Бесплатная электронная книга

УЧУСЬ

C Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#c

1
1: C 2
2
2
C 3
( ): 3
API, C (, ,): 4
4
Examples 4
, 4
5
5
6
6
GCC 6
clang 7
Microsoft C 7
7
«, !» K & R C 7
2: - 10
Examples 10
, 10
10
11
3: Interprocess Communication (IPC) 15
15
Examples 15
15
1.1: 16
1.2: 17
4: Typedef 20

20
20
20
Examples 21
Typedef 21
Typedef 22
22
22
23
Typedef 23
5: Valgrind 26
26
26
Examples 26
Valgrind 26
26
Bytes lost - 26
Valgrind 27
6: X- 29
29
29
Examples 29
X- printfs 30
Enum 30
: X 30
31
X- , 4 , 31
, 32
7: 33
33
33
33

Examples 34
34
35
GNU getopt 35
8: 39
39
39
Examples 40
40
9: 41
41
41
41
41
Examples 41
41
43
43
? 44
- 45
10: 47
Examples 47
, 47
main.c: 47
source1.c: 47
source2.c: 47
headerfile.h: 48
11: 50
50
Pros 50
Cons 50
Examples 50

gcc Basic asm 51
gcc Extended asm 51
gcc 52
12: 54
Examples 54
if () 54
if () ... else 55
switch () 55
if () ... else if () ... else 58
if () ... else VS if () .. else 58
13: 60
..... 60
Examples 60
..... 60
..... 61
..... 62
Xorshift 62
14: 64
..... 64
Examples 64
..... 64
15: 66
Examples 66
C 66
..... 68
..... 68
..... 69
..... 69
..... 69
..... 70
..... 71
16: / : , , - 72
.....

72

72

/ 72

/ 72

Examples 72

72

73

Do-While 74

for 74

75

Loop Duff 76

17: **78**

78

78

78

80

80

80

81

81

Examples 81

81

82

82

84

85

_Thread_local 86

18: **87**

87

87

Examples 87

/ * * / 87

// 88
..... 89
- 89
19: 91
..... 91
..... 91
Examples 93
..... 93
..... 93
..... 94
..... 94
..... 94
..... 95
..... 96
..... 98
..... 100
20: , ... 101
..... 101
Examples 101
..... 101
..... 102
..... 102
..... 103
21: 105
..... 105
Examples 105
stdbool.h 105
#define 105
() _Bool 106
..... 107
bool typedef 108
22: 109

109

109

109

Examples 110

110

() 112

112

114

115

115

116

117

120

121

122

23: 123

123

123

123

Examples 123

C11 Threads 123

24: 125

125

Examples 125

125

126

25: 128

128

128

Examples 130

130

130

132
132
134
135
136
136
137
, 138
138
, 140
140
141
printf 141
142
143
143
printf% s 143
144
fflush 145
- 145
, getenv, strerror setlocale 146
, `_Noreturn` ` noreturn` 147
26: 149
149
149
Examples 149
149
150
27: 152
Examples 152
152
Prototype Scope 152
153

154

28: 156

156

156

Examples 156

ERRNO 156

strerror 156

PError 157

29: 158

158

158

158

Examples 159

«signal ()» 159

30: C 162

Examples 162

162

- void 162

31: 166

166

166

166

Examples 166

, 166

167

, 167

32: 170

170

Examples 170

C 170

171

172

174
Typedef 177
C 178
33: 183
183
Examples 183
183
= == 184
185
\ 0 186
( ) 186
188
realloc 188
189
190
- 192
194
196
, «» 198
200
201
scanf () 202
#define 203
203
205
- 206
'true' 208
double 209
34: 210
210
Examples 210
210
211

35: 212
..... 212
..... 212
..... 212
Examples 214
..... 214
"==" 214
"! =" 215
"!" 215
, ">" 215
, "<" 215
"> =" 216
"<=" 216
..... 216
..... 218
..... 218
..... 218
..... 218
..... 218
..... 219
Modulo 219
/ 220
..... 221
..... 221
..... 221
..... 221
..... 221
/ 222
/ 222
Comma Operator 224
..... 224
..... 224

225
225
225
225
226
226
227
227
227
228
228
228
228
229
_Alignof 231
231
36: 233
233
Examples 233
233
233
234
234
235
, , 235
37: 2D- 238
Examples 238
2D- 238
2D- 244
38: 246
246
246

246
Examples 246
goto 246
247
247
248
break continue 248
39: 250
250
250
250
251
Examples 251
count va_list 251
va_list 252
`printf ()` -like 253
256
40: 258
258
Examples 258
258
1 258
2. 259
Typedef 260
261
typename 261
41: 263
Examples 263
Pre / Post Increment / Decrement 263
42: , 265
265
265

