプルなループがなただし、アルゴリズムをしているは、バインダーとアダプタのがです、ループをしてください。

Examples

ベースのFor

C++ 11

for ループは、インデックスをせずに、またはイテレータにアクセスすることなく、イテレータベースののをするためにできます。

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

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

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

これは、ののをするvalで、vすべてのをりしします。の

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

のものでです。

```

{
    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
    }
}

```

これは、Ranges TSのC++でのサポートのためにされました20。

この、ループはのようになります。

```

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

```

`auto val`はのをすることにしてください。のはにされたのコピーになります。私たちはイテレータからコピーをしています。にされたがなは、`const auto &val`をすることができます。 `auto`をうもありません。のからにであるり、なをできます。

イテレータにアクセスするがあるは、range-based forはあなたをけることはできませんなくともするはありません。

それをしたいは、そうすることができます

```

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

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

```

`const` コンテナをとっている、`const` をすることができます

```

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

```

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

シーケンスがプロキシオブジェクトをし、そのオブジェクトを `const` であるがある、をします。あなたのコードのをさせるがいでしょう。

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

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

ベースの `for` される "range" タイプは、のいずれかになります。

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

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

のりてはカウントされないことにしてください。

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

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

- メンバ `begin()` と `end()` をつで、のにイテレータをします。ライブラリコンテナはされますが、ユーザもできます。

```
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 << " ";
 }
}

```

- するのタイプの `begin(type)` び `end(type)` について、ルックアップをしてさできる `type`。これは、クラスタイプをすることなくタイプをするのにです

```

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 << " ";
 }
}

```

## ループの

`for` ループは、ループ `condition` がであるに、`loop body` ステートメントをします。ループ `initialization statement` がに1される。サイクルの、`iteration execution` がされる。

`for` ループはのようになれます。

```

for (/*initialization statement*/; /*condition*/; /*iteration execution*/)
{
 // body of the loop
}

```

## プレースホルダーステートメントの

- `initialization statement` このステートメントは、`for` ループののに1だけさ `for` ます。 `int i = 0, a = 2, b = 3` など、1つののをできます。これらは、ループのでのみです。じのループのになれたは、ループのになれます。
- `condition` このステートメントは、ループののにされ、ループがされると、ループをし `false`。
- `iteration execution` ないり、このは、にのの、ループのになれます `for` ループは にされた `break`、`goto`、`return` またはスローされた。 `a++`, `b+=10`, `c=b+a` など、`iteration execution` にのステートメントをできます。

まかな `for` のようにきえるループ、 `while` ループがあります。

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

`for` ループをするもなケースは、のだけステートメントをすることです。たとえば、のをしてください。

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

なループもあります

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

ループのにされたをすをにします。

```
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
```

しかし、すでにされているをしてにしたいは、をします。

```
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
```

ノート

- ・ステートメントとインクリメントステートメントは、ステートメントとなをすることができます。しかし、のから、ループにするのみをすることがベストプラクティスです。
- ・ステートメントでされたは、 `for` ループのスコープでのみされ、ループのにされます。
- ・ `initialization statement` されたは、ループにできること、および `condition` チェックされたもできることをれないでください。

0から10までカウントするループの

```
for (int counter = 0; counter <= 10; ++counter)
{
```



```
std::cout << counter << '\n';
}
// counter is not accessible here (had value 11 at the end)
```

## コードの

- `int counter = 0`は`counter`を`int counter = 0`しますこれは`for`ループでのみできます。
- `counter <= 10`は、`counter`が10かどうかをべるブールです。 `true`、ループがされます。 `false`、ループはします。
- `++counter`は、のチェックのに`counter`のを1ずつインクリメントするインクリメントです。

すべてのステートメントをのままにすると、ループをできます。

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

の`while`ループにするものはのとおりです。

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

ただし、`break`、`goto`、または`return`をするか、またはをスローすることによってループをすることが出来ます。

<algorithm>ヘッダーをせずにSTLコレクションたとえば`vector`からすべてのをするのなはのとおりです。

```
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;
}
```

## whileループ

`while`ループは、されたが`false`されるまで、ステートメントをりしし`false`。これは、コードのブロックがされるかがにかっていないにされます。

たとえば、0から9までのすべてのをするには、のコードをします。

```
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、の2つのステートメントをみわせることができます

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

ループをするには、のことができます。

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

whileループの、つまりdo...whileがあります。については、[do-whileループの](#)をしてください。

の

forループとwhileループのでは、オブジェクトをすることもできます。このオブジェクトは、ループのわりまでスコープにあるとみなされ、ループのりしをしてされます。

```
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.
```

しかし、do...whileループでじことをすることはされません。わりに、ループののにをし、ループがしたにをスコープからしたいは、とループのをローカルスコープにみますオプション。

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

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

これは、のためであるdo...whileループはループのの にされてwhile にしており、って、のいずれかのがのにされなくなりますループ。

## Do-whileループ

はありませんに、サイクルのわりでチェックされていることをいて、DO-whileループは、whileループとによくています。したがって、ループはなくとも1はされることがされます。

のコードでは、のりしのにが`false`とされるため、`0`がされます。

```
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
```

`while(condition);`にセミコロンをれないでください`while(condition);`これは`do-while`でとされます。

`do-while`ループとはに、ののにはが`false`とされるため、はもしません。

```
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
```

`while`ループは、`break`、`goto`、または`return`ステートメントをしてがになることなくすることができます。

```
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
```

な`do-while`ループは、のスコープをとするマクロをきむためにもされることがありますその、マクロのからのセミコロンはされ、ユーザーがするがあります。

```
#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);
```

## ループしける

ループステートメントは、のシーケンスからフローをするためにされます。がスコープをれると、そのスコープでされたすべてのオブジェクトがされます。 `break`と`continue`はループです。

`break`ステートメントは、これすることなくループをします。

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

のコードがされます

```
1
2
3
```

`continue`はループをすぐにするのではなく、りのループをスキップしてループのにしますのチェックをむ。

```
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";
}
```

のコードがされます

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

このようなフローのは、がにすることがながあるため、 `break`や`continue`はえめにされます。、よりなでみやすくしやすくなります。たとえば、の`break`つの`for`ループはのよういきされるかもしれせん

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

continueの2のは、のようきすことcontinueできます。

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

## - サブにわたって

ベースのループをすると、ベースのforループのとなるプロキシオブジェクトをすることによって、のコンテナまたはのサブパートをループすることができます。

```
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();
}
```

はできる

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

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

プリントアウトする

forループのfor(:range\_expression)でされるオブジェクトは、forループのにれになることにしてください。

オンラインでループをむ <https://riptutorial.com/ja/cplusplus/topic/589/ループ>

## 93: なコンパイル/リンカエラー—GCC

### Examples

エラー '\*\*\*'はこのスコープでされていません

このエラーは、なオブジェクトがされているにします。

コンパイルしない

```
#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;
 }
}
```

```
#include <iostream>

int main(int argc, char *argv[])
{
 {
 int i = 2;
 std::cout << i << std::endl;
 }

 return 0;
}
```

ほとんどの、このエラーはなヘッダーがまれていないにします `#include <iostream>`な  
いで `std::cout` をする

コンパイルしない

```
#include <iostream>

int main(int argc, char *argv[])
{
 doCompile();

 return 0;
}

void doCompile()
{
 std::cout << "No!" << std::endl;
}
```

```
}
```

```
#include <iostream>

void doCompile(); // forward declare the function

int main(int argc, char *argv[])
{
 doCompile();

 return 0;
}

void doCompile()
{
 std::cout << "No!" << std::endl;
}
```

または

```
#include <iostream>

void doCompile() // define the function before using it
{
 std::cout << "No!" << std::endl;
}

int main(int argc, char *argv[])
{
 doCompile();

 return 0;
}
```

コンパイラはコードをからにします。になくとも**または**されていなければなりません。

`\*\*\* 'へのの

このリンクエラーは、リンクがみのシンボルをできないにします。ほとんどの、されているライブラリがリンクされていないにします。

## qmake

```
LIBS += nameOfLib
```

## cmake

```
TARGET_LINK_LIBRARIES(target nameOfLib)
```

## g++びし

```
g++ -o main main.cpp -Llibrary/dir -lnameOfLib
```



また、されたすべての.cpp ファイルをコンパイルしてリンクすることをやるかもしれません  
functionsModule.cppはなをします。

```
g++ -o binName main.o functionsModule.o
```

なエラー—\*\*\*そのようなファイルやディレクトリはありません

コンパイラはファイルをつけることができませんソースファイルは#include "someFile.hpp"します  
。

## qmake

```
INCLUDEPATH += dir/Of/File
```

## cmake

```
include_directories(dir/Of/File)
```

## g ++びし

```
g++ -o main main.cpp -I dir/Of/File
```

オンラインでなコンパイル/リンカエラーGCCをむ <https://riptutorial.com/ja/cplusplus/topic/4256/>  
なコンパイル-リンカエラー-gcc-

## 94: なテンプレートパターンCRTP

き

クラスがクラステンプレートからテンプレートパラメータの1つとしてするパターン。CRTPは、C++でをするためにされます。

### Examples

#### にりされるテンプレートパターンCRTP

CRTPは、コンパイルにのプロパティをえるためにできる、とのにするでなです。これはテンプレートパラメータの1つとしてクラスをるクラステンプレートをつことによってします。これにより、クラスへの`this`ポインタの`static_cast`をにすることができます。

もちろん、これは、CRTPクラスがのクラスのクラスとしてにされなければならないことをします。そして、クラスはそれをクラスにすがあります。

#### C++ 14

`begin()` と `end()` をすべてサポートするコンテナのセットがあるとしましょう。コンテナのライブラリには、よりくのがです。 `begin()` と `end()` のみについて、そのをるCRTPクラスをすることができます

```
#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);
 }
};
```

のクラスは、begin() および end() を begin() サブクラスの front()、back()、size() および operator[] をし begin()。サブクラスのは、ににりてられたです。

```
#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_;
};
```

DynArray クラスのユーザーは、CRTP クラスによってされるインターフェイスをのようになにできます。

```
DynArray<int> arr(10);
arr.front() = 2;
arr[2] = 5;
assert(arr.size() == 10);
```

このパターンはに、にをするためにするびしをけ、にキャストにします。

```
DynArray<int> arr(10);
DynArray<int>::Base & base = arr;
base.begin(); // no virtual calls
```

クラス Container<DynArray<int>> begin() でのキャストは、コンパイラがコードをにすることをにし、にテーブルのはわれません。

クラスはテンプレートされており、2つのなる DynArray なるため、クラスがクラスにしないのでにできるように、のクラスへのポインタをすることはできませんタイプ

```
class A {};
class B: public A{};

A* a = new B;
```

コードのをけるための **CRTP**

Visitor Patternのでは、CRTPのながされています。

```
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); }
 // ...
};
```

のIShapeは、じでじをするがあります。それはなタイピングです。わりに、たちのためにこれをうにしいタイプをすることができます

```
template <class Derived>
struct IShapeAcceptor : IShape {
 void accept(IShapeVisitor& visitor) const override {
 // visit with our exact type
 visitor.visit(*static_cast<Derived const*>(this));
 }
};
```

そして、はにアクセプタからするがあります

```
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;
};
```

したコードはありません。

オンラインでなテンプレートパターンCRTPをむ <https://riptutorial.com/ja/cplusplus/topic/9269/な>

テンプレートパターン-crtip-

## 95: の

コンストラクトのがされていないには、では、にいくつかのをすが、えられたでがこるかするはありませんにいくつかのをします。これは、 **がされた**と、ががこるかをすることがされると、もこりないのとはです。

## Examples

TUのグローバルのの

では、グローバルののがされていますが、ののはです。

のファイルをつプログラム

- foo.cpp

```
#include <iostream>

int dummyFoo = ((std::cout << "foo"), 0);
```

- bar.cpp

```
#include <iostream>

int dummyBar = ((std::cout << "bar"), 0);
```

- main.cpp

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

としてされるがあります

```
foobar
```

または

```
barfoo
```

スタティック・イニシアチブ・オーダー・フィアスコにつながるがあります。

の

スコープきがをするにはさすぎるにされた、のはされません。

```
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

```

また、がにされ、のがののにある、のはされません。

```

enum Color {
 RED = 1,
 GREEN = 2,
 BLUE = 3,
};
Color c = static_cast<Color>(4);

```

ただし、のでは、ソースがすべてののとしくないにもかかわらず、のにあるため、るいはではありません。

```

enum Scale {
 ONE = 1,
 TWO = 2,
 FOUR = 4,
};
Scale s = static_cast<Scale>(3);

```

ここでs3をとっている、とのになるONE、 TWO、 およびFOUR。

## のvoid \* valueからのキャスト

void\*は、タイプ、オブジェクトへのポインタにされ、T\*、しかしにわせされていないT、られたポインタはです。

```

// 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);

```

p2はintのオブジェクトをすることができないため、p3のはです。そのはしくされたアドレスではありません。

## いくつかのreinterpret\_castの

1つのポインタからのへのreinterpret\_cast、またはあるからのへのreinterpret\_castのはされていません。

```

int f();
auto fp = reinterpret_cast<int(*) (int)>(&f); // fp has unspecified value

```

1つのオブジェクトポインタからのオブジェクトへの`reinterpret_cast`の、またはあるオブジェクトからのオブジェクトへの`reinterpret_cast`のはされていません。

```
int x = 42;
char* p = reinterpret_cast<char*>(&x); // p has unspecified value
```

しかし、ほとんどのコンパイラでは`static_cast<char*>(static_cast<void*>(&x))`ポインタ`p`が`x`のバイトをしていました。これはC++ 11のなです。については、[をしてください](#)。

## いくつかのポインタの

`<`、`>`、`<=`、または`>=`をして2つのポインタをすると、はのにはされません。

- ポインターはなるをします。オブジェクトはサイズ1のとなされます

```
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
```

- ポインターはじオブジェクトをしますが、アクセスのなるメンバーにポイントします。

```
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;
};
```

## リファレンスがめるスペース

はオブジェクトではなく、オブジェクトとはなり、したいいくつかのメモリバイトをすることはされていません。では、リファレンスがストレージをとするかどうかはです。のくのは、がめるのあるをにべることをにしています。

- `sizeof`がにされる、されるのサイズがされるため、がをしているかどうかにするはされません。
- のはです。したがって、のの2つのしたのアドレスをべて、のサイズをすることはできません。
- のアドレスがられた、はのアドレスになるので、へのポインタをすることはできません。
- クラスにメンバがある、`offsetof`をしてそのメンバのアドレスをしようとすると、レイアウト



トクラスではないため、のがします。

- ・ クラスにメンバーがある、そのクラスはもはやレイアウトではないため、をするためにされるデータにアクセスしようとすると、またはのがします。

には、によっては、ポインタとにをすることができ、したがってポインタとじのをすることがありますが、ができるため、はくスペースをしません。たとえば、のようになります。

```
void f() {
 int x;
 int& r = x;
 // do something with r
}
```

コンパイラは、ににである $r$ として $x$ とのすべてのきえる $r$ 、のりので $f$ と $x$ 、およびするために、のストレージりてませ $r$ 。

のの

がのをつ、されるはです。のコードは $x = 1, y = 2$ または $x = 2, y = 1$ をできますが、

```
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

C++ 17では、のののはのままです。

しかし、のはにされ、びしのオブジェクトはのがあるにされます。

```
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));
}
```

これはするがあります

```
bar
make_int(1)
from_int(1)
make_int(2)
from_int(2)
```

または

```
bar
make_int(2)
from_int(2)
make_int(1)
from_int(1)
```

make または from のいずれかのに bar がされず、されないことがあります

```
bar
make_int(2)
make_int(1)
from_int(2)
from_int(1)
```

または。 make\_int のに C++ 17 の bar のには、 from\_int をするに make\_int のを make\_int ていたのと同じように、

もなライブラリクラスの

## C++ 11

すべてのライブラリコンテナは、にであるがののままである。たとえば、のコードでは、 v2 には {1, 2, 3, 4} がまれますが、 v1 はずであるとありません。

```
int main() {
 std::vector<int> v1{1, 2, 3, 4};
 std::vector<int> v2 = std::move(v1);
}
```

クラスによっては、move-from がにされているものがあります。もなケースは、 std::unique\_ptr<T> です。これは、に null になることがされています。

オンラインでのをむ <https://riptutorial.com/ja/cplusplus/topic/4939/>の

## 96: ベエ

`std::sort` ファミリは、 `algorithm` ライブラリにあります。

### Examples

されたでシーケンスコンテナをソートする

コンテナのにすでにオーバーロードされているがある、 `std::sort` をなファンクタとともにして、またはのいずれかで `std::sort` ことができます。

#### 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

C++ 14では、オブジェクトにテンプレートをするはなく、されるについてオブジェクトをさせることができます。

```
std::sort(v.begin(), v.end(), std::less<>()); // ascending order
std::sort(v.begin(), v.end(), std::greater<>()); // descending order
```

オーバーロードされたいによるシーケンスコンテナのソート

けがされない、 `std::sort` は、のペアにして `operator<` をびすことによってをけします。これは、 `bool` または `bool` をすがあります。、ポインタなどにはにでされています。

これをオーバーロードして、デフォルトの `sort` びしをユーザーでさせることができます。

```
// 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;
}

```

## compare をしてシーケンスコンテナをソートする

```

// 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;
}

```

## ラムダをしたシーケンスコンテナのソート **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;
}

```

## コンテナのべえとけ

`std::sort` はライブラリヘッダー `<algorithm>` にあり、のイテレータでされたのをソートするためのライブラリアルゴリズムです。 `std::sort` は、2つのをするためにされたファンクタののパラメータをとります。これはをするです。 `std::sort` は **して** いないことにしてください。

は、に **Strict、Weak Ordering** をすがあります。シンプルなまたはのです。

ランダムアクセスイテレータをつコンテナは、 `std::sort` アルゴリズムをしてソートすることができます。

## C++ 11

```

#include <vector>
#include <algorithm>

std::vector<int> MyVector = {3, 1, 2}

//Default comparison of <
std::sort(MyVector.begin(), MyVector.end());

```

`std::sort`では、イテレータがランダムアクセスイテレータであるがあります。シーケンスコンテナ `std::list`と `std::forward_list` C++ 11がはランダムアクセスイテレータをしないので、`std::sort`できません。しかし、のイテレータでするソートアルゴリズムをする `sort`メンバをとっています。

## C++ 11

```
#include <list>
#include <algorithm>

std::list<int> MyList = {3, 1, 2}

//Default comparison of <
//Whole list only.
MyList.sort();
```

メンバーの `sort` はにリストをソートするので、のサブレンジをソートすることはできません。しかし、 `list`と `forward_list` はなスプライスがあるので、ソートするをリストからし、ソートしてから、かなりなにしておくことができます

```
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);
}
```

## std::map およびでソート

このでは、マップをしてキーの `string` をソートします。のでは、 `std::string` わりに、 `class` をむのことができます。

```
#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';
 }
}

```

```

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)

```

しいキーをつエントリがなは、 `multimap` わりに `multimap` をし `map` ののように。

でをソートするには、なファンクタ `std::greater<>` をしてマップをします。

```

#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';
 }
}

```

```

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)

```

## みみのべえ

`sort` アルゴリズムは、2つのイテレータでされたシーケンスをソートします。これはビルトインCスタイルともばれますをソートするのにです。

## 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>());
```

C++ 11よりでは、のわりをのサイズをとって「する」がありました。

## 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);
```

オンラインでべえをむ <https://riptutorial.com/ja/cplusplus/topic/1675/べえ>

## 97:

C++では、`<random>`ヘッダによってわれます。このヘッダーは、ランダムデバイス、およびをします。

ランダムなデバイスは、オペレーティングシステムによってされるをします。のにするか、またはですがあります。

ランダムジェネレータは、シードについてをします。のは、なしタイプのすべてののにわたっています。ライブラリのすべてのジェネレータは、すべてのプラットフォームのじシードにしてじをします。

は、ランダムジェネレータまたはランダムデバイスからをし、なをつをします。ディストリビューションはプラットフォームにせず、なるプラットフォームでじシードをつじジェネレータにしてなるをすることができます。

## Examples

の

にできるのランダムをするには、`std::random_device`をジェネレータとしてするがあります。

```
#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`は、ランダムジェネレータとじでされます。

しかし、なソースえば、ハードウェアデバイスがにでない、のエンジンにして `std::random_device`をすることができます。

このようなをするには、 `entropy` メンバ ジェネレータがにであるときにゼロをすがですが、くのライブラリGCCのlibstdc ++とLLVMのlibc ++のは、のランダム。

の

ジェネレータは、にされたについてできるをします。つまり、です。のがなでは、ジェネレータをしないでください。

```
#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;
}
```

このコードは、ジェネレータと、がしい[0,9]ののををするをします。に、がされたかをえます。

`std::uniform_int_distribution<T>`のテンプレートパラメータは、されるべきのををする。floatまたはdoubleをするには、 `std::uniform_real_distribution<T>`をします。

のディストリビューションにジェネレータをする

ジェネレータは、のディストリビューションにできますまた、するがあります。

```
#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;
}
```

```
}
```

このでは、ジェネレータは1つだけされています。その、3つのなるでランダムをするためにされます。 `rigged_dice`は、0と5ののをするが、ほとんどのにします5するので、5ある100 / 105。

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/1541/>

## 98: メンバー

- `virtual void f;`
- ボイド `g= 0;`
- `// C ++ 11`
  - ボイド `h` オーバーライド。
  - `void i` オーバーライド。
  - `void jfinal;`
  - `void kfinal;`
- でテンプレートでないメンバのみが `virtual` なり `virtual` 。
- `C ++ 11` を `override` は、クラスからメンバーをオーバーライドするときにオーバーライドをすることをめし `override` 。
- クラスは、したオブジェクトをクラスへのポインタをしてできるようにするデストラクタをつことがよくあります。デストラクタがでない、このようなのは `[expr.delete]§5.3.5/ 3` につながります。

## Examples

### `C ++ 11`でのオーバーライドの

`C ++ 11`では、シグネチャのにされた、の `override` はなをち `override` 。

- ベースクラス
- クラスのは `virtual`

この `run time` はありませんが、にコンパイラとなります

のは、オーバーライドをしないのをしています。

`override` なし

```
#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"; }
};
```

`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"; }
};
```

`override`はキーワードではなく、シグネチャにのみれるなであることにしてください。のすべてのコンテキストでは、`override`はとしてできます。

```
void foo() {
 int override = 1; // OK.
 int virtual = 2; // Compilation error: keywords can't be used as identifiers.
}
```

## メンバ

## メンバ

```
#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()"
}
```

## メンバーなし

```
#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()"
}

```

な

C++ 11では、メソッドシグネチャにれたにメソッドオーバーライドをする`final`をしました。

```

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";
 }
};

```

`final`は`virtual`メンバでのみでき、メンバにはできません

`final`とに、クラスの`virtual`のオーバーライドをぐびしのオーバーライドもあります。

`override`と`final`をみわせて、ましいをることができます。

```

class Derived1 : public Base {
public:
 void foo() final override {
 std::cout << "Derived1::Foo\n";
 }
};

```

## コンストラクタとデストラクタにおけるのりい

コンストラクタとデストラクタののは、にしたときにしばしばします。

```
#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;
}
```

ベースコンストラクタからびされると、`base::v`がびされます。

コンストラクタからびされると、`derived::v`がびされます。

デストラクタからびされると、`::v`がびされます。

ベースデストラクタからびされると、`base::v`がびされます。

このにあるは、クラスがまだコンストラクタのされていないか、デストラクタのにはすでにされているメンバーをしていて、メンバのびしがではないということです。したがって、C++オブジェクトのとのには、`*this` はコンストラクタまたはデストラクタのクラスであり、よりしたクラスではありません。

```
#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);
}

```

な

また、`virtual` がな で、に `= 0` をすることですることもできます。1つのをつクラスは、クラスとみなされ、インスタンスすることはできません。すべてのをまたはするクラスのみをインスタンスできます。

```

struct Abstract {
 virtual void f() = 0;
};

struct Concrete {
 void f() override {}
};

Abstract a; // Error.
Concrete c; // Good.

```

がとしてされていても、デフォルトをえることができます。それにもかかわらず、はとしてであるとなされ、クラスはインスタンスするにそれをするがあります。この、クラスののバージョンは、クラスのバージョンをびすことさえです。

```

struct DefaultAbstract {
 virtual void f() = 0;
};
void DefaultAbstract::f() {}

struct WhyWouldWeDoThis : DefaultAbstract {
 void f() override { DefaultAbstract::f(); }
};

```

これをやりたいがいくつかあります

- インスタンスすることはできませんが、クラスのインスタンスをげないクラスをしたいは、デストラクタをなとしてできます。デストラクタであるため、インスタンスのりてをできるようにするには、それをするがあります。そして、**デストラクタがもがい、すでににメモリリークをぐためであるとして**、々はのをするから、なパフォーマンスヒットをしません `virtual`。これは、インターフェイスをするときにです。

```

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.

```

- のほとんどまたはすべてののにコードがまれている、そのコードをクラスのバージョンにして、コードをにすることができます。

```

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.

```

オンラインでメンバーをむ <https://riptutorial.com/ja/cplusplus/topic/1752/メンバー>

## 99: しているヘッダ

### Examples

`system_clock`は、プログラムのにしたをするためにできます。

#### 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";
}
```

このでは、`sleep_for`をして、アクティブなスレッドを`std::chrono::seconds`でされたスリープさせましたが、のコードはにが加かるびしです。

#### 2つののをする

このは、2つののをつけるをしています。は//、さらには//でします。

プログラムは、2000のをします。

```
#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";
}

```

オンラインでしているヘッダをむ <https://riptutorial.com/ja/cplusplus/topic/3936/している-chrono-ヘッダ>

## Examples

### のキャッチ

`try/catch` ブロックは、をするためにされます。 `try` セクションのコードはをスローするコードで、 `catch` のコードがをします。

```
#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();
 }
}
```

のタイプをするには、の `catch` をできます。の `catch` がする、メカニズムはコードにするにそれらをさせようとしています。

```
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();
}
```

クラスからしたクラスは、クラスにしての `catch` で `catch` できます。のでは、 `std::length_error` と `std::out_of_range` の2つの `catch` を `std::length_error` の1つのにきえることができ `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();
}
```

`catch` はにされるので、よりな `catch` をにしてください。さもなければ、コードはしてびされない

かもしれません

```
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. */
}
```

もう1つの、スローされたオブジェクトをキャッチするcatch-allハンドラです。

```
try {
 throw 10;
} catch (...) {
 std::cout << "caught an exception";
}
```

をスローする

によっては、キャッチしたログのきみやのなどでかをい、のにバブルアップさせたいもあります。これをうには、あなたがまえるをすべてげすことができます

```
try {
 ... // some code here
} catch (const SomeException& e) {
 std::cout << "caught an exception";
 throw;
}
```

throw;をうthrow;なしでは、キャッチされているをスローします。

## C++ 11

されたstd::exception\_ptrにrethrow\_exceptionために、C++ライブラリにrethrow\_exceptionがあります。rethrow\_exceptionは、プログラムに<exception>ヘッダをめることによってできます。

```
#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
```

## ブロックをすコンストラクタ

### イニシャライザのリストでをキャッチするの

```
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;
};
```

## ののためにブロックをす

```
void function_with_try_block()
try
{
 // try block body
}
catch (...)
{
 // catch block body
}
```

### どちらとじです

```
void function_with_try_block()
{
 try
 {
 // try block body
 }
 catch (...)
```

```

{
 // catch block body
}
}

```

コンストラクタとデストラクタでは、`catch`ブロックがをもうスローしますキャッチブロックにのスローがないはキャッチされたもので、はなりません。

`main`は、のとに`try`ブロックをつことができますが、`main`の`try`ブロックは、ローカルのはたはのにするをキャッチしません。わりに、`std::terminate`がびされます。

## デストラクタでブロックをす

```

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.
 }
};

```

これはですが、デストラクタからのげにはがです。スタックのきしにびされたデストラクタがをスローしたように、`std::terminate`がびされます。

## ベストプラクティスし、`const`のキャッチ

に、ポインタではなくでスローするのはいとみなされますが、`const`ではキャッチされません。

```

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;
}

```

によってキャッチするのがいであるの1つは、キャッチブロックにされるときまたはのキャッチブロックにするときオブジェクトををすることです。また、によってキャッチすることで、をにうことができ、オブジェクトのスライスけることができます。しかし、を`throw e;`ている`throw e;`ように、のをしてください、オブジェクトスライシングをことができます`throw e;`はされているとしてのコピーをします



```
#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;
}
```

ののやメッセージのなどをわなないことがなは、`const`をキャッチするとコンパイラはをい、パフォーマンスをさせることができます。しかし、これはとしてオブジェクトスプライシングをきこすがありますのを。

にメモリやリソースをにりてることにして、しないを`catch`ブロックにけることにしてください。たとえば、のコピーにメモリがしたり、マスクがされたロギングにI/Oストリームがスローされたりするがあるため、`logic_error`、`runtime_error`またはそのサブクラスをすると`bad_alloc`がスローされることがあります。

## ネストされた

### C++ 11

に、レベルファイルシステムエラーやデータエラーなどからをキャッチして、よりなをスローすると、いくつかのながであることをすながありますWebにをすることができないなどされません。これにより、がのでのにすることがになります。また、エラーだけでエラーがすると、プログラマーはがしたアプリケーションのをつけることができます。このソリューションのは、コールスタックがりてられ、のがわれることです。これにより、はオリジナルののテキストをしくしたテキストにでめることができます。

ネストされたは、をするレベルのをレベルのにびつけることによってをすることをしています。

`std::nested_exception`のにすることができます `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();
 }
}

```

な

```

exception: run() failed
MyException: Couldn't open nonexistent.file
exception: basic_ios::clear

```

`std::exception` からしたのでのみするは、コードをすることもできます。

## std :: uncaught\_exceptions

### C++ 17

C++ 17は `int std::uncaught_exceptions()` しましたされた `bool std::uncaught_exception()` をきえて、  
 されていないのをりました。これにより、クラスがスタックのきしにされるかどうかをすることが  
 できます。

```

#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;
 }
}
```

```
In ~Foo: Commit
In main: Rollback
exception/main:Error
```

## カスタム

のをとしてげてはならず、わりにのクラスをするか、のクラスをしてください。

`std::exception` からしたのクラスをつことは、それをういです。は `std::exception` からしたカスタムクラス `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;
}

};

```

## スローキャッチの

```

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();
}

```

ダムのエラーメッセージをげるだけでなく、エラーのをにすのをするだけでなく、エラーがはるにかつになります。

エラーメッセージをうまくできるようにするクラスがあります `std::runtime_error`

あなたはこのクラスからすることもできます

```

#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;
 }

};

```

は`what()`をクラス `std::runtime_error` からオーバーライドしていないことにしてください。つまり、クラスの`what()`ます。さらにがあれば、それをきすることができます。

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/1354/>

# 101: とのセマンティクス

## Examples

### ディープコピーとサポート

がセマンティクスをち、にりてられたオブジェクトをするがある、コピーでは、はそれらのオブジェクトのしいコピーをりてるがあります。コピーアサインメントでもこれをするがあります。

こののコピーは「ディープコピー」とばれます。これは、そうでなければセマンティクスをつセマンティクスをにけり、セマンティクスにします。

```
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

セマンティクスは、されたデータをにコピーしないように、Valueようなをします。ユーザーがをきこすようなでをする、オブジェクトからの"コピー"は、したデータののままにすることができま

```
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_);
 }
};

```

に、オブジェクトをさせながらディープコピーをしたいは、このようなタイプをコピーにすることもできます。

```

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_);
}
};

```

`unique_ptr`して、ゼロのルールをすることもできます。

```

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;
};

```

オブジェクトのなが、そののすべてのオブジェクトとになる、はのをちます。つまり、オブジェクトをコピーするといしオブジェクトがされ、しいオブジェクトのはいオブジェクトからえなくなります。

もなC++は、のセマンティクスをっています

```

int i = 5;
int j = i; //Copied
j += 20;
std::cout << i; //Prints 5; i is unaffected by changes to j.