265
266
266
266
267
268
269
269
269
270
270
270
270
270
271
271
272
/ 272
273
273
273
/ 273
Examples 274
274
274
274
274
43: 276
276
276
Examples 276
276
279

279
280
#if 0 281
282
282
282
( ) 283
284
288
__cplusplus C-externals C ++, C ++-name mangli 290
292
Variadic 293
44: **296**
296
296
Examples 296
296
297
297
45: **299**
299
Examples 300
300
301
302
303
304
46: **305**
305
305
305
305

305
Topoliges 305
305
306
306
306
306
307
307
Examples 308
308
309
n- 310
311
312
313
47: 317
317
Examples 317
317
318
318
#pragma once 318
319
: 319
319
319
321
321
, (IWYU) 321
322
324

48: 325
..... 325
..... 325
Examples 325
/ 325
C, C11-§6.5.2.5 / 9: 326
..... 326
..... 326
, ... 327
..... 327
, ... 327
49: 329
Examples 329
..... 329
..... 329
..... 330
50: 332
..... 332
..... 332
Examples 332
: fmod () 332
: fmodf (), fmodl () 333
- pow (), powf (), powl () 334
51: 336
..... 336
Examples 336
..... 336
Typedef Structs 337
..... 338
..... 340
..... 340

341

341

« » 342

342

343

- 344

52: 347

347

347

Examples 347

CppUTest 347

Unity 348

CMocka 349

53: 351

351

351

Examples 351

: strlen () 351

: strcpy (), strcat () 352

Comparsion: strcmp (), strncmp (), strcasecmp (), strncasecmp () 353

Tokenisation: strtok (), strtok_r () strtok_s () 355

/ : strchr (), strrchr () 358

359

359

361

strstr 362

362

363

strspn strcspn 364

365

365

366

strcpy() 366
snprintf() 367
strncat() 367
strncpy() 368
: atoi (), atof () (,) 369
/ 370
: strtoX 370
54: () 373
373
373
C-, , , C11: 373
C, C11, : 373
Examples 374
374
374
55: 376
376
376
377
Examples 377
(const) 377
377
378
56: 380
380
Examples 380
380
382
(C99) 383
384
384

385

386

386

57: 388

388

Examples 389

389

389

390

58: 392

392

392

392

Examples 392

392

393

sizeof 393

394

394

394

/ 396

396

397

398

399

399

: 400

: 400

- (&) 401

401

void * 401

402

402

403

, 405

405

405

405

406

406

409

void 409

59: 411

411

411

Examples 411

411

412

412

typedef 412

: 413

. 414

414

415

415

415

415

416

416

60: 418

418

418

418
418
Examples 419
419
421
421
422
422
422
424
realloc (ptr, 0) (ptr) 425
425
alloca: 427
427
428
61: 429
429
429
429
429
Examples 430
430
431
432
433
434
62: - 435
435
435
435
: 436
Examples 436

436
fprintf 437
438
getline () 438
example.txt 439
440
getline() 440
442
fscanf () 444
445
63: / 447
Examples 447
447
<inttypes.h> uintptr_t 447
: 448
448
449
printf () 451
452
453
456

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: c-language

It is an unofficial and free C Language 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 Language.

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

глава 1: Начало работы с языком C

замечания

C - универсальный, императивный язык программирования, поддерживающий структурированное программирование, лексическую переменную область и рекурсию, а система статического типа предотвращает многие непреднамеренные операции. По дизайну C предоставляет конструкции, которые эффективно сопоставляются с типичными машинными инструкциями, и поэтому он нашел долгосрочное применение в приложениях, которые ранее были закодированы на языке ассемблера, включая операционные системы, а также различное прикладное программное обеспечение для компьютеров от суперкомпьютеров до встроенных систем ,

Несмотря на свои низкоуровневые возможности, язык был разработан для поощрения кросс-платформенного программирования. Стандартно-совместимая и портативно написанная программа C может быть скомпилирована для очень широкого спектра компьютерных платформ и операционных систем с небольшими изменениями в ее исходном коде. Язык стал доступен на очень широком спектре платформ - от встроенных микроконтроллеров до суперкомпьютеров.

C был первоначально разработан Деннисом Ричи в период с 1969 по 1973 год в Bell Labs и использовался для повторной реализации операционных систем Unix . С тех пор он стал одним из наиболее широко используемых языков программирования всех времен, причем компиляторы C из разных поставщиков доступны для большинства существующих компьютерных архитектур и операционных систем.

Общие компиляторы

Процесс компиляции программы C отличается между компиляторами и операционными системами. Большинство операционных систем поставляются без компилятора, поэтому вам придется его установить. Некоторые общие варианты компиляторов:

- GCC, сборник компиляторов GNU
- clang: интерфейс семейства языков C для LLVM
- Инструменты сборки MSVC, Microsoft Visual C / C ++

Следующие документы должны дать вам хороший обзор о том, как начать работу с использованием нескольких наиболее распространенных компиляторов:

- Начало работы с Microsoft Visual C
- Начало работы с GCC

Поддержка версии компилятора C

Обратите внимание, что компиляторы имеют различные уровни поддержки для стандартного C, многие из которых еще не полностью поддерживают C99. Например, с момента выпуска в 2015 году MSVC поддерживает большую часть C99, но все же имеет некоторые важные исключения для поддержки самого языка (например, препроцессинг кажется несоответствующим) и для библиотеки C (например, `<tgmath.h>`), а также они обязательно документируют свои «варианты, зависящие от реализации». В Википедии есть таблица, показывающая поддержку, предлагаемую некоторыми популярными компиляторами.

Некоторые компиляторы (особенно GCC) предлагали или продолжают предлагать *расширения для компилятора,* которые реализуют дополнительные функции, которые производители компиляторов считают необходимыми, полезными или верят, могут стать частью будущей версии C, но в настоящее время они не являются частью любого стандарта C. Поскольку эти расширения специфичны для компилятора, их можно считать несовместимыми, а разработчики компилятора могут удалить или изменить их в более поздних версиях компилятора. Использование таких расширений обычно можно контролировать с помощью флагов компилятора.

Кроме того, у многих разработчиков есть компиляторы, которые поддерживают только определенные версии C, налагаемые средой или платформой, на которую они нацелены.

При выборе компилятора рекомендуется выбрать компилятор, который наилучшим образом поддерживает последнюю версию C, разрешенную для целевой среды.

Стиль кода (вне темы здесь):

Поскольку пробел в C невелик (то есть он не влияет на работу кода), программисты часто используют пробел, чтобы сделать код более понятным и понятным, это называется *стилем кода* . Это набор правил и рекомендаций, используемых при написании исходного кода. Он охватывает такие проблемы, как то, как строки должны быть отступом, должны ли использоваться пробелы или вкладки, как должны быть размещены фигурные скобки, как пространства должны использоваться вокруг операторов и скобок, как переменные должны быть названы и так далее.

Стиль кода не покрывается стандартом и в основном основан на мнениях (разные люди находят разные стили более легкими для чтения), поэтому он обычно считается вне темы на SO. Превосходный совет по стилю в собственном коде состоит в том, что последовательность является первостепенной - выбирайте или создавайте стиль и придерживайтесь его. Достаточно пояснить, что в общем использовании существуют разные именованные стили, которые часто выбирают программисты, а не создают их собственный стиль.

Некоторые общие стили отступа: стиль K & R, стиль Allman, стиль GNU и так далее. Некоторые из этих стилей имеют разные варианты. Например, Allman используется как обычный Allman или популярный вариант Allman-8. Информацию о некоторых популярных стилях можно найти в Википедии . Такие имена стиля берутся из стандартов, которые авторы или организации часто публикуют для использования многими людьми, способствующими их коду, чтобы каждый мог легко прочитать код, когда он знает стиль, например, руководство по форматированию GNU, которое составляет часть документ стандартов кодирования GNU .

Некоторые общие соглашения об именах: UpperCamelCase, lowerCamelCase, lower_case_with_underscore, ALL_CAPS и т. Д. Эти стили объединяются различными способами для использования с разными объектами и типами (например, макросы часто используют стиль ALL_CAPS)

Стиль K & R обычно рекомендуется для использования в документации SO, тогда как более эзотерические стили, такие как Pico, обескуражены.

Библиотеки и API, не охваченные стандартом C (и, следовательно, не относящиеся к теме):

- POSIX API (покрытие, например, PThreads , Sockets , Signals)

Версии

Версия	стандарт	Дата публикации
K & P	н /	1978-02-22
C89	ANSI X3.159-1989	1989-12-14
C90	ISO / IEC 9899: 1990	1990-12-20
C95	ISO / IEC 9899 / AMD1: 1995	1995-03-30
C99	ISO / IEC 9899: 1999	1999-12-16
C11	ISO / IEC 9899: 2011	2011-12-15

Examples

Привет, мир

Чтобы создать простую программу на C, которая выводит на экран *«Hello, World»* ,

используйте текстовый редактор для создания нового файла (например, `hello.c` - расширение файла должно быть `.c`), содержащее следующий исходный код:

Привет

```
#include <stdio.h>

int main(void)
{
    puts("Hello, World");
    return 0;
}
```

Демо-версия Live на Coliru

Давайте посмотрим на эту простую программу за строкой

```
#include <stdio.h>
```

Эта строка сообщает компилятору включить в стандартный `stdio.h` содержимое стандартного файла `stdio.h` библиотеки. Заголовки обычно представляют собой файлы, содержащие объявления функций, макросы и типы данных, и вы должны включать заголовочный файл перед их использованием. Эта строка включает `stdio.h` поэтому она может вызывать функцию `puts()` .