```

ほとんどのライブラリでされたは、のセマンティクスもっています。

```
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.
```

あるタイプのインスタンスがそのオブザーバブルをのオブジェクトのオブジェクトとできる、そのタイプのセマンティクスは、あるオブジェクトをすとのオブジェクトでがするようになります。

C++ポインタは、それらがしているオブジェクトにしてのセマンティクスをっていますが、それらがしているオブジェクトのにしてセマンティクスをっています。

```
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.
```

C++リファレンスにはセマンティクスもあります。

オンラインでとのセマンティクスをむ <https://riptutorial.com/ja/cplusplus/topic/1955/>とのセマンティクス

## 102: のカテゴリ

### Examples

カテゴリの

C++のには、そののについてのカテゴリがりてられます。のカテゴリは、C++のオーバーロードにします。

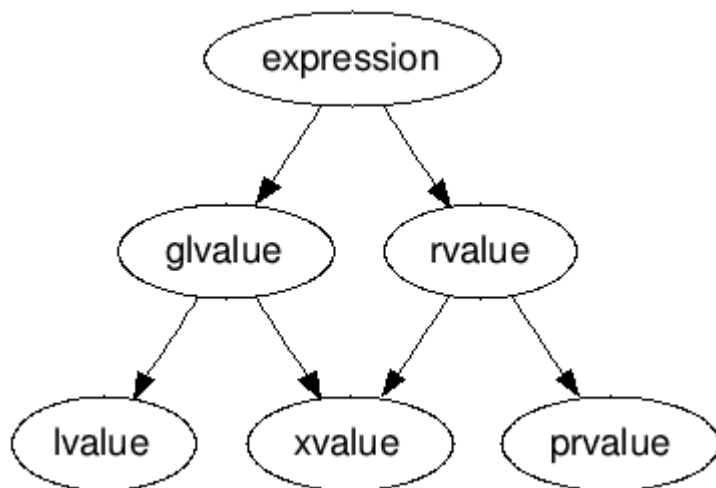
のカテゴリは、にする2つのな、々のプロパティをします。1つのは、そのがをするかどうかである。をつオブジェクトをする、はをちます。はにしないかもしれませんが、オブジェクトにはまだがまれています。

もう1つのプロパティは、のからにすることがかどうかです。よりには、をパラメータとしてする、rのパラメータにバインドするかどうかをします。

C++は、lvalueではあるがではない、xvalueなは、およびprvalueながないの3つののカテゴリをしています。C++にはIDがなく、できないはありません。

C++では、glvalueIDきのとrvalueなの2つのカテゴリがされています。これらは、のカテゴリのなグループけとしてする。

このグラフはイラストとしてちます



け

prvalueなは、がないで、そのはオブジェクトのにされ、にできます。これにはのものがまれますが、これらにされません。

- オブジェクトをす `std::string("123")`。
- をさないびし
- リテラルリテラルをく - これはです、 `1`、 `true`、 `0.5f`、または `'a'`

- ラムダ

みみアドレスの `&` は、これらのにはできません。

## xvalue

`xvalue`は、IDをち、にできるオブジェクトをすです。 `xvalue`のなえは、それらがするオブジェクトがすぐにされることになりしたがって、「エキサイティング」な、のうちにそこからすることはないということです。

えられた

```
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
```

は、をつですが、にすることはできません。これらのには、、、みみである、をするからなるがあります。

なはなるですが、はのでもることができます

```
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
```

さらに、ほとんどのリテラルたとえば `4`、`'x'`などはprvalueですが、リテラルはです。

## glvalue

glvalue「された」は、かどうにかにかかわらず、をつです。このカテゴリには、lvaluesはあるができないとxvaluesをち、ながまれますが、prvaluesのないはきます。

にがある、それはglvalueです。

```
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は、をっているかどうかにかかわらず、にできるのです。

よりには、rvalueは、`T &&` ここで`T`は`expr`ののですパラメータをとるのとしてできます。このようなのには、rvalueのみをとってえることができます。のがされている、はrvalueパラメータをしないをします。そして、しなければ、あなたはエラーをけります。

rvalueのカテゴリには、すべてのxvalueとprvalue、およびそれらののみがまれます。

ライブラリ`std::move`は、のをににするためにします。よりには、をxvalueにえます。これは、のIDなしのprvalueであっても、`std::move`パラメータとしてすことで、IDのパラメータをしてxvalueになるからです。

のをしてください。

```
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`は、`std::string&&`タイプののパラメータをるコンストラクタがあります。これはに「ムーブコンストラクタ」とばれます。ただし、`str`カテゴリはrvalueにではないため、そのコンストラクタをオーバーロードとびすことはできません。わりに、`const std::string&` overloadコピーコンストラクタをびします。

3はをえます。 `std::move`のりは`T&&`で、`T`はされたパラメータののです。したがって、`std::move(str)`は`std::string&&`します。りがrvalueであるびしはrvalueにxvalueであるため、`std::string`コンストラクタをびすことができ`std::string`。 3の、`str`がしましたのはです。

4はに`std::string`にし`std::string`。これには`std::string&&`をとるオーバーロードがあります。`std::string("new value")`はrvalueにprvalueであるため、そのオーバーロードをびすことがあります。したがって、な`str`にされ、のはのにきえられ`str`。

5は、`str`をする`str_ref`というの`str_ref`をし`str`。これは、のカテゴリがするところです。

`str_ref`は`std::string`ですが、`str_ref`カテゴリはではありません。それはです。はい、そうです。このため、`std::string`コンストラクタを`str_ref`でびすことはできません。したがって、6は`str`のを`str`にコピーし`test3`。

それをするには、 `std::move` する必要があります。

オンラインでのカテゴリをむ <https://riptutorial.com/ja/cplusplus/topic/763/>のカテゴリ

## 103: と

き

プロミスとは、あるスレッドからのスレッドに1つのオブジェクトをフェリーするためにされます。

`std::promise` オブジェクトは、をするスレッドによってされます。

`std::future` オブジェクトをしてをしたり、がかどうかをべたり、がになるまでをしたりすることができます。

## Examples

### `std :: future` と `std :: promise`

のでは、のスレッドによってされるをします。

```
{
 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();
}
```

の

このコードは `std::async` バージョンをしますが、`async` が `deferred` ポリシーでにびされたかのようにします。このには、`async` のな `future` もありません。された `future` はをることなくされる `future` があります。

```
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とstd::future

std::packaged\_taskは、りのにしてとそれにするをねます

```

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);
}

```

スレッドはすぐにされます。たちはそれをりすことも、スコープのわりにさせることもできます。  
。 std::threadのびしがすると、はです。

これは、 std::asyncとはなり std::futureされる std::futureは、スレッドがするまでにブロックされます。

## std::future\_errorとstd::future\_errc

std::promiseおよびstd::futureのがたされない、 std::future\_errorのがスローされます。

のエラーコードメンバーのはstd::future\_errcで、はいくつかのテストケースとともにのとおりです。

```

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 */
};

```



な

```
int test()
{
 std::promise<int> pr;
 return 0; // returns ok
}
```

アクティブな、

```
int test()
{
 std::promise<int> pr;
 auto fut = pr.get_future(); //blocks indefinitely!
 return 0;
}
```

```
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();
}
```

## std :: promiseを2する

```
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とstd::async

のなマージソートでは、のmerge\_sortタスクをするためにstd::asyncがされています。

std::futureは、をってそれらをするためにされます

```
#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);
}
}

```

ここでは、`std::async`はポリシー`std::launch_deferred`ます。これは、すべてのびしでしいスレッドがされないようにするためです。このの、`std::async`びしはでわれ、`std::future::get()`びしでします。

`std::launch_async`は、すべてのびしでしいスレッドをします。

デフォルトのポリシーは`std::launch::deferred`| `std::launch::async`は、がしいスレッドをするためのポリシーをすることをします。

## クラス

- `std::async`をします。
- `std::future`へのアクセスをします。
- `std::promise`のをパッケージします。
- `std::packaged_task`りのにしてとそれにするをねます。

オンラインでとをむ <https://riptutorial.com/ja/cplusplus/topic/9840/>と

## 104: ミューテックス

### Examples

#### std::recursive\_mutex

なmutexは、じスレッドがリソースをにロックすることをにします。

これにしてののはほとんどありません。のなでは、ロックをせずにのオーバーロードされたコピーをびすがあります。

```
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();
```

オンラインでミューテックスをむ <https://riptutorial.com/ja/cplusplus/topic/9929/ミューテックス>

## Examples

では、ユーザーはセットのなをできます。これらののはにとばれます。とそれにするは、のようにされます。

```
enum myEnum
{
 enumName1,
 enumName2,
};
```

、のすべてのタイプとはなるものです。この、こののは`myEnum`です。こののオブジェクトは、ののをきぐことがされます。

でされたは、ののです。はでされていますが、にアクセスするためにはスコープ`::`はありません。したがって、ののは`enumName1`です。

### C++ 11

`scope`は、のにアクセスするためにオプションでできます。したがって、`enumName1`は、`myEnum::enumName1`りにすることもできます。

は、のにして0からまり1ずつするがりてられます。したがって、の、`enumName1`は0をち、`enumName2`は1をちます。

には、ユーザーがのをりてることもできます。このはでなければなりません。がにされていないのは、の+1のにされます。

```
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
};
```

### switchの

のなは`switch`のためであり、にマシンにれる。、`switch`ののなは、スイッチにデフォルトのがまれておらず、`enum`のすべてののがされていない、コンパイラがをすということです。

```
enum State {
```

```

 start,
 middle,
 end
};

...

switch(myState) {
 case start:
 ...
 case middle:
 ...
} // warning: enumeration value 'end' not handled in switch [-Wswitch]

```

のりし

エニユメレーションでするみみはありません。

しかし、いくつかのがあります

- するのみをつ<sub>enum</sub>

```

enum E {
 Begin,
 E1 = Begin,
 E2,
 // ..
 En,
 End
};

for (E e = E::Begin; e != E::End; ++e) {
 // Do job with e
}

```

## C++ 11

enum classではoperator ++をするがあります

```

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;
}

```

- コンテナを<sub>std::vector</sub>としてする

```

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();

```

その

```

for (std::vector<E>::const_iterator it = all_E.begin(); it != all_E.end(); ++it) {
 E e = *it;
 // Do job with e;
}

```

## C++ 11

- または `std::initializer_list` とな

```

enum E {
 E1 = 4,
 E2 = 8,
 // ..
 En
};

constexpr std::initializer_list<E> all_E = {E1, E2, /*..*/ En};

```

その

```

for (auto e : all_E) {
 // Do job with e
}

```

## スコープき

C++ 11ではスコープき`enum`としてられているものをしています。これらは、メンバーが`enumname::membername`されなければならないです。スコープきは、`enum class`をしてされます。たとえば、のをするには

```

enum class rainbow {
 RED,
 ORANGE,
 YELLOW,
 GREEN,
 BLUE,
 INDIGO,
 VIOLET
};

```

のにアクセスするには

```
rainbow r = rainbow::INDIGO;
```

enum class、キャストなしでにintすることはできません。したがってint x = rainbow::REDはです。

Scopedでは、メンバをすのにされるであるになるをすることもできます。デフォルトではintです。Tic-Tac-Toeゲームでは、そのを

```
enum class piece : char {
 EMPTY = '\0',
 X = 'X',
 O = 'O',
};
```

enumのとおり、enumはのメンバーのにコンマをつけることができます。

## C++での11

### スコープき

```
...
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;
}
```

### スコープの

```
...
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);
```

なマルチファイルのがここにあります [Blind fruit merchant example](#)

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/2796/>



## 106: りてとコピー

- コピーコンストラクタ
- MyClass const MyClass other;
- MyClass MyClass その;
- MyClass volatile const MyClass other;
- MyClass の MyClass その;
- コンストラクタ
- MyClass operator = const MyClass rhs;
- MyClass = MyClass rhs;
- MyClass operator = MyClass rhs;
- const MyClass operator = const MyClass rhs;
- const MyClass operator = MyClass rhs;
- const MyClass operator = MyClass rhs;
- MyClass = const MyClass rhs;
- MyClass = MyClass rhs;
- MyClass = MyClass rhs;

### パラメーター

|         |                                                                              |
|---------|------------------------------------------------------------------------------|
| rhs     | は、コピーコンストラクタとコンストラクタのにしてです。えは、コンストラクタ <b>MyClass operator = MyClass rhs;</b> |
| プレースホルダ | プレースホルダ                                                                      |

さらなるのためのそののれたリソース

とコピーコンストラクタのいはですか

とコピーコンストラクタC++

[GeeksForGeeks](#)

[C++の](#)

## Examples

は、のにされたオブジェクトをのオブジェクトのデータできえるときです。これをにげてみましょう

```
// 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
}

```

ここでは`foo`オブジェクトをにしているときにをびすことができます。その、はりて`foo2`する`foo`。をびすときにされるすべてののは、`operator=`でされます。ここにながされます <http://cpp.sh/3qtbm>

## コピーコンストラクタ

、コピーコンストラクタは、コンストラクタのなです。は、すでにしないまたはにされていないオブジェクトをするためにされます。つまり、コピーのオブジェクトをにせずに、りてられているオブジェクトからすべてのデータをコピーします。さて、とじコードをてみましょう。しかし、コンストラクタをコピーコンストラクタにしてください

```

// 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;
}

```

ここで行うことができます `Foo foo2 = foo;` では、オブジェクトをにするのにオブジェクトをりてま  
す。これはにべたように、コピーコンストラクタです。そして、はオブジェクトfooからのデー  
タをにったので、 `foo2` オブジェクトのパラメータintをすはないことにつく `foo2`。 に、をします。

<http://cpp.sh/5iu7>

## コピーコンストラクタとコンストラクタ

ここでは、コピーコンストラクタとコンストラクタがあるものをにて、それぞれのをげてみま  
しょう。はをじコードでてみましょう。このコードはの2つのコードとています。これをとてみま  
しょう

```

// 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;
}
```

```
2
2
```

ここでは、に `Foo foo2 = foo;` というをしてコピーコンストラクタをびすことができます `Foo foo2 = foo;` 。 はしていなかったのに、 `foo3=foo` にりてられているので、をfoo3にびします。

オンラインでりてとコピーをむ <https://riptutorial.com/ja/cplusplus/topic/7158/> りてとコピー

## 107: タイプ

- `std::atomic<T>`
- `std::atomic_flag`

`std::atomic`は、`TriviallyCopyable`へのアトミックなアクセスをにします。これは、アトミックまたはロックをしてわれる、にします。されているロックフリーのタイプは`std::atomic_flag`です。

## Examples

### マルチスレッドアクセス

アトミックをして、2つのスレッドでされるメモリをにみきすることができます。

### データをきこすのあり

```
#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;
}
```

のは、したみりをきこし、のにつながるがあります。

## スレッドセーフな

```
#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;
}
```

のは、`atomic` データのすべての `store()` および `load()` オペレーションが、カプセルされた `int` をアクセスからするのでです。

オンラインでタイプをむ <https://riptutorial.com/ja/cplusplus/topic/3804/タイプ>

## 108:

### Examples

の

はにしますが、にconstポインタにているわけではありません。は、アンパサンドの&にされます。

```
int i = 10;
int &refi = i;
```

ここでrefiはiされたです。

は、オブジェクトのエイリアスのようにするポインタのセマンティクスをします。

```
refi = 20; // i = 20;
```

1つののにのをすることもできます。

```
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&!
```

はにしくされなければならず、ですることはできません。のコードは、コンパイルエラーをきこします。

```
int &i; // error: declaration of reference variable 'i' requires an initializer
```

また、ポインタとはなり、 nullptrへのをバインドすることはできません

```
int *const ptri = nullptr;
int &refi = nullptr; // error: non-const lvalue reference to type 'int' cannot bind to a
temporary of type 'nullptr_t'
```

C++はのエイリアスです

C++のリファレンスはなるAliasまたはののです。ちょうどたちのほとんどがパスポートのとニックネームをとってすることができます。

はりせず、メモリをしません。のアドレスをすると、それをするのアドレスとじアドレスがされます。

```
int main() {
 int i = 10;
 int &j = i;

 cout<<&i<<endl;
 cout<<&b<<endl;
 return 0;
}
```

のでは、の`cout`がじアドレスをします。のとしてをと、はじになります

```
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;
}
```

このでも、の`cout`がじアドレスをします。

C++ Referencesはなるエイリアスであり、エイリアスがされるためには、エイリアスができるものがです。

これがこのようながコンパイラエラーをげるなです

```
int &i;
```

なぜなら、エイリアスはもしていないからです。

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/1548/>



## 109:

### Examples

#### ブレーク

もいループまたは`switch`からけします。

```
// 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;
 }
}
```

#### する

のみループのわりにジャンプします。

```
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;
}
```

#### う

**do-whileループ**をします。

```
// Gets the next non-whitespace character from standard input
char read_char() {
 char c;
 do {
 c = getchar();
 } while (isspace(c));
 return c;
}
```

#### にとって

**forループ**、またはC ++ 11では、**ベースのforループ**がされています。

```
// print 10 asterisks
```

```
for (int i = 0; i < 10; i++) {
 putchar('*');
}
```

## while

whileループをします。

```
int i = 0;
// print 10 asterisks
while (i < 10) {
 putchar('*');
 i++;
}
```

## ベースのforループ

```
std::vector<int> primes = {2, 3, 5, 7, 11, 13};

for(auto prime : primes) {
 std::cout << prime << std::endl;
}
```

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/7841/>

## 110: キーワード

### Examples

#### クラスメンバー

このコンテキストにおける`mutable`は、オブジェクトのにをえることなく`const`オブジェクトのデータフィールドがされることをすためにされる。

なのをキャッシュすることをえているなら、おそらくこのキーワードをうべきです。

`const`メソッドでロックされてロックされているロック `std::unique_lock` データフィールドがある、このキーワードもできます。

このキーワードをして、オブジェクトのなをしないでください。

#### キャッシュきの

```
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;
};
```

#### ラムダ

デフォルトでは、ラムダの`operator()`は`const`です。これは、ラムダにして`const`をすることをす。メンバーのをすために、ラムダは`mutable`とマークされ、な`operator() const`ます。

```
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
```

オンラインでキーワードをむ <https://riptutorial.com/ja/cplusplus/topic/2705/キーワード>

# 111: のない

## Examples

のクラス

きクラスまたはとはなり、のないクラスおよびは、されているでインスタンスされ、コンストラクタまたはデストラクタをつことはできません。

```
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;
```

メンバー

C++のとして、なコンパイラではクラスをメンバーとしてできます。

```
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;
```

として

エイリアスをするには、`typedef`を`using`のようにすることもでき`typedef`。

## 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;
```

のメンバーは、のスコープにし、このスコープののすべてのとはなるものでなければなりません。ここのは、"struct"をしたメンバーのとじですが、にしています。

```
struct Sample {
 union {
 int a;
 int b;
 };
 int c;
};

int main()
{
 Sample sa;
 sa.a = 3;
 sa.b = 4;
 sa.c = 5;
}
```

オンラインでのないをむ <https://riptutorial.com/ja/cplusplus/topic/2704/>のない

## 112:

### き

のライブラリをするときののをぐためにされます。は、 、クラス、などのなです。

- `opt { -seq }`
- インラインネームスペース `opt { -seq } /* C ++ 11* /`
- インライン `opt - seq opt { -seq } /* C ++17* /`
- `enclosing-namespace-specifier :: identifier { declaration-seq } /* C ++17* /`
- = された ;
- namespace ネストされた `opt namespace-name`をします。
- `name -names-names -specifier opt namespace-name`をした `- seq 。 /* C ++ 11* /`

**キーワード** `namespace`は、によって3つのなるがあります。

1. オプションとでまれたシーケンスのには、**しいをするか、それらのでのをします**。を  
と、はの**ないになります**。
2. そのにとがくと、**のエイリアスがされます**。
3. `using`してにけると、**`usingディレクティブ`がされます**。これにより、されたのがされてい  
なによってされますしかし、のスコープのはされません。クラススコープで**`usingディレク  
ティブ`をすることはできません**。

`using namespace std;`している。どうして `namespace std`はなのでこれは、**がするがいことをします**  
。

```
//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

とはですか

C ++は、C ++エンティティ、クラス、のであり、ののにがいています。にコードをくとき、その  
にするきにはのをけるはないが、そののはをしなければならない。のは`<namespace>::<entity>`です

。

```
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.
```

は、するをまとめてグループするのにです。ショッピングモールのを。に、ショッピングモールはいくつかのにかれており、はのカテゴリーのをしています。あるがエレクトロニクスをとっているのにし、のがをとっているかもしれないタイプのこれらのなは、いがしているをつけるのにちます。ネームスペースは、いのようなc++プログラマがにすることで、している、クラス、をつけるのにちます。

```
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.)
 };
}
```

つのがあらかじめされています。はありませんが、`::`ですことができるグローバルです。

```
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
 }
}

```

をる

をするのはです

```

//Creates namespace foo
namespace Foo
{
 //Declares function bar in namespace foo
 void bar() {}
}

```

`bar`をびすには、をにし、にスコープ`::`けてするがあります。

```

Foo::bar();

```

あるをのにすることができます。たとえば、のようになります。

```

namespace A
{
 namespace B
 {
 namespace C
 {
 void bar() {}
 }
 }
}

```

## C ++ 17

のコードはのようになすることができます

```

namespace A::B::C
{
 void bar() {}
}

```

の

`namespace`のなは、それらをする事メンバーをすることです。

```

namespace Foo
{
 void bar() {}
}

//some other stuff

```

```
namespace Foo
{
 void bar2() {}
}
```

## ディレクティブの

キーワード'[using](#)'には3つのがあります。キーワード 'namespace'とみわせると、 'using directive' とくことができます

`Foo`すべてののに`Foo::`をかないは、`Foo`をして `using namespace Foo;`できます `using namespace Foo;`すべてのものを`Foo`からインポートします。

```
namespace Foo
{
 void bar() {}
 void baz() {}
}

//Have to use Foo::bar()
Foo::bar();

//Import Foo
using namespace Foo;
bar(); //OK
baz(); //OK
```

ではなく、でしたエンティティをインポートすることもできます。

```
using Foo::bar;
bar(); //OK, was specifically imported
baz(); // Not OK, was not imported
```

ほとんどの、ヘッダーファイルに `using namespace` することはスタイルとしてられます。これがすると、ヘッダーをむすべてのファイルにがインポートされます。 `using` 「`using`ない」はないので、これはのグローバルのよりくの、あるいはしない、あるいはそれのをくがあります。のついては、のをしてください。

```
/****** 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

```

クラススコープで`using`ディレクティブをすることはできません。

## ルックアップ

ななしでをびすとき、コンパイラは、そのパラメータの1つがそのにもまれている、のをびすことをできます。これは「ルックアップ」またはADLとされます。

```

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

```

`Test` からのパラメータが1つもないため、`call` がします。 `SomeClass` は `Test` メンバーであるため、`call_too` するため、ADLルールとなります。

## いつADLがこらないか

のルックアップでクラスメンバー、ブロックスコープでされた、またはではないものがつかった、ADLはしません。えば

```

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"
}

```

## インライン

### C++ 11

`inline namespace`は、`inline namespace`のをむネーム`inline namespace`めるので

```

namespace Outer
{
 inline namespace Inner
 {
 void foo();
 }
}

```

### ほとんどです

```

namespace Outer
{
 namespace Inner
 {
 void foo();
 }

 using Inner::foo;
}

```

`Outer::Inner::`および`Outer::`けられたはです。

### はです

```

Outer::foo();
Outer::Inner::foo();

```

`using namespace Inner;`の`using namespace Inner;`テンプレート`のとしていにくい`についてはではあ  
りません。

### にとって

```

#include <outer.h> // See below

class MyCustomType;
namespace Outer
{

```

```
template <>
void foo<MyCustomType>() { std::cout << "Specialization"; }
}
```

- インラインネームスペースは、`Outer::foo`をにします。

```
// outer.h
// include guard omitted for simplification

namespace Outer
{
 inline namespace Inner
 {
 template <typename T>
 void foo() { std::cout << "Generic"; }
 }
}
```

- `using namespace`すると、`Outer::foo`はされません

```
// 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`.
}
```

インラインネームスペースは、いくつかのバージョンをさせて`inline 1`にデフォルトさせるです

```
namespace MyNamespace
{
 // Inline the last version
 inline namespace Version2
 {
 void foo(); // New version
 void bar();
 }

 namespace Version1 // The old one
 {
 void foo();
 }
}
```

い

```
MyNamespace::Version1::foo(); // old version
```

```
MyNamespace::Version2::foo(); // new version
MyNamespace::foo(); // default version : MyNamespace::Version1::foo();
```

1の

のないをすると、にリンケージがあることをすることができますのでしかできません。そのようなは、のとじでされますが、はありません。

```
namespace {
 int foo = 42;
}
```

fooは、されるでのみされます。

ヘッダーファイルにのないをしないことをおめします。これは、それがまれるすべてののコンテンツのバージョンをえるためです。これは、constグローバルをするににです。

```
// 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
```

コンパクトなネストされた

## 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 {};
}
```

あなたは、のができ<sub>a</sub>と<sub>b</sub>とのワンステップでを`namespace a::b` C++ 17でします。

## エイリアス

これは、ライブラリのコンポーネントをするようないののをする、またはするためにされます。

```
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
```

## エイリアスのスコープ

エイリアスはの*using*ステートメントのをけます

```
namespace boost
{
 namespace multiprecision
 {
 class Number ...
 }
}

using namespace boost;

// Both Namespace are equivalent
namespace Name1 = boost::multiprecision;
namespace Name2 = multiprecision;
```

しかし、あなたがのようなものを持っているときに、あなたがエイリアシングしているをさせるがです。

```
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;

```

## エイリアス

は、`namespace` をしてエイリアス つまり、じののをえることができます。エイリアスのメンバには、エイリアスのものであることでアクセスできます。のでは、ネストされた

`AReallyLongName::AnotherReallyLongName` はになるので、`qux` はエイリアス `N` ローカルでしています。そののメンバーは、`N::` をしてにアクセスできます。

```

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());
}

```

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/495/>



## 113: びしオブジェクト

き

びしオブジェクトは、としてできるすべてのC++のです。には、これはC++のSTL invokeにすることも、`std::function`のコンストラクタでうこともできるすべてのものです。ポインタ、operatorをつクラス、のクラス、への、メンバへのポインタ、メンバデータへのポインタ、lambda。びしオブジェクトは、くのSTLアルゴリズムでとしてされます。

Stephan T. Lavavej <>、[ない スライド](#) のには、このドキュメントのにつながります。

### Examples

ポインタ

ポインタはをすもなであり、CでもできますはCのドキュメントをしてください。

びしオブジェクトののために、ポインタはのようになすることができます。

```
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
```

のベクトルソートをするためにポインタをすると、のようになります。

```
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
```

あるいは、のいずれかのでポインタをびすことができます。

- `(*lessThan)(v.front(), v.back()) // All`
- `std::invoke(lessThan, v.front(), v.back()) // <= C++17`

## operatorをつクラスFunctor

`operator()` をオーバーロードするすべてのクラスをオブジェクトとしてできます。これらのクラスはですることができますしばしばファンクタとされます。あるいはC++ 11からLambdasをくことによってコンパイラによってにされます。

```
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"));
// ...
```

ファンクタはのアイデンティティーをっているため、typedefにれることはできず、テンプレートをしてけるがあります。 `std::find_if` のはのようになります。

```
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;
}
```

C++ 17では、のびしは`invoke` `std::invoke(predicate, *i)` ことができます。

オンラインでびしオブジェクトをむ <https://riptutorial.com/ja/cplusplus/topic/6073/びしオブジェクト>

## 114:

### き

このトピックでは、C++ 11からなキーワード `auto` タイプをむについてします。

、コピーやなどのましくないをぐがあるは、`auto` をするたびに `const`、`&` と `constexpr` をするがよいでしょう。これらののヒントは、コンパイラがのをしないようにします。また、`auto` をすることはおめできません。のがにい、にSTLテンプレートのにのみしてください。

## Examples

### データタイプ

このは、コンパイラができるをしています。

```
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>
```

ただし、`auto` キーワードは、`&` または `const` または `constexpr` ヒントなしに、されるをすとはりません

```
// 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;
```

### ラムダオート

データ `auto` キーワードは、プログラマーがラムダをするのになです。これは、ポインタをするためにプログラマがするがあるテキストのをらすのにちます。

```
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
```

デフォルトでは、ラムダのりののがされていない、りののからにされます。

これらの3はにじものです

```
[](int a, int b) -> int { return a + b; };
[](int a, int b) -> auto { return a + b; };
[](int a, int b) { return a + b; };
```

## ループとオート

これは、forループのをするためにautoをするをしています。

```
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
```

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/8233/>

# 115: タイプのキーワード

## Examples

### int

"のアーキテクチャーによってされたなサイズ"のきで、そのにはなくとも-32767+32767がまれていることをします。

```
int x = 2;
int y = 3;
int z = x + y;
```

のをするために、 `unsigned`、 `short`、 `long`、 および `long long` `qv`とみわせることができます。

### ブール

が `true` または `false` いずれかの。

```
bool is_even(int x) {
 return x%2 == 0;
}
const bool b = is_even(47); // false
```

### チャー

「のセットののメンバーをするのになきさ」の。 `char` がされているかなくとも-127から+127までのをつか、なしがなくとも0から255までのをつかどうかは、によってされます。

```
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

`uint_least16_t` とじサイズとアラインメントをつなし。したがって、UTF-16コードをするのになきさです。

```
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

`uint_least32_t` と同じサイズとわせをつなし。したがって、UTF-32コードをするのになきさです。

```
const char32_t full_house[] = U"?????"; // non-BMP characters
std::cout << sizeof(full_house)/sizeof(char32_t) << "\n"; // prints 6
```

く

。C++の3つののもいっています。

```
float area(float radius) {
 const float pi = 3.14159f;
 return pi*radius*radius;
}
```

ダブル

。そのには `float` がまれます。みわせたに `long` し、`long double` そののものむ、`double`。

```
double area(double radius) {
 const double pi = 3.141592653589793;
 return pi*radius*radius;
}
```

いです

なくとも `int` とじさで、に  $-2^{31}-1$  +  $2^{31}-1$  つまり、 $-2^{31}-1$  +  $2^{31}-1$  をむきをします。  
。このは `long int` とくこともできます。

```
const long approx_seconds_per_year = 60L*60L*24L*365L;
```

`long double` のみわせは、3つののもいをつをします。

```
long double area(long double radius) {
 const long double pi = 3.1415926535897932385L;
 return pi*radius*radius;
}
```

## C++ 11

ときに `long` は、こると `long long`、それは、なくともりであるきし `long`、そしてそのは  $9223372036854775807$  なくとも  $-9223372036854775807$  をむ、すなわち、 $-2^{63}-1$  +  $2^{63}-1$ 。  
。

```
// support files up to 2 TiB
const long long max_file_size = 2LL << 40;
```

## ショート

`char`となくともじさで、になくとも-32767+32767をむきをします。このは`short int`とくこともできます。

```
// (during the last year)
short hours_worked(short days_worked) {
 return 8*days_worked;
}
```

な。オブジェクトのがすることがではない`void`、またのがある`void`にするか、を`void`。もさないのりとしてされます。

さらに、`void`ののパラメータでをしてすることもでき`void`。これはパラメータをたないをすることとじですえ、`int main()`と`int main(void)`はじをします。このは、CとののためにできますはC++とはなるをちます。

`void*` "`void`へのポインタ"は、オブジェクトポインタをそのポインタにしてすことができ、じポインタをすというプロパティをとっています。このは、え、CスタイルのAPI `qsort`、`pthread_create` のなコンテキストのなど、あるのがでないインタフェースにした`void*`します。

のを`void`のにすることができ`void`。これはされたとばれます。

```
static_cast<void>(std::printf("Hello, %s!\n", name)); // discard return value
```

これは、のがではなく、そののみがされることをにえるのにです。

## wchar\_t

サポートされているのセットワイドセットともばれますのすべてののをすのになきさのです。  
`wchar_t`がUTF-16などののエンコーディングをするとすると、がありません。

これは、キャラクタ・タイプ`char`よりきなサイズをつため、ASCII 255のキャラクタをするがあるにされます。

```
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"
```

オンラインでタイプのキーワードをむ <https://riptutorial.com/ja/cplusplus/topic/7839/タイプのキーワード>

## 116: キーワード

### Examples

#### const

。あるにすると、そのconstバージョンがされます。constキーワードののについては、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
```

#### C++ 11

そのオペランドのをします。このはされません。

- ・ オペランドのは $e$ のののないで、その、`decltype(e)`のされたである $e$ 。

```
int x = 42;
std::vector<decltype(x)> v(100, x); // v is a vector<int>
```

- ・ オペランド $e$ がのかっこなしでクラスメンバーアクセスである、`decltype(e)`はアクセスされたメンバーのされたです。

```
struct S {
 int x = 42;
};
const S s;
decltype(s.x) y; // y has type int, even though s.x is const
```

- ・ それの、`decltype(e)`は $e$ のとのカテゴリをのように`decltype(e)`ます。
  - $e$ がタイプ $T$ の、`decltype(e)`は $T\&$ です。
  - $e$ が $T$ の $x$ であれば、`decltype(e)`は $T\&\&$ 。
  - $e$ がタイプ $T$  prvalueである、`decltype(e)`は $T$ です。

これには、なカッコがまれています。

```
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

な `decltype(auto)` は、`auto` ものではなく `decltype` ルールをして、のをそのまたはのりからそのの `return` から `decltype` ます。

```
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
```

## された

のなのであるキーワード。

- ですと、`int` がされるため、`signed`、`signed int`、および `int` はじです。
- みわせたに `char` じる `signed char` とはなるタイプである、`char` としても、`char` また、されています。 `signed char` は、なくとも -127+ 127 をむをちます。
- `short`、`long`、または `long long` とみわせると、すでにされているので、しています。
- `signed` は、`bool`、`wchar_t`、`char16_t`、または `char32_t` とみわせること `signed` できません。

```
signed char celsius_temperature;
std::cin >> celsius_temperature;
if (celsius_temperature < -35) {
 std::cout << "cold day, eh?\n";
}
```

## されていない

のなしバージョンをする。

- ですと、`int` がされるので、`unsigned` は `unsigned int` とじです。
- `unsigned char` は、`char` とはなり、たとえ `char` がなしであってもじです。255までののをできます。
- `unsigned` は、`short`、`long`、または `long long` とみわせることもできます。 `bool`、`wchar_t`、`char16_t`、または `char32_t` とみわせることはできません。

```
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
}
```

の

タイプ。あるにすると、そのバージョンがされます。のはシステムの`const`とじをたしますが、`volatile`はオブジェクトのをげません。そのわりに、コンパイラは、そのようなオブジェクトへのすべてのアクセスをとしてわなければなりません。

は、のでは、`memory_mapped_port`ではなかったことがあればしくないだろうののきみ、するよう、コンパイラは、をすることができ`sizeof(int)` 1である`volatile`、それはすべてののための`sizeof(int)`はなるとしてきみ、したがってすべてをにします。

```
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];
 }
}
```

オンラインでキーワードをむ <https://riptutorial.com/ja/cplusplus/topic/7840/キーワード>

# 117:

## Examples

### クラスをする

なは、なクラスであり、、、およびのコンクリートにきすことができます。

### クラス

ポリモーフィッククラスからめましょう

```
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(); }
};
```

### このをむには

- `virtual`のキーワードをして、されたメンバーによってなるいをすることができます。ここで、`get_surface()`と`describe_object()`は、のよりものには々にされることは確かです。がオブジェクトにしてびされると、オブジェクトのクラスにするがにされます。
- シェイプの`get_surface()`をするのはがありません。これはのに`= 0`がくです。これは、がなであることをします。
- クラスは、にデストラクタをするがあります。
- メンバをすることができます。これらのがオブジェクトにしてびされると、はコンパイルにされるクラスにじてされます。ここで`get_double_surface()`はこのようにされています。
- なくとも1つのをむクラスは、クラスです。クラスはインスタンスできません。ポインターやクラスののみをつことができます。

### クラス

クラスがされると、それをさせることができます。えば

```
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;
}
};
```

## いくつかの

- クラスのものをまたはきできます。ファンクションがクラスでであったというは、それをクラスでします。コンパイラにびキーワード`virtual`をえるはありません。しかし、のにキーワード`override`をすることをおめし`override`。これは、のシグネチャがかれていないことによるなバグをぐためです。
- クラスのすべてののがされているは、このクラスのオブジェクトをインスタンスすることができます。そうでなければクラスにもなります。
- すべてのをオーバーライドするはありません。にじて、のバージョンをすることができます。

## インスタンスの

```
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;
}
```

## なダウンキャスト

ポリモーフィッククラスのオブジェクトへのポインタがあるとします。

```
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);
```

ダウンキャストは、なの`Shape`から、`Square`や`Circle`のようなよりなに`Circle`ます。

## なぜダウンキャスト

たいていの、はからしてオブジェクトをすることができるため、オブジェクトののをるはありません。

```
std::cout << "Surface: " << ps->get_surface() << std::endl;
```

あなたがダウンキャストをとしないなら、あなたのデザインはです。

しかし、ダウンキャストするがあるかもしれません。なは、クラスにしてのみするをびすです。

たとえばをえてみましょう。だけがっています。クラスはのようにされます

```
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; }
};
```

`get_diameter()` メンバは、にしてのみします。 `Shape` オブジェクトにされていませんでした。

```
Shape* ps = get_any_shape();
ps->get_diameter(); // OUCH !!! Compilation error
```

どのようにダウンキャスト

`ps` がサークルをしていることができれば、 `static_cast` することができます

```
std::cout << "Diameter: " << static_cast<Circle*>(ps)->get_diameter() << std::endl;
```

これはトリックをいます。しかし、それはにです `ps` が `Circle` のかによってえる、あなたのコードのるいはです。

だから、ロシアのルーレットをくのではなく、に `dynamic_cast` うべきです。これはにクラスのためのものです

```
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;
}
```

でないクラスでは、 `dynamic_cast` はできません。クラスやそのになくとも1つのがです。

## とデストラクタ

クラスがにされることをされており、インスタンスがベースポインタ/としてされている、そのクラスのデストラクタは `virtual` か `protected` いるがあります。の、これはオブジェクトによって `vtable` をチェックし、にづいてしいデストラクタをにびします。の、クラスのポインタ/によってオブジェクトをすることはできません。オブジェクトは、のとしてにわれたにのみできます。

```
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.
```

これらのは、クラスのデストラクタがクラスインスタンスでにびされ、メモリリークをぐことをします。

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/1717/>

## 118: な

なでは、のref-qualifierをするためにをすることがあります。そのようなは、されたにおいてのみれる。あれは

```
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
}
```

Tはコンストラクタびしからされないなので、はなをまない

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

### C++ 17

C++ 17では、クラステンプレートのをできます。のの"a"のコンストラクタは、のユーザになります

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

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

## Examples

### ファクトリ

のリストをけり、されていないのをのにすファクトリをしたいとします。そのようなのはmake\_uniqueであり、これはTしいインスタンスをにし、そのインスタンスをするunique\_ptr<T>をするためにされます。

バリデーションなテンプレートやrのにするは、このようなをくことをにします。

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

のは...タイプののをすパラメータパックをしています。コンパイラは、このパラメータパックを

コールサイトのしいにします。これらは、 `std::forward` をって `T` のコンストラクタにされます。  
このは、の `ref-qualifier` をするためにです。

```
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);
```

オンラインでなをむ <https://riptutorial.com/ja/cplusplus/topic/1750/>な



## 119: メンバ

クラスの 'constメンバ' とは、にはをしますかなは、constメンバがオブジェクトをできないようです。しかし、「えられない」というのはにここでのです。つまり、クラスデータメンバーのりてをうことはできません。

ただし、にすように、エントリをマップにするなど、なをうこともできます。これをすると、このconstはオブジェクトをしているようにえますあるではそうですが。

したがって、のは、constメンバーはクラスデータのをうことができないということです。しかし、それはでしたようなのものをうことができます。

## Examples

### メンバ

```
#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;
}
```

```
}
```

オンラインでメンバをむ <https://riptutorial.com/ja/cplusplus/topic/7120/メンバ>

---

## 120: のるい

### Examples

Charはなしまたはき

では、`char`がきかなしかをしていません。なるコンパイラでは、なるですか、コマンドラインスイッチをしてコンパイラをすることができます。

のサイズ

のは、としてされます。

- `char`
- き
- なし
- `char16_t`と`char32_t`
- `bool`
- `wchar_t`

いて`sizeof(char) / sizeof(signed char) / sizeof(unsigned char)` §3.9.1.1でされ、[basic.fundamental / 1]及び5.3.3.1 [expr.sizeof]、及び`sizeof(bool)`、にインプリメンテーションされ、サイズをたないため、これらのタイプのサイズは、のセクション3.9.1 [basic.fundamental]にあります。

---

## char サイズ

C++のすべてのバージョンでは、5.3.3.1で、`sizeof`は`unsigned char`、`signed char`、`char`が`signed`か`unsigned`かをするですにして1をします。

C++ 14

`char`は、UTF-8コードユニットをするのにした256のなるをするのになきさです。

---

## きおよびなしのサイズ

では、§3.9.1.2で、`signed char`、`short int`、`int`、`long int`、および`long long int`からなるきのリストで、はなくともする`long long int`それはリストにあります。さらに、§3.9.1.3でされているように、これらののそれぞれは、するのなし、`unsigned char`、`unsigned short int`、`unsigned int`、`unsigned long int`、および`unsigned long long int`をちます。するきタイプさらに、§3.9.1.1でされているように、`char`は`signed char`と`unsigned char`とじサイズとわせのをっています。

C++ 11

C++ 11よりのバージョンでは、`long long`と`unsigned long long`はC++のではありませんでした。しかし、Cへの、C99では、このコンパイラが`long long unsigned long long`として`long long`サポート`long long`れており、Cとじをつ`unsigned long long`の`unsigned long long`の`unsigned long long`がサポート`long long`れています。

したがって、ではのことがされています。

```
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)
```

タイプのなサイズは、によってえられていません。わりに、タイプにはサポートできるのがあります。これは§5.2.4.2.1のCからされた§3.9.1.3でされています。タイプのサイズは、とされるビットをすることによって、このからすることができます。どのプラットフォームでも、にサポートされているタイプのは、よりきくなるがあることにしてください。きのの、は1のにし、よりにされる2のにはしないことにしてください。これは、なプラットフォームがにすることをにすることのためです。

| タイプ            |                                                            | なビット |
|----------------|------------------------------------------------------------|------|
| signed char    | $-127 \sim 127 - 2^7 - 12^7 - 1$                           | 8    |
| unsigned char  | $0 \sim 2^8 - 1$                                           | 8    |
| signed short   | $-32,767 \sim 32,767 - 2^{15} - 12^{15} - 1$               | 16   |
| unsigned short | $0 \sim 2^{16} - 1$                                        | 16   |
| signed int     | $-32,767 \sim 32,767 - 2^{15} - 12^{15} - 1$               | 16   |
| unsigned int   | $0 \sim 2^{16} - 1$                                        | 16   |
| signed long    | $-2,147,483,647 \sim 2,147,483,647 - 2^{31} - 12^{31} - 1$ | 32   |
| unsigned long  | $0 \sim 2^{32} - 1$                                        | 32   |

## C++ 11

| タイプ              |                                                                                    | なビット |
|------------------|------------------------------------------------------------------------------------|------|
| signed long long | $-9,223,372,036,854,775,807 \sim 9,223,372,036,854,775,807 - 2^{63} - 1 - 12^{63}$ | 64   |
| unsigned long    | $0 \sim 2^{64} - 1$                                                                | 64   |

| タイプ  |  | なビット |
|------|--|------|
| long |  |      |

タイプはサイズよりも大きくできるため、でのサイズがなるがあります。これのなは、LLP64システム64ビットWindowsなどが32ビット`ints`と`long`をち、LP64システム64ビットLinuxなどがある64ビットデータモデルLP64とLLP64です。32ビット`int`および64ビット`long`このため、はすべてのプラットフォームでとなすことはできません。

C ++ 11

のがなは、`<cstdint>`ヘッダーのをしますが、では、では、`int8_t`、`int16_t`、`int32_t`、`int64_t`、`intptr_t`、`uint8_t`、`uint16_t`、`uint32_t`、`uint64_t`および`uintptr_t`。

C ++ 11

## char16\_t と char32\_t サイズ

`char16_t`と`char32_t`のサイズは、`char16_t`でされているようにされており、3.9.1.5でされている。

- `char16_t`は、UTF-16コードをすのになきさで、`uint_least16_t`とじサイズ、およびを  
`uint_least16_t`ます。って、なくとも16ビットのサイズであることがされる。
- `char32_t`は、UTF-32コードをすのになきさで、`uint_least32_t`とじサイズ、およびを  
`uint_least32_t`ます。したがって、なくとも32ビットのサイズであることがされる。

## bool きさ

`bool`のサイズはであり、`_1`もなくてもよい。

## wchar\_t のサイズ

§3.9.1.5でされている`wchar_t`は、サポートされているロケールでのセットのすべてのコードを  
のことができますタイプです。これは、になるとしてられているのの1つとじサイズ、およびを  
ちます。こののサイズは、5.3.3.1でされているようにされており、えは、なくとも8,16、また  
は32ビットであってもよい。たとえば、システムがUnicodeをサポートしている、`wchar_t`はな  
くとも32ビットでなければなりませんこのルールのはWindowsです。`wchar_t`はのために16ビットで  
す。これはC90ISO 98991990§4.1.5からされたものであり、ないえである。

にじて、`wchar_t`のサイズは`wchar_t`、または32ビットであることがよくありますが、ずしもそう  
であるとはりません。これらのなはのとおりです。

- UnixおよびUnixライクなシステムでは、`wchar_t`は32ビットで、はUTF-32でされます。
- Windowsでは、`wchar_t`は16ビットで、UTF-16でされます。
- 8ビットのみをサポートするシステムでは、`wchar_t`は8ビットです。

## C++ 11

Unicodeサポートがまれる、することがされる`char` UTF-8の、`char16_t` UTF-16、または`char32_t` わりにする、UTF-32 `wchar_t`。

# データモデル

のように、のはプラットフォームによってなるがあります。もなモデルはのとおりです。サイズはビットでします。

| モデル         | <code>int</code> | <code>long</code> | ポインタ |
|-------------|------------------|-------------------|------|
| 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   |

これらのモデルのうち、

- 16ビットWindowsはLP32をしました。
- 32ビット\* nixシステムUnix、Linux、Mac OSXなどのUnixライクなOSとWindowsではILP32をしています。
- 64ビットWindowsはLLP64をします。
- 64ビット\* nixシステムはLP64をします。

ただし、これらのモデルは、ではにされていません。

## 1バイトのビット

C++では、バイトは`char`オブジェクトがするです。バイトのビットは、`CHAR_BIT`でされ、なくとも8であるがある`climits`によってえられます。ほとんどのシステムは8ビットのバイトをち、POSIXでは`CHAR_BIT`がに8であるがありますが、`CHAR_BIT` 8よりきい、すなわち、バイトは、8,16,32、または64ビットからされる。

## ポインタの

`reinterpret_cast`をつてへのポインタをキャストしたは、ですが、"...は、となるマシンのアドレ

ツシングをとっているにはかないことをしています。"

```
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
```

に、からされたポインタもされています。

ポインタをとしてするしいは、 `uintptr_t` または `intptr_t` をすることです。

```
// `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は、 `uintptr_t` C99、6.3.2.3のではC99をします。

`void` へのなポインタをこのにしてから `void` へのポインタにし、そののをポインタとするというをつなしです。

のプラットフォームのでは、フラットなアドレスをすることができますが、 `uintptr_t` は `char *` ですが、かなり、 `void *` を `uintptr_t` するときには、 `uintptr_t` から `void *` にキャストバックするときにはしてください。

- XSI / Open System Interfaces システムでは、 `intptr_t` と `uintptr_t` がです。それはオプションです。
- Cのでは、はオブジェクトではありません。 `uintptr_t` がポインタをできることがCによってされていません。とにかく、POSIX 2.12.3のにはがです

すべてのポインタは `void` ポインタとじをたなければならない。ポインタを `void *` にしても、そのをしてはならない。そのようなからじた `void *` は、キャストをしてをうことなくのポインタにすることができます。

- C99 §7.18.1

の `u` がしないかしないにのみなる `typedef` がされている、それらは6.2.5にされているするきおよびなしをするものとする。これらのするのうちの1つをするは、のをしなければならない。

`uintptr_t` は、きでかりやすいようにポインタのビットをしたいにがあります。

のはです。 <limits>ヘッダーは、すべてののとをする `std::numeric_limits<T>` テンプレートをします。これらのは、 <climits>および `C ++ 11 <stdintypes>` ヘッダーによってCによってされるをたしています。

- `std::numeric_limits<signed char>::min()` は `SCHAR_MIN` としく、 -127です。
- `std::numeric_limits<signed char>::max()` は `SCHAR_MAX` としく、 127です。
- `std::numeric_limits<unsigned char>::max()` は `UCHAR_MAX` としく、これは255です。
- `std::numeric_limits<short>::min()` は `SHRT_MIN` としく、 `SHRT_MIN` です。
- `std::numeric_limits<short>::max()` は `SHRT_MAX` としく、 32767です。
- `std::numeric_limits<unsigned short>::max()` は `USHRT_MAX` としく、 65535です。
- `std::numeric_limits<int>::min()` は `INT_MIN` としく、 `INT_MIN` です。
- `std::numeric_limits<int>::max()` は `INT_MAX` としく、 32767です。
- `std::numeric_limits<unsigned int>::max()` は `UINT_MAX` としく、 65535です。
- `std::numeric_limits<long>::min()` は `LONG_MIN` にしく、 -2147483647です。
- `std::numeric_limits<long>::max()` は `LONG_MAX` にしく、 2147483647です。
- `std::numeric_limits<unsigned long>::max()` は `ULONG_MAX` にしく、 4294967295です。

## C ++ 11

- `std::numeric_limits<long long>::min()` は `LLONG_MIN` にしく、 -9223372036854775807です。
- `std::numeric_limits<long long>::max()` は、 `LLONG_MAX` としく、 9223372036854775807です。
- `std::numeric_limits<unsigned long long>::max()` は `ULLONG_MAX` としく、これは 18446744073709551615です。

のために `T`、 `max()` しながら、のである `min()` ののしたです。には、でも、 <cfloat>ヘッダをしてCによってされるのをたすメンバがされています。

- メンバ `digits10` は、ののを `digits10` ます。
  - `std::numeric_limits<float>::digits10` は `FLT_DIG` としく、なくとも6です。
  - `std::numeric_limits<double>::digits10` は `DBL_DIG` にしく、なくとも10です。
  - `std::numeric_limits<long double>::digits10` は `LDBL_DIG` にしく、なくとも10です。
- メンバ `min_exponent10` は、ののEであり、その、Eにする10がである。
  - `std::numeric_limits<float>::min_exponent10` は `FLT_MIN_10_EXP` にしく、 -37になります。
  - `std::numeric_limits<double>::min_exponent10` は `DBL_MIN_10_EXP` にしく、 -37になります。
  - `std::numeric_limits<long double>::min_exponent10` は `LDBL_MIN_10_EXP` にしく、 -37です。
- メンバ `max_exponent10` は、パワーEにする10がであるようなEである。
  - `std::numeric_limits<float>::max_exponent10` は `FLT_MIN_10_EXP` にしく、なくとも37です。
  - `std::numeric_limits<double>::max_exponent10` は `DBL_MIN_10_EXP` にしく、なくとも37です。
  - `std::numeric_limits<long double>::max_exponent10` は `LDBL_MIN_10_EXP` にしく、なくとも37です。
- メンバ `is_iec559` がの、そのタイプはIEC 559 / IEEE 754にしているため、そのはそのによ



ってされます。

の

では、`long double`は`float`のをする`double`のをするがあります。その`long double`ことのすことができる`double`しながら、すことができる`double`のすことができる`float`すことができます。ただし、のはされています。

`T`、`std::numeric_limits<T>::radix`は、`T`のでされる`std::numeric_limits<T>::radix`します。

`std::numeric_limits<T>::is_iec559`がの、`T`のは、IEC 559 / IEEE 754でされているフォーマットの1つにします。

からきへのにオーバーフローする

きまたはなしのいずれかがきにされ、そのがでできない、されるはです。

```
// 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)
```

のとなるしたがってサイズ

スコープのないにしてになるがにされていないは、のでされます。

```
enum E {
 RED,
 GREEN,
 BLUE,
};
using T = std::underlying_type<E>::type; // implementation-defined
```

しかし、`int`と`unsigned int`がのすべてのをすことができないをいて、ではのになるが`int`よりきくならないようにしています。したがって、のコードでは、`T`は`int`、`unsigned int`、または`short`ですが、`long long`ではなく、いくつかのをげることができます。

はとしサイズ `sizeof`によってされるをつことにしてください。

オンラインでのるいをむ <https://riptutorial.com/ja/cplusplus/topic/1363>のるい

## 121: の

き

`using`は、にのでされていたのスコープにのをします。

- `typename opt` ネストされた *unqualified-id*をします。
- `::` *unqualified-id*をします。

`using`はされる`using`ディレクティブのをするときに、のにえるようにコンパイラにします。

`using`ディレクティブは、`using namespace`を`using namespace`ことからまり`using namespace`。

`use -declaration`は、`typedef`とじでのにしいをえるとはなり`typedef`。エイリアスにはがまれます。

## Examples

からにをインポートする

`using`するためにされ`cout`から`std`に`main`、`std::cout`オブジェクトをぶことができる`cout`のみ。

```
#include <iostream>
int main() {
 using std::cout;
 cout << "Hello, world!\n";
}
```

のをけるために、クラスのメンバーをします。

`use -declaration`がクラスscopeでしたは、クラスのメンバーのみをできます。たとえば、`using std::cout`はクラススコープではされません。

くの、されたは、さもなければされるです。たとえば、のコードでは、`d1.foo`は`Derived1::foo(const char*)`のみをし、コンパイルエラーがします。`Base::foo(int)`はされていません。しかし、`d2.foo(42)`が`Derived2::foo`というのエンティティのに`Base::foo(int)`をもたらすため、`d2.foo(42)`はあり`Derived2`。Name lookupは`foo`と`overload`ののをつけ、`Base::foo`をします。

```
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
}
```

## コンストラクタ

### C++ 11

なとして、クラススコープでの`using`は、クラスのコンストラクタをできます。これらのコンストラクタは、クラスによってされ、クラスのにできます。

```
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");
}
```

のコードでは、`Derived1`と`Derived2`に、を`Base`するコンストラクタにするコンストラクタがあります。`Derived1`ながら、にをう`Derived2`のでし、コンストラクタのC++ 11をして、。

オンラインでのをむ <https://riptutorial.com/ja/cplusplus/topic/9301/>の

## 122:

き

キーワード `decltype` をすると、、、またはのをできます。

## Examples

な

ここでは、このキーワードのをしています。

```
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'
```

たとえば、かがされたは、「a」として

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

するとbのがにfloatなりfloat。

もうつの

々はベクトルをとっているとしましょう

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

そして、このベクトルのイテレータをしたいといいます。らかなえは、`auto`をうことです。しかし、イテレータをするだけで、かにするはありません。たちはやります

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

しかし、`decltype`では、`intVector`がされたにになり、エラーがしにくくなります。

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

あるいは

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

2のでは、 `begin`のりをしてのをし `vector<int>::iterator`。これは `vector<int>::iterator` です。

々は `const_iterator`のがなは、々だけでするがあります `cbegin`

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

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/9930/>

## 123:

- `[[details]]`なしの
- `[[detailsarguments]]`をつ
- `__attribute` GCC / Clang / IBM
- `__declspec`のMSVC

## Examples

### [[ ]]

#### C++ 11

C++ 11は`[[noreturn]]`をしました。これは、が`return`をするか、それがであればにすることによってがびしにらないことをすためにうことができます `void`にはてはまらないことにすることがです。びしにっても、をさない。このようなは、`std::terminate`または`std::exit`びすか、をスローすることですることがあります。このようなは`longjmp`をすることですことができます。

えは、のはにをスローするか、`std::terminate`びすので、`[[noreturn]]`いです

```
[[noreturn]] void ownAssertFailureHandler(std::string message) {
 std::cerr << message << std::endl;
 if (THROW_EXCEPTION_ON_ASSERT)
 throw AssertException(std::move(message));
 std::terminate();
}
```

こののは、コードがされないことがわかっている、コンパイラが`return`なしでをできるようにします。ここでは、のコードの`ownAssertFailureHandler`でしたのびしが`ownAssertFailureHandler`れないため、コンパイラはそのびしのにコードをするはありません。

```
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]]
}
```

がにるはのなので、のことはされません。

```
[[noreturn]] void assertPositive(int number) {
 if (number >= 0)
 return;
 else
 ownAssertFailureHandler("Positive number expected"s); //< [[noreturn]]
}
```

[[noreturn]] はに void でされることにしてください。ただし、これはではなく、をプログラミングですることができます。

```
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);
```

のライブラリには、このがあります。

- std :: abort
- std :: exit
- std :: quick\_exit
- std :: unexpected
- std :: terminate
- std :: rethrow\_exception
- std :: throw\_with\_nested
- std :: nested\_exception :: rethrow\_nested

## [[fallthrough]]

### C ++ 17

switch で case がするたびに、のケースのコードがされます。こののものは、'break' ステートメントをとってぐことができます。このいわゆるフォールスルーはしないバグをきこすがあるため、いくつかのコンパイラとスタティックアナライザがをします。

C ++ 17 では、がされ、コードがちるときにがであることがされています。コンパイラは、break または [[fallthrough]] をわずにケースがし、なくとも1つのステートメントがある、をにえることができます。

```
switch(input) {
 case 2011:
 case 2014:
 case 2017:
 std::cout << "Using modern C++" << std::endl;
 [[fallthrough]]; // > No warning
 case 1998:
 case 2003:
 standard = input;
}
```

[[fallthrough]]をどのようにできるかについてのよりなについては、をしてください。

## [[deprecated]]と[[deprecated "reason"]]

### C++ 14

C++ 14では、をしてをにするなをしました。 [[deprecated]]は、がであることをすためにできます。 [[deprecated("reason")]]コンパイラができるのをすることができます。

```
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);
```

このはのにされます

- クラスの
- 
- 
- データメンバー
- 
- 
- テンプレートの

ref.c ++ 7.6.5

## [[nodiscard]]

### C++ 17

[[nodiscard]]は、びしにのりをしないことをすためにできます。りがされると、コンパイラはこれについてをすがあります。はのにできます。

- 
- Aタイプ



をにするのは、このをすすべてののにををするのとじです。

```
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::endl; // Expected: 2, Real: 1
}
```

`[[nodiscard]]`にするなは、をしてください。

`Finally / onExit`ののは、このではされています `Finally / ScopeExit`を。

## [[maybe\_unused]]

`[[maybe_unused]]`は、のロジックがされないがあることをコードですためにされます。これは、しばしば、これがされるのあるプリプロセッサにリンクされているか、またはされていないがあります。コンパイラはのにしてをすることがので、これはインテントをすることによってそれらをするです。

プロダクションでなにデバッグビルドでとされるのなは、をすりです。デバッグビルドでは、をアサートするがありますが、ではこれらのアサーションはされています。

```
[[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
```

もっとなとして、のなにあるヘルパーのがあります。コンパイルにこれらのをしないと、コンパイラによってがされることがあります。には、びしとじプリプロセッサタグをしてそれらをしたいたいのですが、になるがあるので、`[[maybe_unused]]`はよりなです。

```
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
}
```

[ [maybe\_unused] ]にするな [は、](#) をしてください。

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/5251/>

## 124: テンプレート

### Examples

のテンプレート

とモチベーション

テンプレートではETとは、にはになのをするためにされるなテンプレートメタプログラミングです。これは、ライブラリなど、さまざまなくされています。

ここでは、のコンテキストをえてみましょう。には、のみをむです。のこののは、エイリアンのもなアプリケーションであり、らはエイリアンがでどのようにするかのとてちます。

けのをてみましょう。ののをえてみましょう。

```
Vector vec_1, vec_2, vec_3;

// Initializing vec_1, vec_2 and vec_3.

Vector result = vec_1 + vec_2*vec_3;
```

ここではにするために、クラスVectorとoperation +vector plus element-wise plus operationとoperation \*ここではvector inner product element-wise operationがしくされているとします。らがどのようにすべきか、に。

ETまたはののをしないのでは、Vectorインスタスのなくとも5つのがresultをるためにわれま

1. vec\_1、vec\_2、およびvec\_3する3つのインスタス。
2. Vectorインスタス\_tmpのをす、\_tmp = vec\_2\*vec\_3;。
3. へののをにすると、result = vec\_1 + \_tmp; resultのresult = vec\_1 + \_tmp;。

ETをしたでは、2でVector \_tmpするがなくなるため、Vectorインスタスののが4つだけされます。よりいことに、よりなのをえてみましょう。

```
Vector result = vec_1 + (vec_2*vec_3 + vec_1)*(vec_2 + vec_3*vec_1);
```

vec\_1, vec\_2, vec\_3およびresultでVectorインスタスののも4つあります。いえると、ここでは、のみがまれているため、からオブジェクトがされないことがされています。

ETのみ

例えば、ののための**ET**は、

1. **PAE** の/である。はのをわず、にワークフローの/モデリングにぎません。 **PAE**は、ののまたはのモデルにすることができます。 **PAE**のインスタンスは、しばしばなコピーとみなされます。
2. レイジー ののです。のでは、ののみをむの、はのアクセスでのをできるため、オンデマンドのスキームがされます。はされませんにアクセス/ねられたにります。

したがって、こので**ET**をにするはすぐきましょう。

にのコードスニペットをえてみましょう

```
Vector vec_1, vec_2, vec_3;

// Initializing vec_1, vec_2 and vec_3.

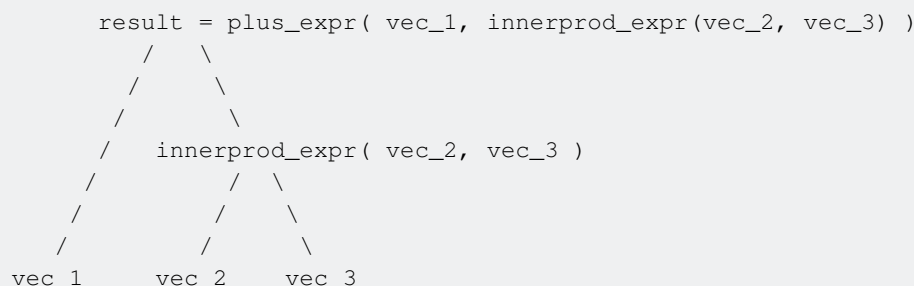
Vector result = vec_1 + vec_2*vec_3;
```

をするは、さらに2つのにできます。

1. ベクトルプラス **plus\_expr**と
2. ベクトル **innerprod\_expr**と。

**ET**のやりはのとおりで

- サブをすぐにするわりに、 **ET**はにグラフィカルなをってをモデルします。グラフのノードは**PAE**をします。ノードのエッジは、のフローをします。したがって、のの、のグラフがられます。



- なは、 グラフのをることによってされるここで、々はののをっているので、インデックスきの`result`、してうことができます。な`result`し、エレメントにすることができます`result`ののな。つまり、 `result elem_res`のは、 `vec_1 elem_1`、 `vec_2 elem_2`、 `vec_3 elem_3` のするをってのようになります。

```
elem_res = elem_1 + elem_2*elem_3;
```

したがって、ののをするためにな`Vector`をするはありません。つまり、1つののをすべてし、アクセスでエンコードすることができます。

のコードをします。

ファイル **vec.hhstd :: vector** のラッパー。かびされたときにログをするためにされます。

```
#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

```

## ファイル **expr.hh** ののためのテンプレートのベクトルプラスとベクトル

セクションにしましょう。

1. セクション1は、すべてののクラスをします。 **Curiously Recurring Template Pattern** **CRTP** をしています。
2. セクション2はの**PAE**をしています。これは、のをむデータのなるラッパー—constです。
3. セクション3は、2の**PAE binary\_operation**をします。これは、でvector\_plusとvector\_innerprodでされるクラステンプレートです。のタイプ、の**PAE**、の**PAE**によってパラメータされています。のは、インデックスきアクセスでエンコードされます。
4. セクション4では、vector\_plusおよびvector\_innerprodをごとのとしてしています。また、**PAE**の+と\*をオーバーロードします。この2つの**PAE**をします。

```

#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

```

ファイル **main.cc** テスト **src** ファイル

```
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;
}

```

GCC 5.3をして `-O3 -std=c++14` コンパイルすると、のようにならね `-O3 -std=c++14`。

```
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> > > >
```

はのとおりです。

- **ET**をすると、これはパフォーマンスがにします > 3。
- な `Vector` オブジェクトのはです。 **ET**のとに、`ctor`はだけびされます。
- `Boost::demangle`は、のETのりのをするためにされました。

---

ドローバックと

- 
- **ET**のらかなは、のさ、コードメンテナンスのしさです。ののみがされているのでは、に  
は、すべてのでよりながし、のがもはやされていないではなく、すでになのがまれています  
たとえば、はになります。
  - **ET**をするのもうつの、`auto`キーワードでうまくいくことです。でべたように、**PAE**はに  
プロキシですそして、プロキシはに`auto`でうまくいきません。のをえてみましょう。

```
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.
```

ここで、`for`ループのでは、されたのわりにグラフが`for`ループにされるため、はされます。

---

**ET**をしているのライブラリ

- 
- **boost::proto**はあなたののためののルールとをし、**ET**をつてできるなライブラリです。
  - **Eigen**はのライブラリで、**ET**をつてにさまざまなをしています。

## テンプレートをすな

エクスプレッションテンプレートは、にでされるコンパイルのです。なは、ななをし、パスをしてループをすることです、でをする。テンプレートは、に、のArrayまたはMatrixをするのなのオーバーロードのをするためにされました。Bjarne Stroustrupは、のバージョン「The C++ Programming Language」の「」とぶテンプレートののをしました。

にテンプレートにるに、なぜそれらをとするのかしておくがあります。これをするために、にすになMatrixクラスをえてみましょう

```
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;
}
```

のクラスをとすると、のようなをくことができるようになりました。

```
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
```

にしたように、`operator+()`をオーバーロードすることは、のなをするをします。

なことに、のは、の「りの」バージョンにべてにです。

をするには、`Matrix d = a + b + c`ようなをくときにがこるかをするがあります。には、`((a + b) + c)`または`operator+(operator+(a, b), c)`ます。いえると、`operator+()`のループは2されますが、1のパスでにできました。この、2つのなファイルがされ、パフォーマンスがさらにします。には、なにをするをすることで、`Matrix`クラスをににしました。

たとえば、のオーバーロードがなければ、のパスをしてはるかになをできます。

```
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;
}
```

これまでのでは、`Matrix`クラスのためにはるかになインターフェースをするため、`Matrix::add2()`、`Matrix::AddMultiply()`などのメソッドをするがあるため、のがあります。

わりにみんなで、オペレータのオーバーロードをよりなでするをてみましょう

このは、のツリーをするに、`Matrix d = a + b + c`があまりにも「に」されるというにしています。いえれば、にしたいのは、1のパスで`a + b + c`を`a + b + c`ことです。には、としてられるを`d`にりてるがあるにります。

これは、テンプレートのにあるコアアイデアです。`operator+()` 2つの`Matrix`インスタンスをしたをちにするのではなく、ツリーがされた、ののために「テンプレート」をします。

たとえば、の2つのタイプのにするテンプレートのがえられます。

```
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;
};

```

そしてここに `operator+()` があり `operator+()`

```

template <typename LHS, typename RHS>
MatrixSum<LHS, RHS> operator+(const LHS& lhs, const RHS& rhs) {
 return MatrixSum<LHS, RHS>(lhs, rhs);
}

```

このように、`operator+()` は、2つのMatrixインスタンスのMatrixインスタンスをしたの "な" をすのではなく、をすテンプレートをしします。にめておくべきもなポイントは、がまだされていないということです。そのオペランドへのをすだけです。

、のように `MatrixSum<>` テンプレートのインスタンスをするものはありません。

```

MatrixSum<Matrix<double>, Matrix<double> > SumAB(a, b);

```

しかし、ので、にのがなときは、のように `d = a + b` をすことができます。

```

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);
 }
}

```

このように、テンプレートをするもう1つのは、には `a` と `b` をし、1のパスで `d` にりてることです。

また、のテンプレートをみわせることをめるものはありません。たとえば、`a + b + c` はのテンプレートになります。

```

MatrixSum<MatrixSum<Matrix<double>, Matrix<double> >, Matrix<double> > SumABC(SumAB, c);

```

ここでもまた、1のパスをしてをすことができます。

```

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);
 }
}

```

に、パズルののは、にテンプレートを `Matrix` クラスにすることです。これはに `Matrix::operator=()` をすることによってされています。これは、テンプレートをととしてとり、に "で" ったように、1のパスでしします。

```

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;
};

```

オンラインでテンプレートをむ <https://riptutorial.com/ja/cplusplus/topic/3404/テンプレート>

## 125: のりみ

Foldは、のでサポートされています

|    |    |    |    |    |    |      |   |     |     |   |     |  |
|----|----|----|----|----|----|------|---|-----|-----|---|-----|--|
|    |    |    |    |    |    |      |   |     |     |   |     |  |
| +  | -  | *  | /  |    | \  | そして、 |   | <<  | >>  |   |     |  |
| += | -= | *= | /= | =  | \= | =    | = | <<= | >>= | = |     |  |
| == | =  | <  | >  | <= | >= | &&   |   | ,   | 。   | * | ->* |  |

のシーケンスをりたたむときは、の3つのをいて、りまれたがされません。

| オペレーター | パラメータパックがの |
|--------|------------|
| &&     |            |
|        |            |
| ,      | void       |

## Examples

り

りみは、のをしてパラメータパックをりむためにされ**ます**。りみには2あります。

- **Unary Left Fold** (`... op pack`) はのようにされます。

```
((Pack1 op Pack2) op ...) op PackN
```

- **フォールド** (`pack op ...`) はのようにされます。

```
Pack1 op (... (Pack(N-1) op PackN))
```

ここにがあります

```
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

```

## バイナリフォールド

バイナリフォールドはに**フォールド**で、ながあります。

バイナリフォールドには2あります。

- バイナリフォールド- (value op ... op pack) -のようにされます。

```
((Value op Pack1) op Pack2) op ... op PackN
```

- バイナリフォールド (pack op ... op value) -のようにされます。

```
Pack1 op (... op (Pack(N-1) op (PackN op Value)))
```

にをします。

```

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

```

## コンマでる

パラメータパックのにしてのをするがあるのはなです。C++ 11では、できることはのとおりです。

```

template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
 using expander = int[];
 (void)expander{0,
 (void(os << args), 0)...
 };
}

```

しかし、をにすると、のようにうまくされます

```

template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
 (void(os << args), ...);
}

```

のではありません。

オンラインでのりみをむ <https://riptutorial.com/ja/cplusplus/topic/2676/のりみ>

## Examples

どのがつかったか

は、に、 $T$ じて、のうちの1つをむ「クラス」および「する」のをすることによってされます。まず、クラス、 $T$ 、およびクラステンプレートののをします。

- $T$ がネストされたクラスであれば、メンバーの、そののクラス。
- は $T$ ですそれはまた、クラスのメンバかもしれません、そののもの。
- は $T$ クラスであるそれはまた、れにすることができる、そのすべてのクラスとクラスを。するすべてのクラスののもの。
- $T$ が`ClassTemplate<TemplateArguments>` これもクラスですの、テンプレートにけられたクラスと、テンプレートテンプレートの、テンプレートテンプレートのクラスです。メンバーテンプレート

や、みみにもいくつかのがあります

- $T$ へのポインタである $U$ またはアレイ $U$ 、クラス、がけられた $U$ 。 `void (*fptr)(A); f(fptr); void(A)` するとクラスをみ`void(A)`のをしてください。
- $T$ がの、パラメータとりのにするクラスと。 `void(A)` には $A$ けられたとクラスがまれます。
- $T$ がメンバへのポインタである、メンバにするクラスとメンバへのポインタとデータメンバへのポインタのにされるかもしれない。 `BA::*p; void (A::*pf)(B); f(p); f(pf); A、 B、 void(B)` するとクラスがまれていますにしてのきをしています。

するすべてのネームスペースのすべてのとテンプレートは、ルックアップによってされます。さらに、するクラスでされた`namespace-scope`フレンドがつかります。はされません。ただし、ディレクティブのはされます。

のすべてのびしは、びしので $f$ をすることなく、です。

```
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!
```

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/5163/>

## 127: り

メンバ<sub>m</sub>パラメータまたはりのは、<sub>T</sub>がクラスの<sub>m</sub>のオーバーライドでよりになるです。に、<sub>T</sub>、<sub>m</sub>するクラスとし `co` でがする。 `C++`は、ポインタまたはのであるリターンのサポートをします。はpointeeまたはreferentです。

のりは`C++`ののな **out-argument**であり、はなout-argumentののなため、`C++`のサポートはりにされています。さもなければ、びしコードは、コードがしていないよりでないのオブジェクトをすることができます。MITのバーバラ・リスコフは、これとするをしました。、Liskov Substitution Principle **LSP**とばれています。

サポートは、にダウンキャストिंगとなタイプのチェックをけるのにちます。

スマートポインタはクラスであるため、スマートポインタのにしてにのビルトインサポートをすることはできませんが、ポインタをする`virtual`の`virtual`スマートポインタラッパーをできます。

## Examples

### 1. リターンのないは、それがましいをしています。

```
// 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. のバージョン、。

```

// 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. のスマートポインタクリーンアップ。

```

// 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.
}
```

オンラインでりをむ <https://riptutorial.com/ja/cplusplus/topic/5411/> [リ](#)



## 128: な

き

は、どののキャストがされているかにじて、`dynamic_cast<T>`、`static_cast<T>`、`static_cast<T>`、`reinterpret_cast<T>`、または`const_cast<T>`をしてにしたり、`T` キャストできます。

C++は、のキャスト `T(expr)` とCスタイルキャスト `(T)expr` もサポートしています。

- `()`
- `()` リスト
- *braced-init-list*
- *typename-specifier* `()`
- `()` リスト
- *typename-specifier braced-init-list*
- `dynamic_cast < type-id > ()`
- `static_cast < type-id > ()`
- `reinterpret_cast < type-id > ()`
- `const_cast < type-id > ()`
- `(-id)` キャスト

6つのキャストにはするものが1つあります。

- `dynamic_cast<Derived&>(base)` ようににキャストすると、がられます。したがって、じオブジェクトでかをしたいが、それをのとしてうは、にキャストします。
- `static_cast<string&&>(s)` ように、`static_cast<string&&>(s)` にキャストすると、`static_cast<string&&>(s)` れます。
- `(int)x` ように、にキャストすると、キャストされるのコピーとしてえられるが、のものとはなるのプリがされます。

`reinterpret_cast` キーワードは、2の「でない」コンバージョンをします。

- 1つのタイプのメモリにアクセスするためにできる「[タイプ・ペニング](#)」。なるタイプのものです。
- どちらのでも、[とポインタの](#)。

`static_cast` キーワードは、さまざまななるコンバージョンをできます。

- [ベースから](#) コンバージョンへの
- のとなコンストラクタまたはをびすのをむ、によってできる。は[こちら](#)と[こちら](#)をごくたさい。
- `void`のは、のをします。

```
// on some compilers, suppresses warning about x being unused
static_cast<void>(x);
```

- との、およびなる。 [を](#)
- クラスのメンバへのポインタから、クラスへのポインタへ。されたりするがあります。 [メンバへのポインタのためのからへのを](#)
- [void\\*からT\\* void\\*へ](#)

## C++ 11

- `std::move`のように、からxにします。 [セマンティクスをしてください](#)。

## Examples

### ベースからした

クラスへのポインタは、`static_cast`をしてクラスへのポインタにできます。`static_cast`はのチェックをわす、ポインタがになをしていないときはのにつながるがあります。

```
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
```

に、クラスへののは、`static_cast`をしてクラスへのにできます。

```
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
```

ソースタイプが`dynamic_cast`フィックである、`dynamic_cast`をして、ベースからしたをできます。ランタイムチェックをし、のをするのではなく、をできます。ポインタの、にヌルポインタがされます。の、`std::bad_cast` または `std::bad_cast` からしたクラスのにがスローされます。

```
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
```

## コンスタンスをてる

constオブジェクトへのポインタは、`const_cast` キーワードをして、constオブジェクトへのポインタにできます。ここでは`const_cast`をして、const-correctでないをびします。それは、たとえばそれがポインタをとってきまなくても、`const char*`だけをけけます

```
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` からへのは、constされたをconstされていないにするためにできます。

`const_cast` は、C++システムがconstオブジェクトをしようとするのをぐことができないため、です。そうすると、がになります。

```
const int x = 123;
int& mutable_x = const_cast<int&>(x);
mutable_x = 456; // may compile, but produces *undefined behavior*
```

## のタイプ

オブジェクトへのポインタそれぞれをして、のオブジェクトへのポインタそれぞれにすることが  
できる`reinterpret_cast`。これはコンストラクタやをびさない。

```
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

`reinterpret_cast`のは、アドレスがタイプにしてにされていれば、オペランドとじアドレスをします。それの、はです。

```
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

`reinterpret_cast`のはされていませんが、デスティネーションタイプのアラインメントがソースタイプよりもでないり、ポインタはソースタイプからデスティネーションタイプへのにえます。

```
int x = 123;
```

```
unsigned int& r1 = reinterpret_cast<unsigned int&>(x);
int& r2 = reinterpret_cast<int&>(r1);
r2 = 456; // sets x to 456
```

ほとんどのでは、`reinterpret_cast`はアドレスをしませんが、これはC++ 11までされていませんでした。

`reinterpret_cast`は、1つのポインタからデータメンバへのからのへのポインタ、または1つのメンバへのポインタからのものへのにもできます。

`reinterpret_cast`みりまたはポインタまたはをしてきみをしてられたので、であるとえられる  
`reinterpret_cast`とタイプはであるのをトリガすることができます。

## ポインタとのの

むオブジェクトポインタ`void*`またはポインタをしてにすることができる`reinterpret_cast`。これは、タイプがににのみコンパイルされます。はインプリメンテーションによってされ、ポインタがすメモリのバイトのアドレスをします。

、`long`または`unsigned long`はポインタをするのにな`unsigned long`ですが、これはによってされていません。

## C++ 11

`std::intptr_t`と`std::uintptr_t`がするは、`void*`したがってオブジェクトへのポインタをするのになさがされます。ただし、ポインタをするのにいことはされていません。

に、`reinterpret_cast`をとってをポインタにすることができます。このも、はインプリメンテーションですが、ポインタのはのによってされないことがされています。では、ゼロがヌルポインタにされることはされません。

```
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...
```

## なコンストラクタまたはによる

なコンストラクタまたはをびすことをうは、にうことはできません。`static_cast`をしてにをすることができます。はなものであることをいて、のとじです。

```
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
}
```

の

`static_cast`はのができます。このような`static_cast`は、ののようになががあります。

- をにすとき、"される"はにられていないので、のはわれません。

```
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);
```

ながなければ、`double`オブジェクトがにされ、のがします。

- クラスは、のようにクラスをびすことができます。

```
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;
 }
};
```

`static_cast`は、またはからスコープまたはスコープなしにできます。もです。のでもあります。

- スコープのないからへののはのです。 `static_cast`をすることはですが、ではありません。

## C++ 11

- スコープきをにすると、のようになります。
  - のがタイプでにできる、はそのになります。
  - そうでない、タイプがの、はです。
  - それの、タイプがタイプの、はタイプへのののとじになり、にタイプへのののとじになります。

```
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

```

- またはをにすると、のようになります。
  - のがenumのにある、はそのになります。このはすべてのとしくないかもしれないことにしてください。
  - それの、は<= C ++ 14または> = C ++ 17になります。

```

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

- をにすると、はのにされた、にされるのとじになります。

```

enum Direction {
 UP = 0,
 LEFT = 1,
 DOWN = 2,
 RIGHT = 3,
};
Direction d = static_cast<Direction>(3.14); // d is RIGHT

```

## メンバーへのポインタの

クラスのメンバへのポインタは、`static_cast` をってクラスのメンバへのポインタにできます。されたはするがあります。

オペランドがメンバへのヌルポインタである、はメンバへのヌルポインタにもなります。

それの、は、オペランドによってされたメンバがにデスティネーションクラスにする、またはデスティネーションクラスが、オペランドがすメンバをむクラスのベースクラスまたはクラスであるにのみです。`static_cast` はをチェックしません。がでない、はです。

```

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 \*からT \*

C++では、`void*`をに`T*`にすることはできません。ここで、`T`はオブジェクトです。わりに`static_cast`をしてにをするがあります。オペランドがに`T`オブジェクトをす、はそのオブジェクトをします。その、はです。

### C++ 11

オペランドが`T`オブジェクトをしていなくても、オペランドがタイプ`T`にしてアドレスがしくされているバイトをしているり、のはじバイトをします。

```

// 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

```

## Cスタイルキャスト

Cスタイルのキャストは「ベストエフォート」のキャストとなすことができ、Cでできるキャストというがけられてい`(NewType) variable`。このキャストのは`(NewType) variable`。

このキャストがされるたびに、のいずれかのC++キャストがにされます。

- `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))`

キャストはにしていますが、のとしていくつかのがあります`NewType(expression)`。その、スペースをたないだけがキャストできます。

しいC++キャストをするがいです。なぜなら、C++ソースコードのどこからでもみやすく、どこでもにつけすことができ、ランタイムではなくコンパイルにエラーがされるからです。

このキャストはしない `reinterpret_cast` をきこすがあるため、しばしばとみなされます。

オンラインでなをむ <https://riptutorial.com/ja/cplusplus/topic/3090/>な



## 129:

き

コンパイルするとき、コンパイラはプログラムをしてパフォーマンスをさせることがよくあります。これは、[if-ifルール](#)によってされ、なるいをしないすべてのをします。

## Examples

インライン/インライン

インラインインラインともばれるは、のびしをそののできえるコンパイラのです。これにより、びしのオーバーヘッドはされますが、がされるがあるため、をにしてみします。

```
// 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()
}
```

インラインは、びしオーバーヘッドがのサイズにべてな、さなにしてみにわれます。

の

がの`class`つまり、でないデータメンバをたない`class`または`struct`であっても、オブジェクトまたはメンバサブオブジェクトのサイズは、なくとも1にするがあります。じタイプのなるオブジェクトのアドレスはにされます。

ただし、`class`サブオブジェクトはがなく、オブジェクトレイアウトからにすることができます。

```
#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));
}
```

**allocator-aware** ライブラリクラス `std::vector`、`std::function`、`std::shared_ptr` などはのをに  
して、アロケータがステートレスであればそのアロケータメンバのストレージをしないようにしま  
す。これは、なデータメンバーの1つえば、`vector` `begin`、`end`、または`capacity`ポインタをする  
ことによってされます。

リファレンス [cppreference](#)

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/9767/>

## 130: の

き

ビヘイビアUBとはですか ISO C ++ §1.3.24、N4296によれば、「このがをさない」です。

これは、プログラムがUBにしたときに、それがむものをでもすることがされることをする。これはしばしばクラッシュをしますが、にもしない、[のをばす](#)、またはしくするようにえるかもしれません

うまでもなく、UBをびすコードをくのはけるべきです。

プログラムにのビヘイビアがまれている、C ++ではビヘイビアにはありません。

- がしたとおりにするようにえるかもしれませんが、クラッシュしたり、なをじるがあります。
- は、じプログラムのでなるがあります。
- されていないをむのにるをめ、プログラムのどのもするがあります。
- では、ののをするはありません。

は、にのてのをするのをすることがありますが、そのようなにするプログラムはがありません。

のがする

には、のるいは、ハンドラのようなエラーをうまくうことができないため、いことです。

しかし、いくつかのをしないままにしておくことは、には、C ++の "あなたがしていないものをわない" というのなです。のにより、コンパイラはががをしているかをり、のでされているりをチェックするためのコードをしないとすることができます。

されていないのと

いくつかのツールをして、にのてをすることができます。

- ほとんどのコンパイラでは、コンパイルにのがすることをするフラグがあります。
- gccとclangのしいバージョンには、にパフォーマンスコストでのをチェックする、いわゆる "Undefined Behavior Sanitizer" フラグ `-fsanitize=undefined` がまれています。
- `lint` ようなツールは、よりなのをすることがあります。

[されていない](#)、[されていない](#)、[のるい](#)

C ++ 14 ISO / IEC 14882:2014 セクション 1.9 プログラムから

1. こののは、パラメータされたをす。[カット]

2. マシンののは、こので えば `sizeof(int)` としてされている。これらはのパラメータをする。には、これらのでとをしたドキュメントがまれていなければなりません。[カット]
3. のあるののおよびは、このではのものとしてされているえば、りりがメモリりてにしたにしいでのの。であれば、このはのされるをする。これらはのをする。したがって、のインスタンスは、のプログラムおよびのにしてのなをすることがでできる。
4. ののは、このではされていないとされていますまたは、えば、 `const` オブジェクトをしようとする。[ このは、のをむプログラムのにはもしない。 - ノート ]

## Examples

### ヌルポインタによるみき

```
int *ptr = nullptr;
*ptr = 1; // Undefined behavior
```

これは、nullポインタがなオブジェクトをしていないため、`*ptr`にきむオブジェクトがないため、のです。

これはもにセグメンテーションフォルトをきこしますが、それはでかがこります。

### voidのりのをつのreturnがありません

りのがvoidでないでreturnをすると、のになります。

```
int function() {
 // Missing return statement
}

int main() {
 function(); //Undefined Behavior
}
```

ほとんどのコンパイラはこのののためにコンパイルにをします。

ルールにするのはmainです。main returnがない、コンパイラはにreturn 0;しreturn 0;あなたのために、それはにすることができます。

### リテラルの

## C ++ 11

```
char *str = "hello world";
```

```
str[0] = 'H';
```

"hello world"はリテラルなので、それをするとのがられます。

の`str`のは、C++ 03でにされましたのバージョンのからの。2003のくのコンパイラは、これにするをすかもしれませんかえ、わしい。2003、コンパイラは、のについてします。

## C++ 11

のはであり、C++ 11ではコンパイラとなります。のは、のようなのをにすることによってのるいをするようにすることができます。

```
char *str = const_cast<char *>("hello world");
str[0] = 'H';
```

## インデックスへのアクセス

またはそのためのライブラリコンテナは、すべてのをしてされているためののインデックスにアクセスすることは、ののです。

```
int array[] = {1, 2, 3, 4, 5};
array[5] = 0; // Undefined behavior
```

のこのは`array + 5`をすポインタをつことはされていますが、なではないため、できません。

```
const int *end = array + 5; // Pointer to one past the last index
for (int *p = array; p != end; ++p)
 // Do something with `p`
```

に、ポインタをすることはできません。ポインタは、のをしていなければなりません。

## ゼロによる

```
int x = 5 / 0; // Undefined behavior
```

0によるはにされていないので、これはのるいであることがにかなっています。

しかしながら

```
float x = 5.0f / 0.0f; // x is +infinity
```

ほとんどのでは、NaN が0.0f、infinity かのまたは-infinity かのをすためにゼロによるをするIEEE-754がされています。

## き オーバーフロー

```
int x = INT_MAX + 1;

// x can be anything -> Undefined behavior
```

のにがにされていない、またはそののなのにない、そのはです。

C++ 11/4

これは、でなるいをもたらすので、がされたるいにきくするようにされるがあるので、よりなもの1つです。

```
unsigned int x = UINT_MAX + 1;

// x is 0
```

、よくされています

されていないとされたなしは、 $2^n$ とするのにわなければならない。ここで、 $n$ はののビットである。

C++ 11パラグラフ3.9.1/4

コンパイラがのをしてすることがある

```
signed int x ;
if(x > x + 1)
{
 //do something
}
```

ここでは、きのオーバーフローがされていないため、コンパイラはしてこらないとすることができ、したがって、ifブロックをすることができます

されていないローカルの

```
int a;
std::cout << a; // Undefined behavior!
```

これはされていないため $a$ のになります。

これは、が「」であるか、または「そのにそのメモリにあった」であることが、ってされることがよくあります。しかし、のでは、のをえるのは $a$ のにアクセスするです。には、これは「ガベージ」のがなですが、これはのの1つのなにぎません。

のハードウェアサポートにしているのではそうはえにくい、のコードサンプルをコンパイルするときにコンパイラはプログラマにもじようにをすることができる。このようなコンパイラとハードウェアのサポートによって、ビヘイビアにするこのようなは、ビヘイビアののなりピングプログラマのをしくめます。

## C++ 14

`unsigned char`の`unsigned char`をしても、をのようにするとはしません。

- の2または3オペランド;
- ビルトインカンマのオペランド。
- `unsigned char`へのオペランド。
- オペランドも`unsigned char`のはのオペランド。
- `unsigned char`オブジェクトの。

またはがされた。そのような、`indeterminate`は、であれば、のににします。

`static`はにゼロでされますな。

```
static int a;
std::cout << a; // Defined behavior, 'a' is 0
```

のではない**1**つのルール

テンプレートのクラス、、インライン、テンプレート、またはメンバがリンケージをち、のでされている、すべてののがであるか、または**1つのルールODR**にってがされていません。

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
}
```

のプログラムは、リンクをつクラス::Foo 2つのをなるとでいるため、のをしますが、2つのではありません。じのクラスのとはなり、このはコンパイラによってされるはありません。

メモリりてとりてのペアがしくありません

オブジェクトは、newによってりてられていてでないにのみ、deleteによってりてをできます。deleteのがnewによってされなかった、またはの、はです。

オブジェクトは、newによってりてられ、であるにのみ、delete[]によってりてをできます。delete[]がnewによってされなかったか、またはでない、はです。

freeへのがmallocによってされなかった、はです。

```
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
```

このようなはにすることによりすることができmallocとfreeのにライブラリスマートポインタをむ、C++プログラムでnewおよびdelete、およびむstd::vectorとstd::stringのにnewとdelete[]。

オブジェクトをったとしてアクセスする

ほとんどの、あるタイプのオブジェクトにアクセスするのはうタイプですcv-qualifierをして。

```
float x = 42;
int y = reinterpret_cast<int&>(x);
```

はのです。

このなエイリアシングにはいくつかのがあります。

- ・クラスのオブジェクトは、のクラスのクラスであるかのようにアクセスできます。
- ・どんなでもcharまたはunsigned charとしてアクセスできますが、そのではありません。  
charはあたかもものであるかのようにアクセスできません。
- ・きは、するなしとしてアクセスでき、そのもです。

するは、メンバがにのクラスまたはクラスとじをたないオブジェクトにしてびされた、のがするということです。がオブジェクトにアクセスしなくても、これはてはまります。



```

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

```

## のオーバーフロー

をするが、ののなのにないをする、そのはC++にとってですが、マシンがしているのあるのによってされ、IEEE 754のようなものである。

```

float x = 1.0;
for (int i = 0; i < 10000; i++) {
 x *= 10.0; // will probably overflow eventually; undefined behavior
}

```

## コンストラクタまたはデストラクタからなメンバをびす

### スタンダード10.4

メンバーは、クラスのコンストラクタまたはデストラクタからびすことができます。そのようなコンストラクタまたはデストラクタからまたはされるオブジェクトにして、またはにびしをびし10.3することのはされていません。

よりには、Scott MeyersなどのC++は、コンストラクタやdstructorsからなものでもをびすことはしてありません。

のリンクからされたのをえてみましょう。

```

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 */ }
};

```

sell\_transactionオブジェクトをするsell\_transactionます。

```

sell_transaction s;

```

これは、`transaction`のコンストラクタをにびす`sell_transaction`のコンストラクタをにびします。しかし、`transaction`のコンストラクタがびされるとき、オブジェクトはまだタイプ`sell_transaction`ではなく、タイプの`transaction`だけです。

したがって、`transaction::transaction()` `log_it`への`log_it`は、なことではないようにわれます。つまり、`sell_transaction::log_it`びします。

- こののように`log_it`がの、そのはです。
- `log_it`がであれば、`transaction::log_it`がびされます。

デストラクタをたないクラスへのポインタをしてオブジェクトをする。

```
class base { };
class derived: public base { };

int main() {
 base* p = new derived();
 delete p; // The is undefined behavior!
}
```

section [expr.delete]§5.3.5/ 3では、が`virtual`デストラクタをたないオブジェクトにして`delete`がびされた、

されるオブジェクトのがそのとなる、は、されるオブジェクトののクラスでなければならず、はをつか、はです。

これは、クラスがクラスにデータメンバーをしたかどうかにかかわらず、ケースです。

ダングリグリファレンスへのアクセス

になったオブジェクトやのでされたオブジェクトへののにアクセスすることはです。このようなは、もはやなオブジェクトをしていないので、ぶらがっているとわれています。

```
#include <iostream>
int& getX() {
 int x = 42;
 return x;
}
int main() {
 int& r = getX();
 std::cout << r << "\n";
}
```

このでは、`getX`がるときにローカル`x`がスコープかられます。のは、それがされているブロックのスコープをえてローカルのをばすことはできないことにしてください。したがって、`r`はなです。してするようにえるかもしれないが、このプログラムでは、のをしている<sup>42</sup>いくつかのケースでは。

std または posix の

17.6.4.2.1 / 1 は、に std をすることをしています。

または std または std のにする、C++ プログラムのはにしないり、されていません。

じことが posix 17.6.4.2.2 / 1 にもてはまり posix

または posix または posix のにしないりすると、C++ プログラムのはです。

のをしてください。

```
#include <algorithm>

namespace std
{
 int foo(){}
}
```

では、じをする algorithm またはそれにまれるヘッダの1つをじているわけではないので、このコードは [One Definition Rule](#) にします。

したがって、に、これはされています。ただし、 [のはされています](#)。おそらくもなのは、ユーザーのをできることです。したがって、たとえば、あなたのコードが

```
class foo
{
 // Stuff
};
```

それでは、はうまくいきます

```
namespace std
{
 template<>
 struct hash<foo>
 {
 public:
 size_t operator()(const foo &f) const;
 };
}
```

へのまたはからの

に

- からへの、
- からへの、または
- タイプからよりいタイプまで、

ソースがタイプでできるのの、はのになります。

```
double x = 1e100;
int y = x; // int probably cannot hold numbers that large, so this is UB
```

ベースからしたキャストがです

`static_cast`をしてクラスへのポインタをクラスへのポインタにするが、クラスのオブジェクトをオペランドがしない、されていません。 [のベース](#)をしてください。

ポインタによるびし

ポインタをしてをびすには、ポインタのがのとにしていなければなりません。そのの、はです。

```
int f();
void (*p)() = reinterpret_cast<void(*)>(f);
p(); // undefined
```

**const** オブジェクトの

`const` オブジェクトをしようとする、のになります。これがされる `const` のメンバー `const` オブジェクト、およびクラスのメンバーは、`const`。ただし、`mutable` のメンバー `const` オブジェクトがない `const`。

そのようなみは `const_cast` をして `const_cast` ことができます

```
const int x = 123;
const_cast<int&>(x) = 456;
std::cout << x << '\n';
```

コンパイラは、`const int` オブジェクトのをインラインするため、このコードはコンパイルして `123` をするがあります。コンパイラは、`const` オブジェクトのをみりメモリにすることもできるため、セグメンテーションフォルトがするがあります。いずれのでも、はであり、プログラムはかをうがあります。

のプログラムははるかになエラーをします

```
#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';
}

```

このコードでは、`getFoo`は`const Foo`のシングルトンをし、メンバー`m_x`は123されます。その`do_evil`びされ、の`foo.m_x`らかになっていた456にされましたか

そのにもかかわらず、`do_evil`はにいいことはもしません。それは`Foo*`をしてセッターをびすことだけです。しかし、`const_cast`がされていなくても、そのポインタは`const Foo`オブジェクトをしています。このポインタは、`Foo`のコンストラクタによってされました。`const`オブジェクトはなりません`const`がするまで、ので`this`をつ`Foo*`、ない`const Foo*`コンストラクタ、。

したがって、このプログラムにはらかにながらないにもかかわらず、のがします。

### メンバーへのポインタをしてしないメンバーにアクセスする

オブジェクトへのポインタをしてオブジェクトのメンバにアクセスするとき、オブジェクトがにポインタによってされるメンバをまない、そのはです。メンバへのそのようなポインタは`static_cast`でできます。

```

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

```

### メンバーへのポインタにするからへのがしくありません

`static_cast`をして`TD::*`を`TB::*`にすると、されるメンバーは、クラスまたは`B`クラスであるクラスにしているがあります。そのの、はです。 [メンバーへのポインタのためのを](#)

### なポインタ

のポインタをすると、のがします。

- がポインタ・オペランドとじオブジェクトにさないは、のまたは。ここでは、の1はまだにしているとされます。

```
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]
```

- がじオブジェクトにしていない、2つのポインタの。やはり、の1はにするとされます。は2つのヌルポインタをして0をすることです。

```
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
```

- がstd::ptrdiff\_tオーバーフローしたの2つのポインタの。
- いずれかのオペランドのpointeeが、cv-qualificationをしてされたオブジェクトのとしないポインタ。によると、"に、にクラスのオブジェクトがまれている、ポインタのにクラスへのポインタをできません。"

```
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
```

## なポジションのシフト

ビルトインシフトの、オペランドはではなく、されたオペランドのビットよりもにさくなければなりません。その、はです。

```
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
```

## [noreturn]から

### C++ 11

#### からの[dcl.attr.noreturn]

```
[[noreturn]] void f() {
 throw "error"; // OK
}
[[noreturn]] void g(int i) { // behavior is undefined if called with an argument <= 0
 if (i > 0)
 throw "positive";
}
```

## にされているオブジェクトをする

ここでは、でにされるオブジェクトにしてデストラクタがにびされます。

```
struct S {
 ~S() { std::cout << "destroying S\n"; }
};
int main() {
 S s;
 s.~S();
} // UB: s destroyed a second time here
```

のは、`std::unique_ptr<T>`がまたはなをつ`T`をすようにされたときにします。

```
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
```

オブジェクトを2するもう1つののは、2つの`shared_ptr`をいにすることなくオブジェクトをすることです。

```
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);
```

のテンプレート

からの[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 ...
};
```

オンラインでのをむ <https://riptutorial.com/ja/cplusplus/topic/1812/>の



## 131: のりの

- *function\_name* [ *function\_args* ] [ *function\_attributes* ] [ *function\_qualifiers* ] -> *trailing-return-type* [ *requires\_clause* ]

のは、ののオプションをすなしののはリストのようにをして、をしたなをしています。

さらに、のりののは、のりののでクラス、、またはをすることをしますのりのはされません。それのは、->にじようにをることができます。

## Examples

ネストされたのをける

```
class ClassWithAReallyLongName {
public:
 class Iterator { /* ... */ };
 Iterator end();
};
```

のりのでメンバ<sub>end</sub>をする

```
auto ClassWithAReallyLongName::end() -> Iterator { return Iterator(); }
```

のりのをたないメンバ<sub>end</sub>をする

```
ClassWithAReallyLongName::Iterator ClassWithAReallyLongName::end() { return Iterator(); }
```

のりのは、クラスのスコープでされますが、するりのは、むのスコープでされ、したがって、「」なをとするがあります。

ラムダ

ラムダはのりしかてません。のリターンはラムダにはされません。くの、ラムダのりのをまったくするはないことにしてください。

```
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) { ... };
```

オンラインでのりのをむ <https://riptutorial.com/ja/cplusplus/topic/4142/のりの>

## 132: り

### Examples

デジットセパレータ

のリテラルはみにくい。

- 7237498123をする
- かどうかについては237498123と237499123をしてください。
- 237499123または20249472がきいかどうかをします。

C++14<sup>なを</sup>とユーザリテラルで、りとして。これにより、のがのをしやすくなります。

C++ 14

```
long long decn = 1'000'000'00011;
long long hexn = 0xFFFF'FFFF11;
long long octn = 00'23'0011;
long long binn = 0b1010'001111;
```

は、そのをするときにされます。

- リテラル<sub>1048576</sub>、<sub>1'048'576</sub>、<sub>0x100000</sub>、<sub>0x10'0000</sub>、および<sub>0'004'000'000</sub>すべてがじをっています。
- リテラル<sub>1.602'176'565e-19</sub>と<sub>1.602176565e-19</sub>はじです。

のはです。はすべてです

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;
```

また、`user-defined`リテラルでもできます。

C++ 14

```
std::chrono::seconds tiempo = 1'674'456s + 5'300h;
```

オンラインでりをむ <https://riptutorial.com/ja/cplusplus/topic/10595/>

## 133: ライブラリアルゴリズム

### Examples

#### std :: for\_each

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

fのすべてのものを [first, last) からして first するとむ last - 1。

パラメーター

first, last - f をする。

f - [first, last) のすべてのもののにされるびしオブジェクト。

り

f C++ 11までおよび std::move(f) C++ 11。

f last - first ににします。

C++ 11

```
std::vector<int> v { 1, 2, 4, 8, 16 };
std::for_each(v.begin(), v.end(), [](int elem) { std::cout << elem << " "; });
```

ベクトルのすべてののためにえられたをし\_vには、このを stdout。

#### std :: 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);
```

[, ]のデータシーケンスをのいにシフトします。 cmpFun をすると、ルールがカスタマイズされます。

パラメーター

first - されるの、

last - されるのわり、

り

そのようながするはtrueをします。  
さもなければ、はにのにスワップされ、falseをす。

On、nはfirstからlastまでのです。

```
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()));
```

に1,2,3のすべてののをします。

```
123
132
213
231
312
321
```

## std :: accumulate

ヘッダ<numeric>されてい<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)
```

std :: accumulateは、のとしてinitまる[first, last) fをしてフォールドをします。

にそれはのものとです

```
T acc = init;
for (auto it = first; first != last; ++it)
 acc = f(acc, *it);
return acc;
```

バージョン1では、fわりにoperator+がされるため、コンテナにすると、コンテナのとです。

パラメーター

first, last - fをする。

init - アキュムレータの。

f - バイナリフォールディング。

り

$f$ アプリケーションの。

$On \times k$ 、ここで、 $n$ は $first$ から $last$ までのであり、 $Ok$ は $f$ のさで $f$ 。

なの

```
std::vector<int> v { 2, 3, 4 };
auto sum = std::accumulate(v.begin(), v.end(), 1);
std::cout << sum << std::endl;
```

10

をにする

C++ 11

```
class Converter {
public:
 int operator()(int a, int d) const { return a * 10 + d; }
};
```

```
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;
```

123

**std::find**

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val);
```

エフェクト

の $val$ ののをします、

パラメーター

$first \Rightarrow$ のをすイテレータ  $last \Rightarrow$ のわりをすイテレータ  $val \Rightarrow$ でする

る

しい==からvalまでのののをすイテレータで、valが見つからないはイテレータがをします。

```
#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;
}
```

```
first occurence of: 9
only occurence of: 43
element after first 9: 10
last element: 48
```

## std :: count

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count (InputIterator first, InputIterator last, const T& val);
```

エフェクト

valにしいのをえます。

パラメーター

first =>のをすイテレータ

last =>のをすイテレータ

val =>のこののがカウントされます

る

でvalとしい==の。

```
#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;
}
```

```
There are 2 9s
There is 1 55
There is 0 101
There are 2 4
```

**std :: count\_if**

```
template <class InputIterator, class UnaryPredicate>
typename iterator_traits<InputIterator>::difference_type
```

```
count_if (InputIterator first, InputIterator last, UnaryPredicate red);
```

エフェクト

されたかであるのをえます。

パラメーター

`first =>のをすイテレータ` `last =>のわりをすイテレータ` `red =>trueまたはfalseをします`

る

がtrueをした、されたのの。

```
#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;
}

```

```

vector size: 15
even numbers: 7 found
odd numbers: 4 found
numbers > 5: 6 found

```

## std :: find\_if

```

template <class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred);

```

エフェクト

`pred`がtrueをすののを `pred` ます。

パラメーター

`first =>`のをすイテレータ `last =>`のわりをすイテレータ `pred =>true`または`false`をします  
る

`pred`のののをすイテレータ—はtrueをします。イテレータは、`val`がつからないにはをす

```

#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;
}
```

```
item not found
First item > 10: 56
First Item n * 10: 10
First Item > 5: 6
```

## std::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);
```

エフェクト

のをします。

## パラメーター

first - がのをしている

last - のわりをすイテレータ `comp` - 2つのをとり、が2よりさいかどうかをすtrueまたはfalseをすポインタまたはオブジェクト。このはをすべきではない

る

のまでのイテレータ

されるのより1つない。

```
#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;
}
```

```
min int from default: 6
min pair from PairLessThanFunction: 2
```

**std::nth\_element**をしてメディアンまたはそののをする

[std::nth\\_element](#)

アルゴリズムは、のイテレータ、 $n$ の、およびの3つのイテレータをします。がすると、 $n$ のでは $n$ にさいになります。このには、よりなオーバーロードがあります例えば、いくつかのをするなど。すべてのバリエーションについてはのリンクをしてください。

メモこのはにです - のさがあります。

こののために、さ $n$ のシーケンスのを「 $n/2$ 」にあるとしてしましょう。例えば、さ5のシーケンスのは3にさいエレメントであり、さ6のシーケンスのもである。

このをしてをめるには、をできます。たちは

```
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].
```

$p$ のをつけるには、ののいくつかをします。

```
const std::size_t pos = p * std::distance(b, e);

std::advance(nth, pos);
```

ポジション<sub>pos</sub>をします。

オンラインでライブラリアルゴリズムをむ <https://riptutorial.com/ja/cplusplus/topic/3177/ライブラリアルゴリズム>

## 134:

き

regexsまたはregexpsともばれるは、されるでするパターンをすテキストです。

**c++ 11**でされたは、マッチしたのリターンや、されたのマッチしたパターンをどのようにきえるかをするのテキストをオプションでサポートします。

- `regex_match` //シーケンスがにマッチしたかどうかをします。オプションでマッチオブジェクトにりみます
- `regex_search` //のがとマッチしたかどうかをします。オプションでマッチオブジェクトにりみます
- `regex_replace` //をしてによってされたをします。
- `regex_token_iterator` //イテレータによってされたシーケンス、するキャプチャインデックスのリスト、およびでされます。Dereferencingは、のインデックスされているをします。インクリメントはのキャプチャインデックスにするか、またはのインデックスがのインデックスにあるは、インデックスをリセットし、シーケンスのマッチののをにします
- `regex_iterator` //イテレータとでされたシーケンスでされます。は、がしているシーケンスのをします。インクリメントすると、シーケンスのマッチののをつける

## パラメーター

|                                                                                                                                                                     |                                                                                                                                                                                                                                                           |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>bool regex_match(BidirectionalIterator first, BidirectionalIterator last, smatch&amp; sm, const regex&amp; re, regex_constraints::match_flag_type flags)</pre> | <p><b>BidirectionalIterator</b>インクリメントするのイテレータである<b>smatch</b>かもしれ<b>cmatch</b>またはの<b>match_results</b>のけれ</p> <p><b>BidirectionalIterator</b> <b>smatch</b>のをしますがな<br/>いか、がされて<b>re</b>のにし<b>first</b>と<b>last</b>された<br/>シーケンス</p>                      |
| <pre>bool regex_match(const string&amp; str, smatch&amp; sm, const regex re&amp;, regex_constraints::match_flag_type flags)</pre>                                   | <p><b>string</b>いずれであってもよい<b>const char*</b>ある<br/>いは<b>Lstring</b>、<b>R-string</b>にされ<b>smatch</b>あつて<br/>もよい<b>cmatch</b>はの<b>match_results</b>のけけ<br/><b>str</b> <b>smatch</b>があればされてもよいしかどうか<br/>をのは、りをとされていない<b>re</b>によってさ<br/>れたのとする<b>str</b></p> |

## Examples

な**regex\_match**と**regex\_search**の

```

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
begining 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"
 }
}

```

## regex\_replaceの

このコードはさまざまなブレーススタイルをりれ、それらを1つのブレーススタイルにします。

```

const auto input = "if (KnR)\n\tfoo();\nif (spaces) {\n foo();\n}\nif
(allman)\n{\n\tfoo();\n}\nif (horstmann)\n{\tfoo();\n}\nif (pico)\n{\tfoo(); }\nif
(whitesmiths)\n\t{\n\tfoo();\n\t}\n";

cout << input << regex_replace(input, regex("(.*?)\\s*\\{?\\s*(.*?;)\\s*\\}\\s*"), "$1
{\n\t$2\n}\n") << endl;

```

## regex\_token\_iteratorの

`std::regex_token_iterator`は、[Comma Separated Valueファイル](#)のをするためのすばらしいツールです。のとはに、このイテレータは、のメソッドがしているエスケープされたカンマをすることもできます。

```

const auto input = "please split,this, csv, ,line,\\,\\,\\n";
const regex re{ "((?:[^\n\\,]|\\n|\\\\.)*)(?:,|$)" };
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"));

```

のイテレータすべきは、`regex`はLでなければならないということです。Rはしません。

## regex\_iteratorの

キャプチャのをりしする `regex_iterator` がある `regex_iterator` は、 `regex_iterator` がいです。  
`regex_iterator` し `match_result`。これは、をつきキャプチャまたはキャプチャにです。C++コード  
をトークンしたいとしましょう。えられた

```
enum TOKENS {
 NUMBER,
 ADDITION,
 SUBTRACTION,
 MULTIPLICATION,
 DIVISION,
 EQUALITY,
 OPEN_PARENTHESIS,
 CLOSE_PARENTHESIS
};
```

このをのように `regex_iterator` `const auto input = "42/2 + -8\t=\n(2 + 2) * 2 * 2 -3"s`

```
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);
}
```

のイテレータですべきは、 `regex` はLでなければならず、Rはしないということです。 [Visual Studio regex\\_iterator Bug](#)

をする

```
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"
```

するとして `const string input` がえられたとしましょう。に、**0**のをつをすることができます `regex_match(input, regex("\\d*"))` または **1**つの `regex_match(input, regex("\\d+"))` は、それらのはい `input` 「123」々は、なくとも**7**のをしていることをするために、**n**のを試してみましょうのようながまれています

```
regex_match(input, regex("\\d{7,}"))
```

これにより、なくとものをすることがされ `input` が、 `input` は「123456789012」のようにすぎるのをめることもできます。そこそこ**n**と**m**ののにくことができます `input`、なくとも**7**のではなく、**11**です。

```
regex_match(input, regex("\\d{7,11}"));
```

これはたちをしますが、`[7]`、`[11]`のにあるなはまだのように、けられている「123456789は、」それでは、なとコードをオプションにしてみましょう

```
regex_match(input, regex("\\d?\\d{7,10}"))
```

それはなは、できるだけないとしていることにすることがだことでした、すでに**10**があるにマッチします。このがあるので `\\d{7,10}`。のをにするためには、`\\d{0,1}`をしなければなりません。は、のすべてののにできます。

ここでは、をオプションにして、があればをくれますか

```
regex_match(input, regex("(?:\\d{3,4})?\\d{7}"))
```

このでは `\\d{7}` に **7**がです。これらの**7**には、**3**または**4**のいずれかをにけることもできます。

なをしなかったことにしてください `\\d{3,4}?\\d{7}`、`\\d{3,4}?3`または**4**のいずれかとし、**3**をんでいました。わりに、グループをで**1**させ、しないことをみます。 `input` 「1234567」のようながまれていないと、がし `input`。

---

のでは、できる、についてしたいといいます。 **lazy**または のいずれかをのにできます。ののは、エンジンにすることです。にこれらのをり、がしてもそれをあきらめないようにします。これはあまりがありません `regex_match(input, regex("\\d{3,4}+\\d{7}"))` "1234567890"のような `input` は `\\d{3,4}+`する**3**がをさせたとしても、`\\d{3,4}+`はに**4**とします。は、されたトークンがなをするときにもよくされます。えば



```
regex_match(input, regex("(?:.*\\d{3,4}){3}"))
```

`input`にのいずれかがまれているかどうかをするためにできます。

```
123 456 7890
123-456-7890
123456-7890
123456 - 7890
```

しかし、このがにいているのは、`input`にながまれているです。

```
12345 - 67890
```

がなければ、`regex`エンジンはにり、するみわせをつけることができるかどうかをべるために、`*`と3または4のすべてのみわせをテストしなければなりません。では2の が'0'のままでまるところからまり、エンジンは`\\d{3,4}`がするように、`*`をしようとしします。がすることができないときは、の `*`がマッチをしたかどうかをするためにバックトラッキングはわれません。

アンカー

C++にはアンカーが4つしかありません

- `^`はのをアサートします
- `$`のわりをす
- `\\b`はのまたはに`\\w`をアサートします
- `\\B`は`\\w`をアサートします

のは、えは、々はそれのをキャプチャしたいとしまししょう

```
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+)" }));
}
```

ここでなのは、アンカーがをしないことです。

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/1681/>

## 135:

### Examples

はです

ほとんどすべてのプログラマーがにいをすのは、このコードがしたとおりにすることをしています。

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

プログラマは0, 0.01, 0.02, 0.03, ..., 1.97, 1.98, 1.99ののすべてののをして<sub>199</sub>をとします。これはにです。

これをにせかける2つのことがこります

1. かれたプログラムはしてわらない。  $a$ はして<sub>2</sub>にしくなく、ループはしてしません。
2. わりに $a < 2$ をチェック  $a < 2$ ループロジックをきえと、ループはしますが、は<sub>199</sub>とはなるになります。 IEEE754にしているマシンでは、くの<sub>201</sub>がされます。

これがするは、かりてられたのをすためです。

なは、のです。

```
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;
```

プログラマがるものは、base10でかれた3つのですが、コンパイラとそのとなるハードウェアにはバイナリのがあります。ので、0.1、0.2、および<sub>0.3</sub>と<sub>10</sub>ベース2の-10システムではにである-whichが、をシステムこれらのはに、なですがどのように<sub>1/3</sub>は、10のな0.333333333333333...にする0.333333333333333...。

```
//64-bit floats have 53 digits of precision, including the whole-number-part.
double a = 0011111110111001100110011001100110011001100110011001100110011010; //imperfect
representation of 0.1
double b = 0011111111001001100110011001100110011001100110011001100110011010; //imperfect
representation of 0.2
double c = 0011111111010011001100110011001100110011001100110011001100110011; //imperfect
representation of 0.3
```

```
double a + b = 0011111110100110011001100110011001100110011001100110011001100110100; //Note that
this is not quite equal to the "canonical" 0.3!
```

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/5115/>

## 136: のオーバーロード

き

C++では、ユーザーにして+や-などのをすることができます。たとえば、`<string>`ヘッダーは、`+`をする+をします。これは、`operator` キーワードをして`operator` をすることによってわれます。

みみのはできません。はユーザーにしてのみオーバーロードできます。すなわち、オペランドのなくとも1つはユーザでなければならない。

のはオーバーロードできません。

- メンバアクセスまたはドットオペレータ`.`
- メンバーアクセスへのポインタ`*`
- スコープ`::`
- `&`、`?:`
- `dynamic_cast`、`static_cast`、`reinterpret_cast`、`const_cast`、`typeid`、`sizeof`、`alignof`、および`noexcept`
- のタイプがになるにされるディレクティブ `#`および`##`。

いくつかのがありますの99.98オーバーロードしないでください

- `&&`と`||` わりに、`bool`へのなをすることをお`bool`
- `,`
- アドレス`&`

どうしてなぜなら、のプログラマーがしないかもしれないをオーバーロードするため、しないとなります。

たとえば、ユーザーが`&&`と`||`したとします。これらののはをい、なシーケンスC++ 17をいます。シーケンシングの、オペレータのにもてはまります。

## Examples

すべてのなをオーバーロードすることができます

- `+`および`+=`
- `-`および`-=`
- `*`と`*=`
- `/`および`/=`
- `&`と`&=`
- `|`そして`|=`
- `^`および`^=`
- `>>`および`>>=`

- <<と<<=

すべてののオーバーロードはじです。スクロールダウン

class / structへのオーバーロード

```
//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;
}
```

class / structのオーバーロード

```
//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;
}
```

`operator+`は、をすることはなく、新しいオブジェクトをします、`const`をすることもなくには`const`によってされるべきではありません、`const`によってされます。のはしです、なぜですかなぜなら

1. のオブジェクトをすることはできません `Object foobar = foo + bar; foo`をしないでください。  
。がありません
2. オブジェクトをすることがあるため、`const`することはできません `operator+`は、オブジェクトを  
する`operator+=`からされているためです

`const&`すとオプションになりますが、されたオブジェクトのなコピーをするがあります。をすることによって、コンパイラはそれをいます。

`operator+=`は、チェーンへののがであるため、へのをしますただし、じをしないでください。シーケンスポイントのためにのになります。

のはしたいですが、`const`はできません。2のはしないでください。そのため、パフォーマンスのから、`const&` `const`しはしよりもいにされます。

2つのをオーバーロードすることができます。

- `++foo`と`foo++`
- `--foo`と`foo--`

オーバーロードはのタイプ `++`と`--` でじです。スクロールダウン

class / structへのオーバーロード

```
//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;
}
```

class / structのオーバーロード

```
//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;
}
```

---

は、へのをすため、をできます。がオブジェクトをするので、のはです。これが`const`でないです  
のすることはできません。

---

postfixはなのをします。なへのであるため、のにガベージがりますの。また、することができな  
ければならないので、`const`にすることもできません。

のは、"びし"オブジェクトにする`const`です。これは、`const`はできないため、でないはのをしな  
いためです。

これは、のオーバーロードになコピーがあるため、 `for` ループで postfix `++` のわりに `++` をするがになるがいです。 `for` ループのからは `for` はにははですが、プレフィックス `++` をする、に "fat" クラスでくのメンバーをコピーしてパフォーマンスをさせるがあります。 `for` ループで `++` をする

```
for (list<string>::const_iterator it = tokens.begin();
 it != tokens.end();
 ++it) { // Don't use it++
 ...
}
```

すべてののをオーバーロードすることができます

- `==` と `!=`
- `>` および `<`
- `>=` および `<=`

これらののをすべてオーバーロードするためのされるは、2つの `==` および `<` をし、りをするためにそれらをすることです。 スクロールダウン

`class / struct` へのオーバーロード

```
//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); }
```

`class / struct` のオーバーロード

```
//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); }
```

はらかにするにして `true` または `false` を `bool` します。

すべてののは `const&` でをとっています。なぜなら、をうのことはだからです。オブジェクトをししないでください。 `&` をすることはよりもであり、がそれをしなないことをするために、それは `const` です。

`class`

/structのはconstとしてされていることにしてください。なぜなら、がconst、コンパイラはオペレータがもしないことをらないので、constオブジェクトをすることはできません。

をオーバーロードすると、をにされたにできます。

は、class / struct するがあります。

```
operator T() const { /* return something */ }
```

オペレータがあるconstできるようにするconstオブジェクトをします。

```
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;
```

[] オーバーロードすることもできます。

あなたはいつも の99.98パーセント2つのバージョン、するがありconstとnot-constオブジェクトがあるので、バージョンをconst、によってされるオブジェクトできないようにするがあり[]。

はしではなくconst&によってされます。これは、しがしよりもであり、constがってインデックスをしないためです。

はによってされます。これは、、オブジェクト[]り、つまり、

```
std::vector<int> v{ 1 };
v[0] = 2; //Changes value of 1 to 2
 //wouldn't be possible if not returned by reference
```

あなたはclass / struct でだけオーバーロードすることができます

```
//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
}
```

の [][]...、プロキシオブジェクトをしてできます。な - クラスののは、これをしています

```
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;
};
```

びし

びしoperator ( ) オーバーロードすることができます

オーバーロードはclass / struct とうがあります

```
//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, ...);
```

えは

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

//Create instance of struct
Sum sum;
int result = sum(1, 1); //result == 2
```

は、のステータスをできるため、ものの1つです。

`class / struct` ために `assignment` をオーバーロードしないと、にされたは "メンバーワイズリで" をします。つまり、すべてのメンバーのをびして1つのオブジェクトをコピーしますのメンバーには、にはメンバー。たとえば、オブジェクトのディープコピーをするがあるなど、なメンバーワイズリでが `class / struct` していないは、をオーバーロードするがあります。

=オーバーロードはですが、いくつかのなになってください。

- りてのテスト。このチェックは、の2つのからです。
  - りてはなコピーなので、するのはにかかっていません。
  - りてのはのステップはしません。
- いデータをしてください。いデータはしいデータにきえなければなりません。は、のステップの2のをすることができます。オブジェクトのコンテンツがされた、りてによってコピーができなくなります。
- すべてのメンバーをコピーします。 `class` または `struct` ために `assignment` をオーバーロードすると、コンパイラーによってにされるわけではないので、のオブジェクトからすべてのメンバーをコピーするがあります。
- これをす `*this`。はをしているのでつまり、 `int b = (a = 6) + 4; //b == 10`、それがによってされます。

```
//T is some type
T& operator=(const T& other)
{
 //Do something (like copying values)
 return *this;
```

```
}
```

`other`は`const&`によってされます。りてられるオブジェクトはしないでください。しはより多く、`operator=`はよりもってしないことをするため`const`です。

`=`ののには `class / struct` そのものなので、は `class / struct` でのみオーバーロードすることができます。なとしてそれをするのはこのをたず、そのためにされています。

`class / struct` であると、のには `class / struct` そのものなので、はありません。

## ビットごとのNOT

ビットのNOT ~ のオーバーロードはかなりです。 スクロールダウン

`class / struct` へのオーバーロード

```
T operator~(T lhs)
{
 //Do operation
 return lhs;
}
```

`class / struct` のオーバーロード

```
T operator~()
{
 T t(*this);
 //Do operation
 return t;
}
```

`operator~`はをするのではなく、しいされたをさなければならぬため、によってりますオブジェクトのとなります。オペレータはする。 `const`ではなく、びしコードがでそれをできるようにするがありますつまり、 `int a = ~a + 1;`でなければなりません。

`class / struct` では、オブジェクトをするがあります。 `this`はできないため、のオブジェクトがされるがあります。

## I/Oのビットシフト

`<<`と`>>`は、に、「きみ」および「みし」としてされます。

- `std::ostream` オーバーロード `<<`をして、をになるストリームにきみます `std::cout`
- `std::istream` overloads `>>`は、のストリームからへみみます `std::cin`

`class / struct` でそれらを ""オーバーロードしたいは、のがじでないをいて、これをうはています

-

りののは、Chaining `std::cout << a << b;` をにするために、しでオーバーロードしたいストリーム `std::ostream` `std::cout << a << b;`。 `std::ostream&`

- `lhs` はりののとじになります
- `rhs` は、オーバーロードをしたいつまり `T`、パフォーマンスのからのわりに `const&` によってされます `rhs` はしないでください。 `const Vector&`。

```
//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;
```

なの

のコードは、なフィールドがにされるにシンプルなをしており、なるフィールドのメンバーをつ4つの+、-、\*、および/ので、の `complex<T>` かスカラーか。

これは、テンプレートのなとしてのオーバーロードをカバーするなをしています。

```
#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;
}

```

き

なC++によって""されたきでC++をすることができます。

に、ダースラインのライブラリからめます。

```

namespace named_operator {
 template<class D>struct make_operator{constexpr make_operator(){};};

 template<class T, char, class Op> 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 Rh>
 auto operator*(half_apply<Lhs, '*', Op>&& lhs, Rh&& rhs)
 -> decltype(named_invoke(std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rh>(rhs)))
 {
 return named_invoke(std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rh>(rhs));
 }
}

```

これはまだもしません。

まず、ベクトルをする



```

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;

```

ここでのコアは、`append_t:named_operator::make_operator<append_t>`の`append`オブジェクトをすることと`append_t:named_operator::make_operator<append_t>`。

に、でなにして`named_invoke`lhs、`append_t`、`rhs`をオーバーロードします。

ライブラリは`lhs*append_t`オーバーロードし、な`half_apply`オブジェクトを`half_apply`ます。また、`half_apply*rhs`を`named_invoke( lhs, append_t, rhs )`をびすためにオーバーロードします。

な`append_t`トークンをし、なのADLにしい`named_invoke`をする`named_invoke`で、すべてがフックアップしてします。

よりなとして、`std::array`のをごとになりたいとします。

```

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])...
 }};
 });
 }
}

```

```
}
```

ライブラリ。

このエレメントワイズコードは、タプル、ペアまたはCスタイルの、またはのコンテナでもするようにでき、さがしないのをします。

また、ので、 `lhs *element_wise<'+'>* rhs` をすることもでき `lhs *element_wise<'+'>* rhs` 。

`*dot*` と `*cross*` プロダクトをくこともです。

`*` のは、 `+` ようなのりをサポートするようにすることができます。 `delimiter` のは、きののをします。これは、 `extra ()` ののでをC++にするときにになるがあります。

のライブラリをしするだけで、のにつて `->*then*` をサポートし、 `std::function` をしたり、 `monadic` `->*bind*` く `->*bind*` 。ステートフルなきをすることもできます。ここでは、 `op` をびしににします。

```
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);
};
```

C++でコンテナをする。

オンラインでのオーバーロードをむ <https://riptutorial.com/ja/cplusplus/topic/562/> のオーバーロード

## 137: の

は、でがからにされます。じをつは、じとじをちます。

1. ::
2. [] () T(...) . -> ++ -- dynamic\_cast static\_cast reinterpret\_cast const\_cast typeid
3. ++ -- \* & + - ! ~ sizeof new delete delete[] 。 Cスタイルのキャスト、 (T)... ; C++ 11  
sizeof... alignof noexcept
4. .\* と -> \*
5. \* 、 / 、 および % 、 バイナリ
6. + と - 、 2
7. << と >>
8. < 、 > 、 <= 、 >=
9. == と !=
10. & 、 ビットのAND
11. ^
12. |
13. &&
14. ||
15. ?:
16. = 、 \* = 、 / = 、 % = 、 + = 、 - = 、 >> = 、 << = 、 & = 、 ^ = 、 | =
17. throw
18. , カンマ

、はです。のバイナリはすべてです。

のルールは、ルールよりもです。

- ・オペランドはあまりにバインドされてい?そのまたは:そのののオペレータにより。、の2  
オペランドはあたかもかこのようにされます。これにより、  $a ? b , c : d$  はです。
- ・オペランドはよりにaにバインドされ?よりもにあるか、にthrowされるので  $a = b ? c : d$  は  $a$   
 $= (b ? c : d)$  し  $throw a ? b : c$  は  $throw (a ? b : c)$  とし  $throw (a ? b : c)$  。
- ・オペランドはよりも、そのにをよりに:そのに、そう  $a ? b : c = d$  は  $a ? b : (c = d)$  。

## Examples

C++のは、とじをちます。

とはからにされることをするをしており、とよりもがく、もっています。

また、( ) をしてのをすることもできます。のでやっているのと同じように。

```
// 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

```

## AND

これらのは、C++ではのをちます。

```

// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || has_foreign_license && num_days <= 60;

```

このコードは、のコードとじです。

```

// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || (has_foreign_license && num_days <= 60);

```

カッコをしてもはわりませんが、みやすくなります。これらのカッコをすることによって、のについてすることはありません。

## &&と||

&&は||よりもされます。これは、にされるものをするためにかっこがされることをします。

C++は&&および||でをします。なをしない。

||のならばをするはありません。

```

#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
}

```

は、ひかれたオブジェクトにし、がくなります。

postfixをすると、がされたにのみアクションがし、いがわれます。

```

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
```

オンラインでのをむ <https://riptutorial.com/ja/cplusplus/topic/3895/>の

## Examples

### デストラクタとデストラクタ

されるようにされたクラスは、Baseクラスとされます。そのようなクラスのなメンバーにするがあります。

にクラスへのポインタをしてされるようにされたクラスは、デストラクタを`virtual`するがあり`virtual`。これにより、たとえオブジェクトがクラスへのポインタによってされたとしても、オブジェクトのをにすることができます。

```
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
```

クラスがポリモーフィックであるはなく、そのインタフェースをさせるがあるは、`protected`デストラクタをします。

```
class NonPolymorphicBase {
public:
 // some methods

protected:
 ~NonPolymorphicBase() = default; // note: non-virtual

private:
 // etc.
};
```

そのようなクラスは、スライスによるかなリークをけるために、ポインタをしてしてすることはできません。

これは、`private` ベースクラスとしてされたクラスににされます。そのようなクラスは、いくつかのものをカプセルし、`virtual` メソッドをカスタマイズポイントとしてするためにされます。このクラスはポリモーフィックにされるべきではなく、`protected` デストラクタはこのをコードにするのにちます。

に、いくつかのクラスは、それらがクラスとしてしてされないことをすることがあります。この、クラスは`final`とマークすることができます。この、のパブリックデストラクタはありません。

```
class FinalClass final { // marked final here
public:
 ~FinalClass() = default;

private:
 // etc.
};
```

## なとコピー

デストラクタをすると、コンパイラがのムーブコンストラクタをしてをすることができなくなることにしてください。デストラクタをするは、になをすることもれないでください。

さらに、をするとコピーのがされるため、これらのオブジェクトもするがありますこのクラスのオブジェクトにコピーセマンティクスがな。

```
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;
};
```

## コピーとスワップ

リソースをするクラスをするは、すべてのなメンバーをするがあります [3/5 /ゼロのルール](#)を。コピーコンストラクタとをくためのもなアプローチはのとおりです。

```
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;
}

```

しかし、このアプローチにはいくつかのがあります。それは、`new[]`は、`delete`はすでにがするリソースをクリアしました`this`してすることはできません。たちは、コピーのりてにおいてくのコピーのロジックをしています。そして、たちは、コピーにオーバーヘッドをするりてチェックをえておくがありますが、としてです。

なをたし、コードのをけるためにのにする、コピー・アンド・スワップ・イディオムをできます。

```

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;
 }
};

```

なぜこれはしますかたちがいるときがこるかをえて

```

person p1 = ...;
person p2 = ...;
p1 = p2;

```

まず、`p2`から`rhs`をコピーしますここではするはありません。そのがスローされた、`operator=`と`p1`はもしません。に、`*this`と`rhs`でメンバーをれえると、`rhs`はになります。とき`operator=`、それはのうちに、のリソースクリーン`this`々がするはありませんでしたデストラクタをして。りてもします。コピーアンドスワップりてとのがですではあまりではありませんが、それがこりそうなシナリオでは、なユースケースをしてしません。

## C++ 11

のは、すでにりてのためににわれています。

```

p1 = std::move(p2);

```

ここでは、`rhs`を`p2`からしてします。りはすべてです。クラスがであるがコピーできない、このはされたコピーコンストラクタのためにになになるため、コピーをするはありません。

## デフォルトコンストラクタ

デフォルトのコンストラクタは、ひされたときにパラメータをとしないコンストラクタのです。それは、それがするのにされ、そのメンバですすべてのコンストラクタがそうです。

```
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
```

「パラメータなし」をたすのは、がすべてのパラメータにデフォルトをすることです。

```
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
```

によってはつまり、がコンストラクタをせず、のものない、コンパイラはにのデフォルトコンストラクタをします。

```
class C{
 std::string s; // note: members need to be default constructible themselves
};

C c1; // will succeed -- C has an implicitly defined default constructor
```

のタイプのコンストラクタをすることは、にべたの1つです。

```
class C{
```

```

 int i;
public:
 C(int i) : i(i){}
};

C c1; // Compile ERROR: C has no (implicitly defined) default constructor

```

## C++ 11

なデフォルトのコンストラクタのをぐため、なは、それを`private` なしとしてすることです。かがコンストラクタをしようとすると、コンパイルエラーがすることになりますこれは、コンパイラによってプライベートエラーまたはリンクエラーになる。

デフォルトのコンストラクタにたがされていることをするために、はのものをにくことができます。

## C++ 11

C++ 11では、は`delete` キーワードをして、コンパイラがデフォルトのコンストラクタをしないように`delete` こともできます。

```

class C{
 int i;
public:
 // default constructor is explicitly deleted
 C() = delete;
};

C c1; // Compile ERROR: C has its default constructor deleted

```

さらに、は、コンパイラにデフォルトのコンストラクタをすることをすることもできます。

```

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

<type\_traits> `std::is_default_constructible` をして、にデフォルトのコンストラクタがあるかどうかまたはプリミティブかどうかをできます。

```

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

C++ 11では、`std::is_default_constructible`のバージョンの`std::is_default_constructible`をすることは`std::is_default_constructible`。

```
cout << boolalpha << is_default_constructible<C1>::value << endl; // prints true
```

## デストラクタ

デストラクタは、ユーザオブジェクトがされるときにびされるのたいます。それは~ですのたいます。

```
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
```

ほとんどのつまり、ユーザがデストラクタをせず、そののたいます、コンパイラはデフォルトのデストラクタをにします。

```
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
}
```

## C++ 11

C++ 11では、は、コンパイラがデフォルトのデストラクタをできないようにすることで、このをにできます。

```
class C{
 int m;
public:
 ~C() = delete; // does NOT have implicit destructor
};

void f{
 C c1;
} // Compile ERROR: C has no destructor
```

さらに、は、コンパイラにデフォルトのデストラクタをすることをすることもできます。

```
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

`<type_traits> std::is_destructible`をして、にデストラクタがあるかどうかまたはプリミティブかどうかをできます。

```
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
```

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/1476/>

## 139:

はになツールですが、いくつかのながあります

- もされたメンバーではないユニオンのにアクセスすることは、C++にってのです。くのC++コンパイラはこのアクセスをなでしていますが、これらはであり、コンパイラですることとはできません。

`std::variant` C++ 17はユニオンとていますが、のをしていますのは、のでされているのです。そのタイプに。

- では、ずしもなるサイズのメンバーをじアドレスにえるはありません。

## Examples

な

は、すべてののがするメモリーをするなです。

```
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;
```

な

は、データをするなど、なデータのメモリーをにえるのにです。

```
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;
}
};

```

の

```

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).
}

```

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/2678/>



## 140: クラス

き

クラスは、でできるキーワードです。のにはしませんが、はエンティティのをします。

auto C ++ 11まで、register C ++ 17まで、static、thread\_local C ++ 11、extern、およびextern 6つのストレージクラスがあります。mutableです。

によると、

くの、あるstorage-class-specifierは、されたdecl-specifier-seqにれます。ただし、thread\_localはstaticまたはexternされstatic。

には、クラスがまれていないがあります。その、はデフォルトをします。たとえば、デフォルトでは、ブロックスコープでされたにはにがあります。

## Examples

な

クラスのなデータメンバーのにできる。オブジェクトがconst あっても、クラスのメンバはconst ではありません。

```
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

mutable 2のはC ++ 11でされました。ラムダのパラメータリストをたどると、ラムダのびしのがconst されます。したがって、なラムダは、コピーによってキャプチャされたエンティティのをすることができます。については、ラムダをしてください。

```
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;
}
```

`mutable`は、なラムダをするためにこのでされるとき、ストレージクラスではないことにしてください。

## C++ 17

がにされることをコンパイラにらせるストレージクラス。「レジスタ」というのは、コンパイラがCPUレジスタにそのようなをして、よりないクロックサイクルでアクセスできるようにするというにしています。C++ 11はとなりました。

```
register int i = 0;
while (i < 100) {
 f(i);
 int g = i*i;
 i += h(i, g);
}
```

とパラメータのを `register` としてすることができます。Cとはなり、C++では `register` をってができるかについてはありません。たとえば、`register` のアドレスをすることはですが、コンパイラがそのようなをにレジスタにするのをけるがあります。

## C++ 17

キーワード `register` はでみです。キーワード `register` をするプログラムはです。

`static` クラスには、3つのなるがあります。

1. のスコープでされたまたはへのリンケージをします。

```
// 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. がなをつようにします `thread_local`。スコープはにです。ローカルは、だけされます。のコントロールはをし、スコープがするたびにされません。

```
void f() {
 static int count = 0;
 std::cout << "f has been called " << ++count << " times so far\n";
}
```

### 3. クラスメンバーのにすると、そのメンバーをメンバーとしてします。

```
struct S {
 static S* create() {
 return new S;
 }
};
int main() {
 S* s = S::create();
}
```

クラスのデータメンバーの、2と3のがにされることにしてください。 `static` キーワードは、メンバーをデータメンバーにしてをつにします。

オート

## C++ 03

のをします。はブロックスコープではすでにデフォルトであり、のスコープではがされていないので、です。

```
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
```

C++ 11では、`auto`はをにし、もはやストレージクラスではなく、わりににされます。

## extern

`extern` ストレージクラスは、コンテキストにじての3つのいずれかでをできます。

1. それをすることなくをするのにうことができます。、これはのファイルでされるのヘッダーファイルでされます。

```
// 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. それは、`const` や `constexpr` がリンクをとっていたとしても、スコープのへのリンケージをえま  
す。

```
// 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. にリンケージでされていたは、ブロックスコープのをします。それのは、リンケージをつし  
いをします。これは、もいむのメンバーです。

```
// 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
}
```

は`extern`としてすることもできますが、これはがありません。これは、ここでされたがのでされ  
ていることをヒントとしてされます。えは

```
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
```

は、のコードでは、`f`にされました`extern`および`g`に`extern`、それがすべてでプログラムのしさを  
にをえないだろうが、のいコードのをさせる。

オンラインでクラスをむ <https://riptutorial.com/ja/cplusplus/topic/9225/クラス>

# 141:

き

CおよびC++でもしいことの1つは、リソースです。ありがたいことに、C++では、たちのプログラムでリソースをするくのがあります。このでは、りてられたリソースをするためにされるイディオムやメソッドのいくつかについてします。

## Examples

リソースがされる

RAIIResource Acquisition Is Initializationは、リソースにおけるのイディオムです。メモリの、[スマートポインタ](#)をしてリソースをいます。RAIIをする、されたリソースは、スマートポインタまたはのリソースマネージャにすぐに加えられます。リソースはこのマネージャをしてのみアクセスされるため、マネージャはさまざまなをできます。たとえば、`std::auto_ptr`は、スコープかられるか、またはされると、するリソースをにします。

```
#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

`std::auto_ptr`のなは、をせずにコピーできないことです。

```
#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
}
```

これらのなコピーセマンティクスのために、`std::auto_ptr`は、とりわけコンテナではできません。このは、メモリを2することをぐためです。じリソースのをつ`auto_ptr`sが2つある、それらはされたときにしようします。すでにされているリソースをするとにがするがあるため、そのリソ

ースをぐことができます。しかし、 `std::shared_ptr`は、これをけ、コピーにをさないがあります

```
#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
```

### ミューテックスとスレッドセーフティ

のスレッドがリソースにアクセスしようとする、がするがあります。なとして、にスレッドをするスレッドがあるとします。これは、にをみみ、を1つしてからします。このを1にし、このスレッドの2つのインスタンスをする、のスレッドがする、こののは3になるはず、。ただし、のにはがっているかがされています。

|       | スレッド1       | スレッド2       |
|-------|-------------|-------------|
| ステップ1 | から1をみむ      |             |
| ステップ2 |             | から1をみむ      |
| ステップ3 | 1プラス1をえて2をる |             |
| ステップ4 |             | 1プラス1をえて2をる |
| ステップ5 | 2をにする       |             |
| ステップ6 |             | 2をにする       |

このように、のわりに2がに3ではなく、っています。そのは、スレッド1がスレッド1をするにをみるためです。ソリューションミューテックス。

ミューテックス **MUT UAL** の clusionのかばんはこののをするためにされたリソースです。スレッドがリソースにアクセスしたい、スレッドはリソースのミューテックスを「」します。リソースへのアクセスがすると、スレッドはミューテックスを「」します。ミューテックスがされている、ミューテックスをするためのすべてのびしは、ミューテックスがされるまでりません。これをよりよくするには、スーパーマーケットでちとしてミューテックスをえてみましょう。ミューテックスをしてするスレッドがするのをってから、リソースをしてスレッドをします。ミューテックスをリリースしてください。もがにリソースにアクセスしようとする、なパンデモニウムが

します。

## C ++ 11

`std::mutex`はC ++ 11のミューテックスのです。

```
#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
}
```

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/8336/>

## 142: の

のはいくつかのなるでこります

- きオーバーロードされたをびします。は、によってつかったすべてのです。
- クラスオブジェクトをびします。は、クラスのすべてのオーバーロードされたびしです。
- オペレータの。は、のスコープでオーバーロードされた、クラスオブジェクトするのオーバーロードされた、およびみみです。
- のためにびすしいまたはコンストラクタをつけるためのオーバーロードの
  - リスト `Class c(value)` の、は`Class`コンストラクタです。
  - リストコピー `Class c = value` と、ユーザーのシーケンスでびすユーザーのをつけるため。は`Class`のコンストラクタであり、ソースがクラスオブジェクトの、そのがします。
  - クラスオブジェクトからクラスをする `Nonclass c = classObject` クラス`Nonclass c = classObject`。は、イニシャライザオブジェクトのです。
  - クラスオブジェクトにをするために `R &r = classObject`、クラスに`r`バインドできるをするがある。は、そのようなです。
  - クラスオブジェクト `Class c{1, 2, 3}` のリストの、はによるのパスのリストコンストラクタです。これでながつからないは、`Class`のコンストラクタをとして、2のパスのオーバーロードがわれます。

## Examples

に

パラメータ・タイプにながないオーバーロード、またはとなされるタイプでなのみが、コールするためにのをとするオーバーロードよりもされます。

```
void f(int x);
void f(double x);
f(42); // calls f(int)
```

がじへののにバインドされるとき、そのはよりくのCVされていてをもをとしなとなされます。

```
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
```

のために、"`T`"は"`T`へのポインタ"とにするとみなされ、`T`はポインタ`T*`とにするとみなされる。  
コンバージョン



```
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
```

## のパラメータコストへの

オーバーロードのは、をにすコストを、「シーケンス」とばれる4つのなるカテゴリのいずれかにします。シーケンスは、ゼロ、1つまたはのコンバージョンをむことができる

- なシーケンス

```
void f(int a); f(42);
```

- ユーザーのシーケンス

```
void f(std::string s); f("hello");
```

- シーケンス

```
void f(...); f(42);
```

- リストのシーケンス

```
void f(std::vector<int> v); f({1, 2, 3});
```

なは、シーケンスがもであり、そのにユーザシーケンスがき、そのにシーケンスがくことである。

なケースは、リストのシーケンスです。これはをしませんリストはをつではありません。そのコストは、パラメータのタイプとイニシャライザリストののじて、の3つのシーケンスの1つとになるようにすることによってされます。

## のとアクセスのチェック

オーバーロードのは、のにします。これは、のがわれた、オーバーロードののによってよりよくするがされないことをします。

```
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
};
```

のはアクセスチェックのにわれます。アクセスできないは、アクセスなよりもいであれば、によってすることができます。

```
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.
```

に、オーバーロードのは、のコールが`explicit`であるかどうかをチェックせずにします。

```
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
```

## のオーバーロード

フオーワーディングのオーバーロードがうまくいくかもしれないので、あなたはにすることがあります

```
struct A {
 A() = default; // #1
 A(A const&) = default; // #2

 template <class T>
 A(T&&); // #3
};
```

ここでののは、`A`がコピーであり、のメンバーをするかもしれないこののコンストラクタがあることです。しかしながら

```
A a; // calls #1
A b(a); // calls #3!
```

コールには2つのなマッチングがあります

```
A(A const&); // #2
A(A&); // #3, with T = A&
```

どちらもですが、`#3`は`#2`よりもcvされていないオブジェクトへのをります。したがって、よりなシーケンスがあり、ななです。

ここののは、にこれらのコンストラクタをすることですえば、`SFINAE`をして。

```
template <class T,
```

```

class = std::enable_if_t<!std::is_convertible<std::decay_t<T>*, A*>::value>
>
A(T&&);

```

ここでは、 $A$  クラスを  $A$  からからとにすることであり、これはのではこのコンストラクタをなにするしたがってからする。その、コピーコンストラクタがびされます。これが、たちがむものです。

## のステップ

のステップはのとおりで。

1. のをしてをつける。なびしは、のルックアップとのルックアップするのをします。
2. のセットをなのセットにフィルタリングします。がびされるとがけるパラメータのになシーケンスがするな。

```

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. もなをびなさい。  $F_1$  のなよりもれである  $F_2$  ののなシーケンス  $F_1$  のするのシーケンスよりくない  $F_2$  、そして...

- 3.1. いくつかのについては、  $F_1$  そのののシーケンスは、  $F_2$  そののシーケンスよりもれています。

```

void f(int); // (1)
void f(char); // (2)

f(4); // call (1), better conversion sequence

```

- 3.2. ユーザーのでは、  $F_1$  のりからタイプへのなシーケンスは、りタイプの  $F_2$  よりもれたシーケンスです。

```

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. では、  $F_1$  は  $F_2$  とじのをちません。

```

struct A
{
 operator X&(); // #1
 operator X&&(); // #2
};
A a;
X& lx = a; // calls #1
X&& rx = a; // calls #2

```

3.4。  $F_1$ はテンプレートではありませんが、  $F_2$ はまたは

```

template <class T> void f(T); // #1
void f(int); // #2

f(42); // calls #2, the non-template

```

3.5。  $F_1$ と $F_2$ はどちらもテンプレートなのですが、  $F_1$ は $F_2$ よりもされています。

```

template <class T> void f(T); // #1
template <class T> void f(T*); // #2

int* p;
f(p); // calls #2, more specialized

```

ここでののはです。よりいシーケンスチェックは、テンプレートとテンプレートチェックのにわれます。これにより、のオーバーロードにするなエラーがします。

```

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

```

にのが1つもないは、びしがです。

```

void f(double) { }
void f(float) { }

f(42); // error: ambiguous

```

のプロモーションとコンバージョン

をするにするが、のにするよりもれています。

```
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
```

floatをdoubleすることをのにするよりもれています。

```
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
```

プロモーションのコンバージョンはおいにれているわけでもなくいありません。

```
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
```

したがって、のまたはのいずれかをつ<sub>f</sub>をびすにあいまいさがないようにするには、8つのオーバーロードがです。したがって、それぞれのなにして、オーバーロードのにするか、されたのユニークなオーバーロードがされます。

```
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);
```

## クラスでのオーバーロード

のでは、このクラスをします。

```
struct A { int m; };
struct B : A {};
struct C : B {};
```

クラスからクラスへのは、ユーザーのよりもされます。これは、しまたはしの、およびポインタのからポインタのへのにされます。

```
struct Unrelated {
 Unrelated(B b);
};
void f(A a);
```

```
void f(Unrelated u);
B b;
f(b); // calls f(A)
```

クラスからクラスへのポインタも、`void*`へのよりもれてい`void*`。

```
void f(A* p);
void f(void* p);
B b;
f(&b); // calls f(A*)
```

のじチェーンにのオーバーロードがあるは、もしたクラスのオーバーロードがされます。これは、ディスパッチとのにづいています。「もした」がされています。ただし、オーバーロードのはにコンパイルにし、にダウンキャストされることはありません。

```
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
```

クラスにしてであるメンバへのポインタについては、のにものがされます。クラスがされます。

```
void f(int B::*p);
void f(int C::*p);
int A::*p = &A::m;
f(p); // calls f(int B::*)
```

## とボラティリティのオーバーロード

であれば、ポインタを`T*`パラメータにすことは、`const T*`パラメータにすよりもれています。

```
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
```

に、`T&`パラメータにをすことは、であれば、たとえがのランクであっても、`const T&`パラメータにすよりもれています。

```
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
```

これはconstされたメンバにもされます。constオブジェクトへのなアクセスとconstオブジェクトへのなアクセスをすることがです。

```
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
```

に、は、よりもましくない。

```
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
```

オンラインでのをむ <https://riptutorial.com/ja/cplusplus/topic/2021/>の

## 143:

き

は、するメモリにされたじのです。は、されたインデックスをつのによってにできます。

これにより、のののをし、にをすることなく、にアクセスすることができます。

## Examples

サイズコンパイルにをにします。

```
#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";
}
```

サイズの`sizeof(a)/sizeof(a[0])`のCイディオムは、ポインタをとってけれ、にったをします。

C++ 11の

C++ 11をすると、のことができます。

```
std::extent<decltype(MyArray)>::value;
```

```
char MyArray[] = { 'X','o','c','e' };
const auto n = std::extent<decltype(MyArray)>::value;
std::cout << n << "\n"; // Prints 4
```



C++ 17までこれからのC++には、のサイズをるためのみみのコアやライブラリユーティリティはありませんでしたが、これはテンプレートへののによってをすことでできます。にしました。きめかいなテンプレートサイズのパラメータは`size_t`で、された`Size`ののとややしています。これは、々テンプレートマッチングのために`size_t`をするg++コンパイラにするためです。

C++ 17では、わりににした`std::size`することができます。

なサイズのの

```
// 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;
}
```

`T a[n];`をするプログラム`T a[n];`、`n`はにされ、C99 VLAをととしてサポートするのコンパイラでコンパイルできます。しかし、VLAはのC++ではサポートされていません。このは、`new[]` - `expression`をしてサイズをでりてををしています。

```
int* a = new int[n]; // ← Allocation of array of n items.
```

...それをつて、に`delete[]` -`expression`をつてりてをします

```
delete[] a;
```

ここでりてられたにはながありますが、`new int[n]()`のようにの`()`するだけでゼロできます。よりには、のタイプにして、これはのをします。

びしののとして、このコードはにではありません。これは、`delete[]`ののおよび`new[]`がメモリリークをきこすためです。そのにする1つののは、えは`std::unique_ptr`スマートポインタをつてクリー

ンアップをすることです。しかし、それにするには、に `std::vector` うことです `std::vector` がそこにあります。

**std :: vector** をってなサイズのをする。

```
// 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` は、サイズのをするライブラリクラステンプレートです。すべてのメモリをし、バッファはしているため、バッファへのポインタ `&v[0]` または `v.data()` をのをとするAPIにすことができます。えは、アイテムをする `push_back` メンバをして、に `vector` をすることさえできます。

ベクトルにまれるコピーまたはをむ  $n_{\text{push\_back}}$  のさは、された  $O\ n$  である。「」して。

には、これは、よりきなバッファがなときに、そのバッファサイズ、そのを2にするベクトルによってされます。えは、サイズ1からまり、 $n = 17_{\text{push\_back}}$  コールのためににじてりしするバッファの、これは、 $1 + 2 + 4 + 8 + 16 = 31$  のコピーをみ、 $2 \times n = 34$  である。よりには、このシーケンスのは  $2 \times n$  をえることはできません。

サイズのののとして、この `vector` ベースのコードでは、ユーザーはののをおよびするはありません。わりに、ベクトルは、ユーザーがしたいごとに、にじてされます。

サイズののマトリックスつまり、2D のの。

```
// 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';
 }
}

```

```

1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21

```

C++では、のインデックスのためのなはサポートされていません。わりに、そのようなはのおそらくなどとみなされ、ののインデックス `[ i ]` がレベルでされます。のでは、`m[y]` は `my` をします。ここで、`y` は0からまるインデックスです。このは、に、えはでけすることができ `m[y][x]` をし、`x` または `-` の `y`。

つまり、のインデックスがもくします。では、このインデックスのは1あたりので、にされた「も」のサイズです。

C++では、ダイナミックアロケーションのダイナミックサイズのビルトインサポートがされていないため、サイズマトリックスはクラスとしてされることがよくあります。に、のけ `m[y][x]` は、をすることつまり、されたのビューがにになるようにすること、またはすことによっていくらかのオーバーヘッドとわずかなさをすること `operator[]` オブジェクトです。したがって、そのようなのためのけは、ルック・アンド・フィールおよびの、えは `m(x,y)` または `m.at(x,y)` または `m.item(x,y)`。

に **std::vector** をするサイズ。

なことにC++ 14のでは、C++ライブラリになサイズのクラスはありません。しかし、サイズをサポートするMatrixクラスは、Boost MatrixライブラリBoostライブラリのサブライブラリをむくのサードパーティライブラリからできます。

Boostやそののライブラリにしたくないは、C++のなサイズの1つがまるでじです

```
vector<vector<int>>> m(3, vector<int>(7));
```

... `vector` は `std::vector` です。ここで、`n` は、ここでは3であるベクトルを `n` コピーすることによっ

てされます。サイズののと同じ`m[y][x]` インデックスをするというがありますが、になりてがまれているため、です。にのサイズをするがあるため、ではありません。

よりでなアプローチは、のとしてのベクトルをし、そのベクトルのするインデックスにクライアントコードの `x`、`y` をマッピングすることです。

```
// 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';
 }
}

```

```

1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21

```

のコードはではありません。なをし、C++をぶのニーズにえるようにされています。

たとえば、`operator()` オーバーロードをして、けのをすることができます。

の

は、のタイプののシーケンシャルなメモリのなるブロックです。はのとじでりてられますが、メモリにまるのをむ`[]`にはがされています。

ののは、`typ int`、`arrayOfInts`、およびにスペースがある`[5]`をしています。

```
int arrayOfInts[5];
```

このようににをしてすることができます

```
int arrayOfInts[5] = {10, 20, 30, 40, 50};
```

すべてのメンバをしてをするとき、のにをれるはありません。コンパイラによってにされます。のでは、5

```
int arrayOfInts[] = {10, 20, 30, 40, 50};
```

また、のだけをし、よりくのをりてることもできます。この、にさをするとはです。では、なでさ5のをりてます。コンパイラは、りのすべてののをこのは0でします。

```
int arrayOfInts[5] = {10,20}; // means 10, 20, 0, 0, 0
```

のデータのもじですることができます。

```
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!"};
```

にアクセスするとき、のインデックスまたはは0からまることにすることもです。

```
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
```

オンラインでをむ <https://riptutorial.com/ja/cplusplus/topic/3017/>

## 144: からいくつかのをす

き

からいくつかのをすとながくあります。えは、をしてとをす、このはにちます。C++でこれをうにはくのがあり、ほとんどがSTLにします。しかし、らかのでSTLをけたいのであれば、`structs/classes`や`arrays`をめ、これをうにはまだいくつかのがあり`arrays`。

### Examples

パラメータの

パラメータは、1つのをすためにできます。それらのパラメータは`const`ポインタまたはであることがされる。

```
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;
}
```

ポインタ

```
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;
}
```

のライブラリやフレームワークでは、の 'OUT' `#define` をして、どのパラメータがシグニチャのパラメータであるかを`#define`にします。これにはのはなく、コンパイルされますが、のシグネチャはしになります。

```
#define OUT

void calculate(int a, int b, OUT int& c) {
 c = a + b;
}
```

**std :: tuple**をう

C++ 11

`std::tuple`は、になるのをむののをのりオブジェクトにバンドルできます。

```
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);
}
```

C++ 17では、きのリストをできます。

## C++ 17

```
std::tuple<int, int, int, int> foo(int a, int b) {
 return {a + b, a - b, a * b, a / b};
}
```

されたtupleからをするのはなので、`std::get`テンプレートをうがあります

```
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);
```

がるにをできれば、tupleをのにするために`std::tie`をうことができます

```
int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 12);
```

りの1つがない、`std::ignore`をできます。

```
std::tie(add, sub, std::ignore, div) = foo(5, 12);
```

## C++ 17

ストラクチャードバイディングをして`std::tie`をできます。

```
auto [add, sub, mul, div] = foo(5,12);
```

あなたのわりにのタプルののタプルをすようにしたいは、`std::tie`するわりに`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);
}
```

する

```
void increase_least(int& a, int& b) {
 std::get<0>(minmax(a,b))++;
}
```



まれに、`std::tie`代わりに`std::tie std::forward_as_tuple`します。なものがされるのになさではないかもしれないので、あなたがそうするならばしてください。

## std :: arrayをう

### C ++ 11

コンテナ`std::array`は、のりをにねることができます。これはコンパイルにるがあり、すべてのりはじでなければなりません

```
std::array<int, 4> bar(int a, int b) {
 return { a + b, a - b, a * b, a / b };
}
```

これは、`int bar[4]`というのcスタイルのをきえます。は、さまざまなc++ `std`がではそれでできることです。また、バインドされたチェックきのなメンバアクセスと、なしでのサイズをすことができる`size`など`at`なメンバも`size`います。

## std :: pairをする

テンプレート`std::pair`は、2つののに 2つのりをねることができます。

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
 return std::make_pair(a+b, a-b);
}
```

C ++ 11では、`std::make_pair`わりにイニシャライザリストをできます。

### C ++ 11

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
 return {a+b, a-b};
}
```

ペアの`first`および`second`メンバオブジェクトをして、された`std::pair`の々のをりすことができます。

```
std::pair<int, int> mrvs = foo(5, 12);
std::cout << mrvs.first + mrvs.second << std::endl;
```

10

をう

`struct`をしてのりを`struct`することができます。

### 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

々のフィールドにするわりに、コンストラクタをしてりのをにすることができます。

```

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);

```

foo() がす々ののは、 struct calc メンバにアクセスすることでできます。

```

std::cout << calc.add << ' ' << calc.sub << ' ' << calc.mul << ' ' << calc.div << '\n';

```

## 17-7 60 0

struct をする、りはのオブジェクトにグループされ、のあるをしてアクセスです。これは、されたのにされたなのをらすのにもちます。

## C++ 17

からされた struct をアンパックするために、 [された](#) をすることができます。これは、 out-parameters を in-parameters をしてにします。

```

int a=5, b=12;
auto[add, sub, mul, div] = foo(a, b);
std::cout << add << ' ' << sub << ' ' << mul << ' ' << div << '\n';

```

このコードのはとじです。 struct はききののをすためにされます。これにより、フィールドをにすることができます。

## バインディング

### C++ 17

C++ 17ではストラクチャードバインディングがされています。これにより、`std::tie()`にすることがないため、またはタプルのアンパックをうがいないため、のりのをうがです。

```
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';
}
```

バインディングは、`std::pair`、`std::tuple`、およびデータメンバーがパブリックメンバーまたはナクラスのメンバーであるすべてのので、デフォルトでできます。

```
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;
```

あなたのタイプを "タプルのようなもの"にすると、あなたのタイプもにします。タプルのようなは、な`tuple_size`、`tuple_element`をつであり`get`かれる

```
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");
 }
}

```

これはします

```

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';
}

```

オブジェクトコンシューマの

のするでびされるをすることができます。

## 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';
});

```

これは「スタイル」としてられています。

タプルをすを、のようにスタイルをするにさせることができます。

## 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';
};

```

よりなバージョンはC++ 14またはC++ 11でできます。

## std::vectorをう

std::vectorは、じのをすのにです。のでは、intをデータとしてしていますが、std::vectorはにコピーなのをできます。

```

#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;
}

```

## イテレータの

イテレータをにすことで、じのいくつかのをすことができます。これは、なライブラリのアルゴリズムなどでにです。

```

template<typename Incrementable, typename OutputIterator>
void generate_sequence(Incrementable from, Incrementable to, OutputIterator output) {
 for (Incrementable k = from; k != to; ++k)
 *output++ = k;
}

```

```
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}
```

オンラインでからいくつかのをすをむ <https://riptutorial.com/ja/cplusplus/topic/487>/からいくつかのをす

## 145: テンプレートのオーバーロード

- じ、じタイプであっても、のはしてテンプレートにしません。
- のびしとされたテンプレートびしは、じ、じり、じリストをしていてもなっています

### Examples

なテンプレートのオーバーロードはですか

テンプレートは、テンプレートののオーバーロードじだなるパラメータタイプのルールでオーバーロードすることができ、それにえて、オーバーロードはif

- りのなるか、または
- テンプレートのパラメータリストは、パラメータのとデフォルトのをいてなりますのではありません

のでは、2つのパラメータをするのはコンパイラにとってです。なぜなら、すべてのinformatをつっているからです。しかし、テンプレートのはまだされていないかもしれません。したがって、2つのパラメータがしいのルールはであり、デブエンドエンドとがするがあり、とのスペルがじであるがあるよりには、いわゆるODRルール、ただしテンプレートパラメータのをすることができます。ただし、このようななるスペルのでは、の2つのがなるとみなされますが、にじにインスタンスされます。オーバーロードはですが、コンパイラからのはありません。

```
template<typename T>
void f(T*) { }

template<typename T>
void f(T) { }
```

"T"と "T \*"はなるりなので、これはなオーバーロードです。しかし、はであり、はありません

```
template<typename T>
void f(T (*x)[sizeof(T) + sizeof(T)]) { }

template<typename T>
void f(T (*x)[2 * sizeof(T)]) { }
```

オンラインでテンプレートのオーバーロードをむ <https://riptutorial.com/ja/cplusplus/topic/4164/テンプレートのオーバーロード>

## 146: のオーバーロード

き

にするのトピックもしてください。

あるタイプをのタイプににすることができ、そののタイプにしてするがない、さがするがあります。

えば

```
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

のオーバーロードとはですか

のオーバーロードでは、じスコープでされたのがじにする スコープとばれます 、 シグネチャだけがなります。

`std::string`でまるされたのをいっているとします。

```
void print(const std::string &str)
{
 std::cout << "This is a string: " << str << std::endl;
}
```

これはにしますが、 `int` をけり、それもするがであるとします。あなたはくことができます

```
void print_int(int num)
{
 std::cout << "This is an int: " << num << std::endl;
}
```

しかし、2つのがなるパラメータをけれるので、にくことができます

```
void print(int num)
{
 std::cout << "This is an int: " << num << std::endl;
}
```

は`print`というの2つのがありますが、なるがあります。1つは`std::string`けり、もう1つは`int`けり



ます。すぐあなたはのについてすることなくそれらをびすことができます

```
print("Hello world!"); //prints "This is a string: Hello world!"
print(1337); //prints "This is an int: 1337"
```

のわりに

```
print("Hello world!");
print_int(1337);
```

オーバーロードされたがあると、コンパイラはそれをするパラメータからどのをびすかをします。のオーバーロードをするときはがです。たとえば、なの

```
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;
}
```

すぐあなたがくときに`print`オーバーロードがびされるのはすぐにはわかりません

```
print(5);
```

コンパイラにいくつかのがかりをえるがあるかもしれません

```
print(static_cast<double>(5));
print(static_cast<int>(5));
print(5.0);
```

オプションのパラメータをけるオーバーロードをするときには、いくつかがです。

```
// 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;
}
```

`print(17)` ようなびしがオプションの2のために1または2ののためにされているかどうかをコンパイラがするはないので、コンパイルにします。

のオーバーロードでのりの

りのについてをオーバーロードすることはできません。えば

```
// WRONG CODE
std::string getValue()
{
 return "hello";
}

int getValue()
{
 return 0;
}

int x = getValue();
```

りのが`int`りてられていても、コールする`getValue`バージョンをコンパイラができないため、コンパイルエラーがします。

## メンバー**cv-qualifier**オーバーロード

クラスのは、そのクラスへの`cv`をしてアクセスされたときにオーバーロードされます。これはもには`const`オーバーロードにされますが、`volatile`と`const volatile`オーバーロードにもできます。これは、すべてのスタティックメンバが`cv`がされるのパラメータとして`this`をとるためです。これは`const`オーバーロードにもにされますが、`volatile`および`const volatile`でもできます。

メンバーは、びされたインスタスとなくとも`cv`されているにのみびすことができるため、これはです。ながら`const`インスタスがびすことができる`const`と`const`メンバーを、`const`インスタスにのみびすことができます`const`メンバー。これにより、はびしのインスタスの`cv`によってなるをすることができ、プログラマは、そのをつバージョンをしないことによって、ましくない`cv`のをすることができます。

いくつかのな`print`メソッドをつクラスは、のように`const`オーバーロードすることができます

```
#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"
}

```

これは、`const` さのなです。メンバを `const` としてマークすることによって、`const` インスタンスにしてびすことができます。するがない、は `const` ポインタ/としてインスタンスをります。これにより、されていないパラメータを `const` として、`cv` なしでされたパラメータをり、よりでみやすいコードにすることで、コードがをするかどうかをすることができます。

```

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.
}

```

これのなは、アクセッサを `const` として、`mutator` を `const` としてすることです。

`const` メンバでクラスメンバをすることはできません。 `std::mutex` をロックするなど、にがなメンバがあるは、`mutable` としてできます。

```

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;
};
```

オンラインでのオーバーロードをむ <https://riptutorial.com/ja/cplusplus/topic/510/>のオーバーロード

## 147: メンバー

- `//`びし
  - `variable.member_function;`
  - `variable_pointer-> member_function;`
- `//`
  - `ret_type class_name :: member_functioncv-qualifiers {`
    - `;`
  - `}`
- `//`プロトタイプ
  - `class class_name {`
    - `virt-specifier ret_type`メンバー`cv-qualifiers virt-specifier-seq;`
    - `// virt-specifier`する、「`」`。
    - `// cv-qualifiers "const"および/または "volatile"`する
    - `// virt-specifier-seq`する、「`き`」および/または「`」`。
  - `}`

`static`メンバは、`class / struct / union`メンバであり、のインスタンスにしてびされ、インスタンスにしてする。`static`メンバーとはなり、インスタンスをしなければびすことはできません。

クラス、およびについては、[トピック](#)をしてください。

## Examples

### メンバー

`class`または`struct`は、メンバとメンバをつことができます。これらは、スタンドアロンとほとんどじをち、クラスのまたはでできます。クラスのでされているは、ののにクラスのとスコープ`::`がきます。

```
class CL {
public:
 void definedInside() {}
 void definedOutside();
};
void CL::definedOutside() {}
```

これらは、ドット `.`をつクラスのインスタンスまたはインスタンスへの、または `->`をつインスタンスへのポインタでびされ、びしはのインスタンスにびけられますびされました。メンバがインスタンスでびされると、[thisポインタ](#)をしてそのインスタンスのすべてのフィールドにアクセスできますが、それらのインスタンスがパラメータとしてされているにのみのインスタンスのフ

フィールドにアクセスできます。

```
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.
```

これらは、やのアクセスになく、メンバやそののメンバにアクセスすることができます。コンパイラがクラスのコンパイルをするに、クラスをするがあるため、メンバーやそのにされたメンバーにアクセスすることで、ですることもできます。

```
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)); }
};
```

## カプセル

メンバのなは、フィールドにアクセスするのではなく、アクセッサにゲッタと呼ばれるとミューテータにセッタと呼ばれるをしてカプセルすることです。

```
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);
 }
};
```

クラスでは、メンバーによって `encapsulated` されたものにアクセスできます。クラスで、それへのアクセスをして、メンバによってされる `get_encapsulated()` それをむし、 `set_encapsulated()` それをします。これにより、のしないがされます。これは、したがみきにされるためです。[ゲッターとセッターがカプセルをしたりちっているかどうかについてくのがあり、どちらのにもいがある。このようないはこののです。]

## のとインポート

クラスがオーバーロードされたのセットをし、クラスがそのセットにのオーバーロードをすると、クラスによってされるすべてのオーバーロードがになります。

```
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.
```

これはのルールによるものですのにしいがつかると、そののエンティティのしいバージョンがつからないでも `hd.f(s)` ;このため、クラスのをオーバーロードすると、によってクラスのオーバーロードがされなくなります。これをけるために、`using`をして、クラスからクラスにを「インポート」することができます。これにより、のにできるようになります。

```
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

```

クラスが、`using`でをインポートするだけでなく、クラスのとじシグネチャでをすると、クラスはにオーバーライドまたはになります。

```

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).
};

```

インポートされたエンティティがクラスで `public` または `protected` れていれば、`using` をしてアクセスをすることもできます。

```

struct ProMem {
 protected:
 void func() {}
};

struct BecomesPub : ProMem {
 using ProMem::func;
};

// ...

ProMem pm;
BecomesPub bp;

pm.func(); // Error: protected.
bp.func(); // Good.

```

に、ののクラスからメンバーをにびしたい、をびすときにをし、そのクラスをですることができます。

```

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.

```

## メンバー

メンバーを`virtual`としてすることもでき`virtual`。この、ポインタまたはインスタンスへのびされた、それらはアクセスされません。むしろ、テーブル `vtable` または `vftable` としてよくられているのためのポインタのメンバのリストでを `vftable`、それをしてインスタンスのにしたバージョンをびすのタイプです。がクラスのからびされた、ルックアップはされません。

```

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.

```

なおながら `pd` ある `Base*`、`rd` ある `Base&` びす `func()` のつのびしのいずれかに `Derived::func()` のわりに、`Base::func()` これは、`Derived` の `vtable` が `Base::func()` エントリを `Derived::func()` すようにするためです。に、インスタンスを `slicer()` すと、されたインスタンスが `Derived` であっても、`Base::func()` がびされることになります。これは、するデータスライスとしてられているものであ

るDerivedにインスタンスをBaseによりパラメータがDerivedないインスタンスBaseアクセスできないインスタンス。

メンバがとしてされている、オーバーライドがvirtualとしてされているかどうかにかかわらず、シグネチャをつすべてのクラスメンバがします。これは、プログラマがするためにしたクラスをよりしくすることができます。ただし、どのがvirtualあるかについてのがないためです。

```
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.
};
```

ただし、は、そのがするにのみをオーバーライドすることにしてください。がにvirtualされたとしても、がであれば、わりにしいをします。

```
struct BadB {
 virtual void f() {}
};

struct BadD : BadB {
 virtual void f(int i) {} // Does NOT override BadB::f.
};
```

## C++ 11

C++ 11、きするは、のキーワードのoverrideできoverride。これは、プログラマがクラスをオーバーライドすることをしていることをコンパイラにえます。コンパイラはもオーバーライドしないとエラーをします。

```
struct CPP11B {
 virtual void f() {}
};

struct CPP11D : CPP11B {
 void f() override {}
 void f(int i) override {} // Error: Doesn't actually override anything.
};
```

これはまた、がであり、なくとも1つのクラスでもされていることをプログラマーにえるメリットがあります。なクラスをしやすくすることができます。

がvirtualされ、クラスのでされるとき、virtualはにまれなければならない、でりされません。

## C++ 11

これはoverrideもてはまりoverride。

```
struct VB {
 virtual void f(); // "virtual" goes here.
 void g();
};
/* virtual */ void VB::f() {} // Not here.
virtual void VB::g() {} // Error.
```

クラスが `virtual` をオーバーロードする、`virtual` としてにされたオーバーロードのみが `virtual` になります。

```
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).
```

については、[するトピック](#)をしてください。

## Const Correctness

this `cv` のなの1つは、[const](#) **しさ**です。これは、オブジェクトをするのあるアクセスのみがオブジェクトをできること、およびオブジェクトをするのないメンバまたはメンバには、そのオブジェクトへのきみアクセスがないことをするですオブジェクトにもにも。これによりしないがされ、コードのエラーがなくなります。また、をするのないは、をきえたり、オーバーロードすることなく、`const` または `const` オブジェクトのいずれかをすることができます。

`const` しさは、ボトムアップからまります。をするのないクラスメンバは、`const` インスタンスとして `const` ように `const` としてされています。これはに、されたしパラメータをするがないときに `const` することをにします。これにより、は `const` や `non-const` オブジェクトのいずれかををわずにすることができます、`const` -nessはこの。このため、ゲッターは、をするのないのとに、に `const` です。

```
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.

```

ConstIncorrectとConstCorrectコメントでされているように、なcvもドキュメントとしてします。クラスがあるはconstしい、ではありませんのconstをするとすることができ、あるのconstなをしないとすることが出来ます。

オンラインでメンバーをむ <https://riptutorial.com/ja/cplusplus/topic/5661/メンバー>

## クレジット

| S. No |                     | Contributors                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1     | C ++をいめる            | <a href="#">Adhokshaj Mishra</a> , <a href="#">ankit dassor</a> , <a href="#">aquirdturtle</a> , <a href="#">ArchbishopOfBanterbury</a> , <a href="#">Bakhtiar Hasan</a> , <a href="#">Bart van Nierop</a> , <a href="#">Ben H</a> , <a href="#">Bo Persson</a> , <a href="#">Brandon</a> , <a href="#">Brian</a> , <a href="#">BullshitPingu</a> , <a href="#">cb4</a> , <a href="#">celtschk</a> , <a href="#">Cheers and hth. - Alf</a> , <a href="#">chrisb2244</a> , <a href="#">Cody Gray</a> , <a href="#">Community</a> , <a href="#">cpatricio</a> , <a href="#">Curious</a> , <a href="#">Daemon</a> , <a href="#">Daksh Gupta</a> , <a href="#">Danh</a> , <a href="#">darkpsychic</a> , <a href="#">David Bippes</a> , <a href="#">David G.</a> , <a href="#">DeepCoder</a> , <a href="#">Dim_ov</a> , <a href="#">dlemstra</a> , <a href="#">Donald Duck</a> , <a href="#">Dr t</a> , <a href="#">Dylan Little</a> , <a href="#">Edward</a> , <a href="#">emlai</a> , <a href="#">Erick Q.</a> , <a href="#">ethanwu10</a> , <a href="#">Fantastic Mr Fox</a> , <a href="#">Florian</a> , <a href="#">GIRISH kuniyal</a> , <a href="#">greatwolf</a> , <a href="#">honk</a> , <a href="#">Humam Helfawi</a> , <a href="#">Hurkyl</a> , <a href="#">Ilyas Mimouni</a> , <a href="#">Isak Combrinck</a> , <a href="#">itzmukeshy7</a> , <a href="#">Jason Watkins</a> , <a href="#">JedaiCoder</a> , <a href="#">Jerry Coffin</a> , <a href="#">Jim Clark</a> , <a href="#">Johan Lundberg</a> , <a href="#">Jon Harper</a> , <a href="#">jotik</a> , <a href="#">Justin</a> , <a href="#">Justin Time</a> , <a href="#">JVApn</a> , <a href="#">K48</a> , <a href="#">Ken Y-N</a> , <a href="#">Keshav Sharma</a> , <a href="#">kiner_shah</a> , <a href="#">krOoze</a> , <a href="#">Leandros</a> , <a href="#">maccard</a> , <a href="#">Malcolm</a> , <a href="#">Malick</a> , <a href="#">Manan Sharma</a> , <a href="#">manetsus</a> , <a href="#">manlio</a> , <a href="#">Marco A.</a> , <a href="#">Mark Gardner</a> , <a href="#">MasterHD</a> , <a href="#">Matt</a> , <a href="#">Matt Lord</a> , <a href="#">mnoronha</a> , <a href="#">Muhammad Aladdin</a> , <a href="#">Mustaghees</a> , <a href="#">muXXmit2X</a> , <a href="#">mynameisausten</a> , <a href="#">Nathan Osman</a> , <a href="#">Neil A.</a> , <a href="#">Nemanja Boric</a> , <a href="#">neuro</a> , <a href="#">Nicol Bolas</a> , <a href="#">nouλdλzeλO</a> , <a href="#">Optimus Prime</a> , <a href="#">Pavel Strakhov</a> , <a href="#">Peter</a> , <a href="#">Praetorian</a> , <a href="#">Qchmqz</a> , <a href="#">Quirk</a> , <a href="#">RamenChef</a> , <a href="#">Rushikesh Deshpande</a> , <a href="#">SajithP</a> , <a href="#">Sam Cristall</a> , <a href="#">Serikov</a> , <a href="#">Shoe</a> , <a href="#">SirGuy</a> , <a href="#">Soapy</a> , <a href="#">Soul_man</a> , <a href="#">theo2003</a> , <a href="#">λolλez əλλ qoq</a> , <a href="#">Tom K</a> , <a href="#">TriskalJM</a> , <a href="#">Trizzle</a> , <a href="#">UncleZeiv</a> , <a href="#">VermillionAzure</a> , <a href="#">Walter</a> , <a href="#">Wen Qin</a> , <a href="#">Wexiwa</a> , <a href="#">πάντα ρεί</a> , <a href="#">パスカル</a> |
| 2     | 1つのルールODR           | <a href="#">Brian</a> , <a href="#">Jarod42</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| 3     | 3、5、0のルール           | <a href="#">Adrien Descamps</a> , <a href="#">Barry</a> , <a href="#">ChrisN</a> , <a href="#">hello</a> , <a href="#">honk</a> , <a href="#">Johan Lundberg</a> , <a href="#">Justin Time</a> , <a href="#">JVApn</a> , <a href="#">Loki Astari</a> , <a href="#">mpromonet</a> , <a href="#">Nicol Bolas</a> , <a href="#">Nirmal4G</a> , <a href="#">NonNumeric</a> , <a href="#">Null</a> , <a href="#">Peter</a> , <a href="#">relgukxilef</a> , <a href="#">Scott Weldon</a> , <a href="#">T.C.</a> , <a href="#">TriskalJM</a> , <a href="#">Venemo</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 4     | Arithmiticメタプログラミング | <a href="#">Meena Alfons</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| 5     | C ++ 11メモリモデル       | <a href="#">NonNumeric</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 6     | C ++コンテナ            | <a href="#">John DiFini</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| 7     | C ++ストリーム           | <a href="#">Ami Tavory</a> , <a href="#">didiz</a> , <a href="#">JVApn</a> , <a href="#">mpromonet</a> , <a href="#">Sergey</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

|    |                                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|----|--------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 8  | C ++でのデータ                                  | <a href="#">Gaurav Sehgal</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 9  | C ++でのデザインパターンの                            | <a href="#">Antonio Barreto</a> , <a href="#">datosh</a> , <a href="#">didiz</a> , <a href="#">Jarod42</a> , <a href="#">JVApen</a> , <a href="#">Nikola Vasilev</a>                                                                                                                                                                                                                                                                                                                         |
| 10 | C ++でのユニットテスト                              | <a href="#">elvis.dukaj</a> , <a href="#">VermillionAzure</a>                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 11 | C ++での                                     | <a href="#">celtschk</a> , <a href="#">R_Kapp</a>                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 12 | C ++での                                     | <a href="#">John Bargman</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 13 | C ++での                                     | <a href="#">Daemon</a> , <a href="#">Nicol Bolas</a> , <a href="#">Владимир Стрелец</a>                                                                                                                                                                                                                                                                                                                                                                                                      |
| 14 | C ++での                                     | <a href="#">4444</a> , <a href="#">JVApen</a> , <a href="#">lorro</a> , <a href="#">mindriot</a>                                                                                                                                                                                                                                                                                                                                                                                             |
| 15 | C ++でののビヘイビア                               | <a href="#">didiz</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| 16 | C ++デバッグとデバッグツールとテクニック                     | <a href="#">Adam Trhon</a> , <a href="#">JVApen</a> , <a href="#">King's jester</a> , <a href="#">Misgevolution</a>                                                                                                                                                                                                                                                                                                                                                                          |
| 17 | C ++ "value by call" と "reference by call" | <a href="#">Error - Syntactical Remorse</a> , <a href="#">Henkersmann</a>                                                                                                                                                                                                                                                                                                                                                                                                                    |
| 18 | Const Correctness                          | <a href="#">amanuel2</a> , <a href="#">Justin Time</a>                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 19 | constexpr                                  | <a href="#">Ajay</a> , <a href="#">Brian</a> , <a href="#">diegodfrf</a> , <a href="#">mtb</a> , <a href="#">Null</a>                                                                                                                                                                                                                                                                                                                                                                        |
| 20 | constキーワード                                 | <a href="#">Barry</a> , <a href="#">Jarod42</a> , <a href="#">Jatin</a> , <a href="#">Justin</a> , <a href="#">Podgorskiy</a> , <a href="#">tenpercent</a> , <a href="#">ThyReaper</a>                                                                                                                                                                                                                                                                                                       |
| 21 | Cの                                         | <a href="#">パスカル</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 22 | Elisionをコピー                                | <a href="#">Nicol Bolas</a> , <a href="#">TartanLlama</a>                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| 23 | ISO C ++                                   | <a href="#">Bakhtiar Hasan</a> , <a href="#">Barry</a> , <a href="#">C.W.Holeman II</a> , <a href="#">ComicSansMS</a> , <a href="#">didiz</a> , <a href="#">diegodfrf</a> , <a href="#">Guillaume Pascal</a> , <a href="#">Ivan Kush</a> , <a href="#">Johan Lundberg</a> , <a href="#">Justin Time</a> , <a href="#">JVApen</a> , <a href="#">manlio</a> , <a href="#">Marco A.</a> , <a href="#">MSalters</a> , <a href="#">Nicol Bolas</a> , <a href="#">sth</a> , <a href="#">vishal</a> |
| 24 | OpenMPでの                                   | <a href="#">Andrea Chua</a> , <a href="#">JVApen</a> , <a href="#">Nicol Bolas</a> , <a href="#">Sumurai8</a>                                                                                                                                                                                                                                                                                                                                                                                |
| 25 | Pimpl Idiom                                | <a href="#">Danh</a> , <a href="#">Daniele Pallastrelli</a> , <a href="#">emlai</a> , <a href="#">Jordan Chapman</a> , <a href="#">JVApen</a> , <a href="#">manlio</a> , <a href="#">Stephen Cross</a> , <a href="#">Yakk</a>                                                                                                                                                                                                                                                                |
| 26 | RAIIリソースがされる                               | <a href="#">Barry</a> , <a href="#">defube</a> , <a href="#">Jarod42</a> , <a href="#">JVApen</a> , <a href="#">Loki Astari</a> , <a href="#">Niall</a> , <a href="#">Nicol Bolas</a> , <a href="#">RamenChef</a> , <a href="#">Sumurai8</a> , <a href="#">Tannin</a>                                                                                                                                                                                                                        |

|    |                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|----|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 27 | RTTI                       | Brian, deepmax, Pankaj Kumar Boora, Roland, Savas Mikail KAPLAN                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 28 | SFINAEはりではない               | Barry, Fox, Jarod42, Jason R, Jonathan Lee, Luc Danton, sp2danny, SU3, w1th0utnam3, Xosdy, Yakk                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 29 | static_assert              | Jarod42, JVApn, lorro, Marco A., Richard Dally, T.C.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| 30 | std :: any                 | demonplus, Marco A.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| 31 | std :: array               | CinCout, Daksh Gupta, Dinesh Khandelwal, Error - Syntactical Remorse, Malcolm, Nikola Vasilev, plasmacel                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| 32 | std :: atomics             | Nikola Vasilev                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| 33 | std :: forward_list        | Nikola Vasilev                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| 34 | std :: function呼びなをラップする   | elimad, Evgeniy, Nicol Bolas, Tarod                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| 35 | std :: integer_sequence    | Dietmar Kühl                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| 36 | std :: iomanip             | kiner_shah, Nikola Vasilev, Yakk                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| 37 | std :: map                 | 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                                                                                                                                                                                                                                                                                                                    |
| 38 | std :: optional            | Barry, diegodfrf, Jahid, Jared Payne, JVApn, Null, Yakk                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 39 | std :: pair                | Ajay, Bim, demonplus, kiner_shah, Nikola Vasilev, Ravi Chandra                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| 40 | std :: setとstd :: multiset | G-Man, JVApn, Mikitori                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 41 | std :: string              | 1337ninja, 3442, Andrea Corbelli, Barry, Bim, caps, Christopher Oezbek, cpplearner, crea7or, Curious, drov, Edward, Emil Rowland, emlai, Fantastic Mr Fox, fbrereto, ggrr, Holt, honk, immerhart, Jack, Jahid, Jerry Coffin, Jonathan Mee, jotik, JPNNotADragon, jpo38, Justin, JVApn, 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 |

|    |                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|----|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 42 | std :: unordered_mapの | <a href="#">tulak.hord</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| 43 | std :: variant        | <a href="#">Yakk</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 44 | std :: vector         | <a href="#">2power10</a> , <a href="#">A. Sarid</a> , <a href="#">Aaron Stein</a> , <a href="#">alain</a> , <a href="#">Alex Logan</a> , <a href="#">Ami Tavory</a> , <a href="#">anatolyg</a> , <a href="#">anderas</a> , <a href="#">Andy</a> , <a href="#">AndyG</a> , <a href="#">arunmoezhi</a> , <a href="#">Bakhtiar Hasan</a> , <a href="#">Barry</a> , <a href="#">Benjamin Lindley</a> , <a href="#">bluefog</a> , <a href="#">bone</a> , <a href="#">CHess</a> , <a href="#">CinCout</a> , <a href="#">Cody Gray</a> , <a href="#">Colin Basnett</a> , <a href="#">ComicSansMS</a> , <a href="#">Community</a> , <a href="#">cute_ptr</a> , <a href="#">Daksh Gupta</a> , <a href="#">Daniel</a> , <a href="#">Daniel Stradowski</a> , <a href="#">Dario</a> , <a href="#">David G.</a> , <a href="#">David Yaw</a> , <a href="#">DeepCoder</a> , <a href="#">diegodfrf</a> , <a href="#">dkg</a> , <a href="#">Dr t</a> , <a href="#">Duly Kinsky</a> , <a href="#">Ed Cottrell</a> , <a href="#">Edward</a> , <a href="#">ehudt</a> , <a href="#">emlai</a> , <a href="#">enrico.bacis</a> , <a href="#">Falias</a> , <a href="#">Fantastic Mr Fox</a> , <a href="#">Fox</a> , <a href="#">foxcub</a> , <a href="#">gaazkam</a> , <a href="#">Galik</a> , <a href="#">gartenriese</a> , <a href="#">granmirupa</a> , <a href="#">Holt</a> , <a href="#">honk</a> , <a href="#">Hurkyl</a> , <a href="#">iliketocode</a> , <a href="#">immerhart</a> , <a href="#">Isak Combrinck</a> , <a href="#">Jarod42</a> , <a href="#">Jason Watkins</a> , <a href="#">JHBonarius</a> , <a href="#">Johan Lundberg</a> , <a href="#">John Slegers</a> , <a href="#">jotik</a> , <a href="#">jpo38</a> , <a href="#">JVApn</a> , <a href="#">Kevin Katzke</a> , <a href="#">krOoze</a> , <a href="#">Loki Astari</a> , <a href="#">lordjohncena</a> , <a href="#">manetsus</a> , <a href="#">manlio</a> , <a href="#">Marco A.</a> , <a href="#">Matt</a> , <a href="#">Michael Gaskill</a> , <a href="#">Misha Brukman</a> , <a href="#">MotKohn</a> , <a href="#">Motti</a> , <a href="#">mtk</a> , <a href="#">NageN</a> , <a href="#">Niall</a> , <a href="#">Nicol Bolas</a> , <a href="#">Null</a> , <a href="#">patmanpato</a> , <a href="#">Paul Beckingham</a> , <a href="#">paul-g</a> , <a href="#">Ped7g</a> , <a href="#">Praetorian</a> , <a href="#">Pyves</a> , <a href="#">R. Martinho Fernandes</a> , <a href="#">Rakete1111</a> , <a href="#">Randy Taylor</a> , <a href="#">Richard Dally</a> , <a href="#">Roddy</a> , <a href="#">Romain Vincent</a> , <a href="#">Rushikesh Deshpande</a> , <a href="#">Ryan Hilbert</a> , <a href="#">Saint-Martin</a> , <a href="#">Samar Yadav</a> , <a href="#">Samer Tufail</a> , <a href="#">Sayakiss</a> , <a href="#">Serikov</a> , <a href="#">Shoe</a> , <a href="#">silvergasp</a> , <a href="#">Skipper</a> , <a href="#">solidcell</a> , <a href="#">Stephen</a> , <a href="#">sth</a> , <a href="#">strangeqargo</a> , <a href="#">T.C.</a> , <a href="#">Tamarous</a> , <a href="#">theo2003</a> , <a href="#">Tom</a> , <a href="#">towl</a> , <a href="#">Trevor Hickey</a> , <a href="#">TriskalJM</a> , <a href="#">user1336087</a> , <a href="#">user2176127</a> , <a href="#">Vladimir Gamalyan</a> , <a href="#">Wolf</a> , <a href="#">Yakk</a> |
| 45 | typedefとのエイリアス        | <a href="#">Brian</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 46 | アラインメント               | <a href="#">Brian</a> , <a href="#">Marco A.</a> , <a href="#">Nicol Bolas</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| 47 | イテレータ                 | <a href="#">Barry</a> , <a href="#">chrisb2244</a> , <a href="#">cute_ptr</a> , <a href="#">Daniel Jour</a> , <a href="#">Edgar Rokyan</a> , <a href="#">EvgeniyZh</a> , <a href="#">fbrereto</a> , <a href="#">Gal Dreiman</a> , <a href="#">Gaurav Kumar Garg</a> , <a href="#">GIRISH kuniyal</a> , <a href="#">honk</a> , <a href="#">Hurkyl</a> , <a href="#">JPNotADragon</a> , <a href="#">JVApn</a> , <a href="#">Mike H-R</a> , <a href="#">Null</a> , <a href="#">Oz.</a> , <a href="#">Sergey</a> , <a href="#">Serikov</a> , <a href="#">tilz0R</a> , <a href="#">Yakk</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| 48 | インライン                 | <a href="#">Brian</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 49 | インライン                 | <a href="#">amanuel2</a> , <a href="#">Aravind .KEN</a> , <a href="#">Bim</a> , <a href="#">Brian</a> , <a href="#">legends2k</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| 50 | オート                   | <a href="#">Artalus</a> , <a href="#">Barry</a> , <a href="#">celtschk</a> , <a href="#">Daniele Pallastrelli</a> , <a href="#">Edward</a> , <a href="#">Igor Oks</a> , <a href="#">Jarod42</a> , <a href="#">Johan Lundberg</a> , <a href="#">manlio</a> , <a href="#">Yakk</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 51 | オブジェクトタイプのレイアウト       | <a href="#">Brian</a> , <a href="#">Justin Time</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 52 | キーワード                 | <a href="#">ADITYA</a> , <a href="#">amanuel2</a> , <a href="#">Brian</a> , <a href="#">Danh</a> , <a href="#">John London</a> , <a href="#">Justin Time</a> ,                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |



|    |                                         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|----|-----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    |                                         | JVApen, Kerrek SB, Loki Astari, manlio, Marco A., Nicol Bolas, OliPro007, Rakete1111, RamenChef, Roland, start2learn                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 53 | クライアントサーバーの                             | Abhinav Gauniyal                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| 54 | クラス/                                    | Alexey Voytenko, anderas, aquirdturtle, Brian, callyalater, chrisb2244, Colin Basnett, Dan Hulme, darkpsychic, Dragma, 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                                                                                                                                                                                               |
| 55 | このポインタ                                  | amanuel2, Justin Time, RamenChef                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| 56 | コンパイルとビルド                               | 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, JVApen, 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                    |
| 57 | サイドバイサイドC++とC++11とC++14とC++17でされたなC++のの | wasthishepful                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| 58 | シングルトンデザインパターン                          | deepmax, Galik, Jarod42, Johan Lundberg, JVApen, Stradigos                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 59 | スコープ                                    | deepmax, Error - Syntactical Remorse                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 60 | ストリームマニピュレータ                            | Nicol Bolas, Владимир Стрелец                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| 61 | スマートポインタ                                | 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 |

|    |            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|----|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 62 | スレッディング    | <a href="#">Alejandro</a> , <a href="#">amchacon</a> , <a href="#">Brian</a> , <a href="#">CaffeineToCode</a> , <a href="#">ComicSansMS</a> , <a href="#">Dair</a> , <a href="#">defube</a> , <a href="#">didiz</a> , <a href="#">Diligent Key Presser</a> , <a href="#">Galik</a> , <a href="#">James Adkison</a> , <a href="#">james large</a> , <a href="#">Jason Watkins</a> , <a href="#">Jeremi Podlasek</a> , <a href="#">mpromonet</a> , <a href="#">Niall</a> , <a href="#">nwp</a> , <a href="#">Rakete1111</a> , <a href="#">Stephen Cross</a> , <a href="#">Sumurai8</a> , <a href="#">Yakk</a> , <a href="#">ysdx</a> , <a href="#">Yuushi</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| 63 | スレッド       | <a href="#">didiz</a> , <a href="#">Galik</a> , <a href="#">JVApn</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| 64 | セマフォ       | <a href="#">didiz</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| 65 | セマンティクスをする | <a href="#">Barry</a> , <a href="#">Cheers and hth. - Alf</a> , <a href="#">ChemiCalChems</a> , <a href="#">David Doria</a> , <a href="#">didiz</a> , <a href="#">Guillaume Racicot</a> , <a href="#">Justin Time</a> , <a href="#">JVApn</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 66 | タイプキーワード   | <a href="#">Brian</a> , <a href="#">Justin Time</a> , <a href="#">Omnifarious</a> , <a href="#">RamenChef</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 67 | タイプ        | <a href="#">Barry</a> , <a href="#">Brian</a> , <a href="#">Emmanuel Mathi-Amorim</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| 68 | タイプ        | <a href="#">Brian</a> , <a href="#">celtschk</a> , <a href="#">greatwolf</a> , <a href="#">Jarod42</a> , <a href="#">Yakk</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 69 | タイプ        | <a href="#">Jarod42</a> , <a href="#">silvergasp</a> , <a href="#">TartanLlama</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| 70 | テンプレート     | <a href="#">Barry</a> , <a href="#">Benjy Kessler</a> , <a href="#">Brian</a> , <a href="#">callyalater</a> , <a href="#">cb4</a> , <a href="#">celtschk</a> , <a href="#">CodeMouse92</a> , <a href="#">Colin Basnett</a> , <a href="#">DeepCoder</a> , <a href="#">Diligent Key Presser</a> , <a href="#">Eldritch Cheese</a> , <a href="#">eXPerience</a> , <a href="#">FedeWar</a> , <a href="#">Gabriel</a> , <a href="#">Greg</a> , <a href="#">Holt</a> , <a href="#">honk</a> , <a href="#">J_T</a> , <a href="#">Johannes Schaub - litb</a> , <a href="#">Justin</a> , <a href="#">JVApn</a> , <a href="#">Loki Astari</a> , <a href="#">M. Viaz</a> , <a href="#">manlio</a> , <a href="#">Maxito</a> , <a href="#">MSalters</a> , <a href="#">Nicol Bolas</a> , <a href="#">Pontus Gagge</a> , <a href="#">Praetorian</a> , <a href="#">Rakete1111</a> , <a href="#">Ricardo Amores</a> , <a href="#">Ryan Haining</a> , <a href="#">Sergey</a> , <a href="#">SirGuy</a> , <a href="#">Smeeheey</a> , <a href="#">Sumurai8</a> , <a href="#">user1887915</a> , <a href="#">W.F.</a> , <a href="#">WMios</a> , <a href="#">Wolf</a> , <a href="#">πάντα ῥεῖ</a> |
| 71 | パラメータパック   | <a href="#">Marco A.</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| 72 | ビットフィールド   | <a href="#">Ajay</a> , <a href="#">Perette Barella</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| 73 | ビット        | <a href="#">A. Sarid</a> , <a href="#">Barry</a> , <a href="#">Cody Gray</a> , <a href="#">CroCo</a> , <a href="#">FedeWar</a> , <a href="#">Jarod42</a> , <a href="#">JVApn</a> , <a href="#">manlio</a> , <a href="#">tambre</a> , <a href="#">Tarod</a> , <a href="#">Trevor Hickey</a> , <a href="#">Алексей Неудачин</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 74 | ビット        | <a href="#">Loki Astari</a> , <a href="#">Mads Marquart</a> , <a href="#">manlio</a> , <a href="#">txtechhelp</a> , <a href="#">Алексей Неудачин</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| 75 | ビルドシステム    | <a href="#">Ami Tavory</a> , <a href="#">celtschk</a> , <a href="#">Florian</a> , <a href="#">Jahid</a> , <a href="#">Jason Watkins</a> , <a href="#">Justin</a> , <a href="#">JVApn</a> , <a href="#">Nathan Osman</a> , <a href="#">RamenChef</a> , <a href="#">VermillionAzure</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| 76 | ファイルI/O    | <a href="#">anderas</a> , <a href="#">ankit dassor</a> , <a href="#">Anonymous1847</a> , <a href="#">AProgrammer</a> , <a href="#">Bakhtiar Hasan</a> , <a href="#">bitek</a> , <a href="#">Chachmu</a> , <a href="#">ComicSansMS</a> , <a href="#">didiz</a> , <a href="#">Dietmar Kühl</a> , <a href="#">Dr t</a> , <a href="#">Emanuel Vintilă</a> , <a href="#">Galik</a> , <a href="#">honk</a> , <a href="#">Hurkyl</a> , <a href="#">Jérémie Bolduc</a> , <a href="#">John Strood</a> , <a href="#">JVApn</a> , <a href="#">Loki Astari</a> , <a href="#">manlio</a> , <a href="#">Mathieu K.</a> , <a href="#">MikeMB</a> , <a href="#">mindriot</a> , <a href="#">Nicol Bolas</a> , <a href="#">nwp</a> , <a href="#">patmanpato</a> , <a href="#">Rakete1111</a> ,                                                                                                                                                                                                                                                                                                                                                                                              |

|                                                             |            |                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------------------------------------------------------|------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RomCoo, Serikov, sheng09, shrike, svgspnr, Алексей Неудачин |            |                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 77                                                          | プリプロセッサ    | 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, phandinhlán, Richard Dally, Sean, signal, silvergasp, Sumurai8, T.C., Tanjim Hossain, tenpercent, The Philomath, Владимир Стрелец, パスカル                            |
| 78                                                          | フレンドキーワード  | Perette Barella, Sergey                                                                                                                                                                                                                                                                                                                                                                                                           |
| 79                                                          | フロー        | anotherGatsby, Brian, JVApen, mkluwe, Qchmq, RamenChef, Tejendra                                                                                                                                                                                                                                                                                                                                                                  |
| 80                                                          | プロファイリング   | Ami Tavory, paul-g                                                                                                                                                                                                                                                                                                                                                                                                                |
| 81                                                          | ヘッダーファイル   | RamenChef, VermillionAzure                                                                                                                                                                                                                                                                                                                                                                                                        |
| 82                                                          | ポインタ       | Baron, daB0bby, FedeWar, Hindrik Stegenga, Nicol Bolas, Nitinkumar Ambekar, Pietro Saccardi, Reverie Wisp, West                                                                                                                                                                                                                                                                                                                   |
| 83                                                          | ミューテックス    | didiz, hyoslee, JVApen                                                                                                                                                                                                                                                                                                                                                                                                            |
| 84                                                          | メタプログラミング  | 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 ρῆι                                                                                                                                                                                                                          |
| 85                                                          | メモリ        | Andrei, Brian, callyalater, Daksh Gupta, Galik, JVApen, madduci, nnrales, RamenChef, ThyReaper                                                                                                                                                                                                                                                                                                                                    |
| 86                                                          | メンバーへのポインタ | John Burger, start2learn                                                                                                                                                                                                                                                                                                                                                                                                          |
| 87                                                          | ユーザーのリテラル  | Brian, Cid1025, Jarod42, Roland, sigalor, sth                                                                                                                                                                                                                                                                                                                                                                                     |
| 88                                                          | ラムダ        | Adi Lester, Aganju, Ajay, alain, anderas, Andrea Corbelli, Barry, bcmpinc, Brian, Christopher Oezbek, Community, cpplerner, derekerdmann, Edd, Falias, Firas Moalla, honk, Jean-Baptiste Yunès, Johan Lundberg, Johannes Schaub - litb, John Slegers, JVApen, Loki Astari, Loufylouf, M. Viaz, Mike Dvorkin, Nicol Bolas, Patryk, Praetorian, Rakete1111, RamenChef, Ryan Haining, Sergio, Serikov, Snowhawk, teivaz, Yakk, ygram |
| 89                                                          | リテラル       | Brian, Nikola Vasilev, RamenChef                                                                                                                                                                                                                                                                                                                                                                                                  |
| 90                                                          | リファクタリング   | asantacreu, Cody Gray, Edward, Jarod42, JVApen,                                                                                                                                                                                                                                                                                                                                                                                   |

| RamenChef |                  |                                                                                                                                                                                                                                                                                                                       |
|-----------|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 91        | リンケージ            | Brian                                                                                                                                                                                                                                                                                                                 |
| 92        | ループ              | 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 |
| 93        | なコンパイル/リンカエラーGCC | Asu, immerhart                                                                                                                                                                                                                                                                                                        |
| 94        | なテンプレートパターンCRTP  | Barry, Brian, Gabriel, honk, Nicol Bolas, Ryan Haining                                                                                                                                                                                                                                                                |
| 95        | の                | AndreiM, Brian, Jarod42, Yakk                                                                                                                                                                                                                                                                                         |
| 96        | べえ               | anatolyg, Barry, Daniel, Ivan Kush, maccard, manetsus, manlio, MikeMB, MKAROL, Nicol Bolas, Patrick, Ravi Chandra, SajithP, timrau, Trevor Hickey                                                                                                                                                                     |
| 97        |                  | Ha., manlio, merlinND, Sumurai8                                                                                                                                                                                                                                                                                       |
| 98        | メンバー             | 0x5f3759df, Daksh Gupta, Johan Lundberg, Justin Time, Motti, Sergey, T.C.                                                                                                                                                                                                                                             |
| 99        | しているヘッダ          | Edward, marcinj, Naor Hadar, RamenChef                                                                                                                                                                                                                                                                                |
| 100       |                  | Alexey Guseynov, Brian, callyalater, Dr t, Jahid, Jarod42, Johan Lundberg, jotik, Martin Ba, Nemanja Boric, Null, Peter, Rakete1111, Ronen Ness                                                                                                                                                                       |
| 101       | とのセマンティクス        | JVApen, Nicol Bolas                                                                                                                                                                                                                                                                                                   |
| 102       | のカテゴリ            | Barry, ChemiCalChems, Curious, fefe, Johannes Schaub - litb, mnoronha, Nicol Bolas, Praetorian, SirGuy                                                                                                                                                                                                                |
| 103       | と                | didiz, Nicol Bolas                                                                                                                                                                                                                                                                                                    |
| 104       | ミューテックス          | didiz                                                                                                                                                                                                                                                                                                                 |
| 105       |                  | Denkkar, Fantastic Mr Fox, Jarod42, Nicol Bolas, SajithP, stackptr, T.C.                                                                                                                                                                                                                                              |
| 106       | りてとコピー           | amanuel2, Roland                                                                                                                                                                                                                                                                                                      |
| 107       | タイプ              | JVApen, Stephen                                                                                                                                                                                                                                                                                                       |

|     |           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 108 |           | <a href="#">Andrea Corbelli</a> , <a href="#">Asu</a> , <a href="#">Daksh Gupta</a> , <a href="#">darkpsychic</a> , <a href="#">rockoder</a>                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 109 |           | <a href="#">Brian</a> , <a href="#">Daniel Käfer</a> , <a href="#">Emmanuel Mathi-Amorim</a> , <a href="#">marquesm91</a> , <a href="#">RamenChef</a>                                                                                                                                                                                                                                                                                                                                                                                                        |
| 110 | キーワード     | <a href="#">Barry</a> , <a href="#">Community</a> , <a href="#">Dean Seo</a> , <a href="#">start2learn</a> , <a href="#">T.C.</a> , <a href="#">tenpercent</a>                                                                                                                                                                                                                                                                                                                                                                                               |
| 111 | のない       | <a href="#">jotik</a> , <a href="#">Roland</a> , <a href="#">ThyReaper</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 112 |           | <a href="#">anderas</a> , <a href="#">Andrea Chua</a> , <a href="#">Barry</a> , <a href="#">Brian</a> , <a href="#">DeepCoder</a> , <a href="#">emlai</a> , <a href="#">Isak Combrinck</a> , <a href="#">Jarod42</a> , <a href="#">Jérémy Roy</a> , <a href="#">Johannes Schaub - litb</a> , <a href="#">Julien-L</a> , <a href="#">JVApn</a> , <a href="#">Nicol Bolas</a> , <a href="#">Null</a> , <a href="#">Rakete1111</a> , <a href="#">randag</a> , <a href="#">Roland</a> , <a href="#">T.C.</a> , <a href="#">tenpercent</a> , <a href="#">Yakk</a> |
| 113 | びしオブジェクト  | <a href="#">JVApn</a> , <a href="#">turon</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 114 |           | <a href="#">Andrea Chua</a> , <a href="#">Jim Clark</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| 115 | タイプのキーワード | <a href="#">amanuel2</a> , <a href="#">Brian</a> , <a href="#">Kerrek SB</a> , <a href="#">RamenChef</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| 116 | キーワード     | <a href="#">Brian</a> , <a href="#">RamenChef</a> , <a href="#">start2learn</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| 117 |           | <a href="#">A. Sarid</a> , <a href="#">Christophe</a> , <a href="#">Jarod42</a> , <a href="#">Jeremi Podlasek</a> , <a href="#">Justin Time</a> , <a href="#">manlio</a>                                                                                                                                                                                                                                                                                                                                                                                     |
| 118 | な         | <a href="#">In silico</a> , <a href="#">Johannes Schaub - litb</a> , <a href="#">Nicol Bolas</a> , <a href="#">Roland</a>                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| 119 | メンバ       | <a href="#">Vijayabhaskarreddy CH</a> , <a href="#">Yakk</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 120 | のるい       | <a href="#">2501</a> , <a href="#">Bo Persson</a> , <a href="#">Brian</a> , <a href="#">Dutow</a> , <a href="#">Jahid</a> , <a href="#">Jarod42</a> , <a href="#">jotik</a> , <a href="#">Justin Time</a> , <a href="#">Iz96</a> , <a href="#">manlio</a> , <a href="#">Nicol Bolas</a> , <a href="#">Peter</a>                                                                                                                                                                                                                                              |
| 121 | の         | <a href="#">Brian</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| 122 |           | <a href="#">Ajay</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 123 |           | <a href="#">ibrahim5253</a> , <a href="#">JVApn</a> , <a href="#">Kerrek SB</a> , <a href="#">MathSquared</a> , <a href="#">SingerOfTheFall</a>                                                                                                                                                                                                                                                                                                                                                                                                              |
| 124 | テンプレート    | <a href="#">Ajay</a> , <a href="#">BigONotation</a> , <a href="#">celtschk</a> , <a href="#">defube</a> , <a href="#">Jarod42</a> , <a href="#">Jonathan Lee</a> , <a href="#">Roland</a> , <a href="#">T.C.</a> , <a href="#">Yakk</a>                                                                                                                                                                                                                                                                                                                      |
| 125 | のりみ       | <a href="#">AndyG</a> , <a href="#">Barry</a> , <a href="#">cpplerner</a> , <a href="#">Firas Moalla</a> , <a href="#">Marco A.</a> , <a href="#">Rakete1111</a> , <a href="#">T.C.</a> , <a href="#">Yakk</a>                                                                                                                                                                                                                                                                                                                                               |
| 126 |           | <a href="#">Fanael</a> , <a href="#">Johannes Schaub - litb</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| 127 | り         | <a href="#">Cheers and hth. - Alf</a> , <a href="#">sorosh_sabz</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| 128 | な         | <a href="#">4444</a> , <a href="#">Brian</a> , <a href="#">JVApn</a> , <a href="#">Nikola Vasilev</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

|     |             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 129 |             | <a href="#">chema989</a> , <a href="#">ralismark</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 130 | の           | <a href="#">Ami Tavory</a> , <a href="#">AndreiM</a> , <a href="#">Ben Steffan</a> , <a href="#">Brian</a> , <a href="#">Cody Gray</a> , <a href="#">cshu</a> , <a href="#">Dovahkiin</a> , <a href="#">Elias Kosunen</a> , <a href="#">emlai</a> , <a href="#">Emma X</a> , <a href="#">FedeWar</a> , <a href="#">fefe</a> , <a href="#">ggrr</a> , <a href="#">GIRISH kuniyal</a> , <a href="#">Hiura</a> , <a href="#">Jeremi Podlasek</a> , <a href="#">Johannes Schaub - litb</a> , <a href="#">JVApn</a> , <a href="#">kd1508</a> , <a href="#">Ken Y-N</a> , <a href="#">manetsus</a> , <a href="#">manlio</a> , <a href="#">Marco A.</a> , <a href="#">Mat</a> , <a href="#">mceo</a> , <a href="#">Motti</a> , <a href="#">Naor Hadar</a> , <a href="#">nbro</a> , <a href="#">Nicol Bolas</a> , <a href="#">Peter</a> , <a href="#">Rakete1111</a> , <a href="#">ralismark</a> , <a href="#">RamenChef</a> , <a href="#">Sebastian Ärleryd</a> , <a href="#">Tannin</a> , <a href="#">Trevor Hickey</a> , <a href="#">Tyler Durden</a>                                                                                                                                                             |
| 131 | のりの         | <a href="#">Brian</a> , <a href="#">define cindy const</a> , <a href="#">Torbjörn</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| 132 | り           | <a href="#">diegodfrf</a> , <a href="#">JVApn</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 133 | ライブラリアルゴリズム | <a href="#">Ami Tavory</a> , <a href="#">Barry</a> , <a href="#">Daniel</a> , <a href="#">Duly Kinsky</a> , <a href="#">Edgar Rokyan</a> , <a href="#">Guillaume Pascal</a> , <a href="#">Jarod42</a> , <a href="#">NinjaDeveloper</a> , <a href="#">Patrik Obara</a> , <a href="#">Peter</a> , <a href="#">Riom</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 134 |             | <a href="#">honk</a> , <a href="#">Jonathan Mee</a> , <a href="#">Justin</a> , <a href="#">JVApn</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 135 |             | <a href="#">Xirema</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 136 | のオーバーロード    | <a href="#">ArchbishopOfBanterbury</a> , <a href="#">Archie Gertsman</a> , <a href="#">Ates Goral</a> , <a href="#">Barry</a> , <a href="#">Brian</a> , <a href="#">Candlemancer</a> , <a href="#">chrisb2244</a> , <a href="#">defube</a> , <a href="#">enzom83</a> , <a href="#">James Adkison</a> , <a href="#">Rakete1111</a> , <a href="#">Sergey</a> , <a href="#">start2learn</a> , <a href="#">Xeverous</a> , <a href="#">Yakk</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 137 | の           | <a href="#">an0o0nym</a> , <a href="#">Brian</a> , <a href="#">didiz</a> , <a href="#">JVApn</a> , <a href="#">start2learn</a> , <a href="#">turon</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 138 |             | <a href="#">Barry</a> , <a href="#">krOoze</a> , <a href="#">OliPro007</a> , <a href="#">Reuben Thomas</a> , <a href="#">TriskalJM</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| 139 |             | <a href="#">manlio</a> , <a href="#">ThyReaper</a> , <a href="#">txtechhelp</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| 140 | クラス         | <a href="#">Brian</a> , <a href="#">start2learn</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| 141 |             | <a href="#">Anonymous1847</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 142 | の           | <a href="#">Barry</a> , <a href="#">Brian</a> , <a href="#">didiz</a> , <a href="#">Johannes Schaub - litb</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| 143 |             | <a href="#">Cheers and hth. - Alf</a> , <a href="#">Isak Combrinck</a> , <a href="#">manlio</a> , <a href="#">Matthew Brien</a> , <a href="#">Wen Qin</a> , <a href="#">Wolf</a> , <a href="#">ΦXocę Πεπεúπα ツ</a>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 144 | からいくつかのをす   | <a href="#">aaron斯诺well</a> , <a href="#">Bakhtiar Hasan</a> , <a href="#">Barry</a> , <a href="#">bitek</a> , <a href="#">celtschk</a> , <a href="#">Christopher Oezbek</a> , <a href="#">Community</a> , <a href="#">DeepCoder</a> , <a href="#">Dr t</a> , <a href="#">Ela782</a> , <a href="#">Fantastic Mr Fox</a> , <a href="#">Galik</a> , <a href="#">honk</a> , <a href="#">J_T</a> , <a href="#">Jarod42</a> , <a href="#">Johan Lundberg</a> , <a href="#">Johannes Schaub - litb</a> , <a href="#">John Slegers</a> , <a href="#">Jon Chesterfield</a> , <a href="#">Kevin Katzke</a> , <a href="#">Let_Me_Be</a> , <a href="#">Loki Astari</a> , <a href="#">M. Sadeq H. E.</a> , <a href="#">manetsus</a> , <a href="#">Menasheh</a> , <a href="#">Michael Gaskill</a> , <a href="#">mnoronha</a> , <a href="#">Niall</a> , <a href="#">Nicol Bolas</a> , <a href="#">Null</a> , <a href="#">Peter</a> , <a href="#">Rakete1111</a> , <a href="#">Richard Forrest</a> , <a href="#">Ryan Hilbert</a> , <a href="#">Stephen</a> , <a href="#">T.C.</a> , <a href="#">templatetypedef</a> , <a href="#">tenpercent</a> , <a href="#">user3384414</a> , <a href="#">Yakk</a> , <a href="#">Ze</a> |

| Rubeus, パスカル |                |                                                                                                                                                                |
|--------------|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 145          | テンプレートのオーバーロード | Johannes Schaub - litb, Kunal Tyagi, RamenChef                                                                                                                 |
| 146          | のオーバーロード       | Bakhtiar Hasan, Barry, Cody Gray, CoffeeandCode, didiz, Galik, Jatin, Johannes Schaub - litb, Justin Time, JVApen, Rakete1111, Sean, Sumurai8, Tim Straubinger |
| 147          | メンバー           | Justin Time, RamenChef                                                                                                                                         |