Подробнее о заголовках.

```
int main(void)
```

Эта строка начинает определение функции. В нем указано имя функции (`main`), тип и количество ожидаемых аргументов (`void` , значение none) и тип значения, возвращаемого этой функцией (`int`). Выполнение программы начинается с функции `main()` .

```
{
    …
}
```

Кудрявые фигурные скобки используются парами, чтобы указать, где начинается и заканчивается блок кода. Их можно использовать многими способами, но в этом случае они указывают, где начинается и заканчивается функция.

```
    puts("Hello, World");
```

Эта строка вызывает функцию `puts()` для вывода текста на стандартный вывод (по умолчанию - экран), за которым следует новая строка. Строка, которая будет выводиться, включена в круглые скобки.

`"Hello, World"` - это строка, которая будет записана на экран. В C каждое строковое литеральное значение должно быть внутри двойных кавычек `"…"` .

Подробнее о строках.

В C-программах каждое утверждение должно заканчиваться точкой с запятой (т.е `;`).

```
    return 0;
```

Когда мы определили `main()` , мы объявили его как функцию, возвращающую `int` , то есть нужно вернуть целое число. В этом примере мы возвращаем целочисленное значение 0, которое используется для указания успешного выхода программы. После `return 0;` , процесс выполнения завершится.

Редактирование программы

Простые текстовые редакторы включают `vim` или `gedit` в Linux или `Notepad` в Windows. Кросс-платформенные редакторы также включают `Visual Studio Code` или `Sublime Text` .

Редактор должен создавать текстовые текстовые файлы, а не RTF или другой другой формат.

Компиляция и запуск программы

Для запуска программы этот исходный файл (`hello.c`) сначала необходимо скомпилировать в исполняемый файл (например, `hello` в системе Unix / Linux или `hello.exe` в Windows). Это делается с использованием компилятора для языка C.

Подробнее о компиляции

Компилировать с использованием GCC

GCC (сборник компиляторов GNU) - широко используемый компилятор C. Чтобы использовать его, откройте терминал, используйте командную строку, чтобы перейти к местоположению исходного файла, а затем запустите:

```
gcc hello.c -o hello
```

Если в исходном коде (`hello.c`) не обнаружены ошибки, компилятор создаст **двоичный**

файл , имя которого задается аргументом в параметре командной строки `-o` (`hello`). Это окончательный исполняемый файл.

Мы также можем использовать параметры предупреждения `-Wall -Wextra -Werror` , которые помогают выявлять проблемы, которые могут привести к сбою программы или возникновению неожиданных результатов. Они не нужны для этой простой программы, но это способ их добавления:

```
gcc -Wall -Wextra -Werror -o hello hello.c
```

Использование компилятора clang

Чтобы скомпилировать программу, используя `clang` вы можете использовать:

```
clang -Wall -Wextra -Werror -o hello hello.c
```

По дизайну параметры командной строки `clang` аналогичны параметрам GCC.

Использование компилятора Microsoft C из командной строки

При использовании компилятора Microsoft `cl.exe` в системе Windows, которая поддерживает Visual Studio, и если все переменные среды установлены, этот пример C может быть скомпилирован с использованием следующей команды, которая приведет к выполнению исполняемого файла `hello.exe` в каталоге, в котором выполняется команда (Существуют опции предупреждения, такие как `/W3` для `cl` , примерно аналогичные `-Wall` т. Д. Для GCC или clang).

```
cl hello.c
```

Выполнение программы

После компиляции двоичный файл может быть выполнен, введя `./hello` в терминал. После выполнения скомпилированная программа напечатает `Hello, World` , а затем новую строку, в командной строке.

Оригинал «Привет, мир!» в K & R C

Ниже приводится оригинал «Hello, World!». программа из книги «Язык программирования C » Брайана Кернигана и Денниса Ричи (Ritchie был оригинальным разработчиком языка программирования C в Bell Labs), называемым «K & R»:

K & P

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Обратите внимание, что язык программирования C не был стандартизован во время написания первого издания этой книги (1978) и что эта программа, вероятно, не будет компилироваться на большинстве современных компиляторов, если они не будут проинструктированы принять код C90.

Этот самый первый пример в книге K & R теперь считается некачественным, отчасти потому, что ему не хватает явного типа возврата для `main()` а частично потому, что ему не хватает оператора `return` . Второе издание книги было написано для старого стандарта C89. В C89 тип `main` будет по умолчанию для `int` , но пример K & R не возвращает определенное значение в среду. В стандартах C99 и более поздних версиях требуется тип возврата, но безопасно исключить оператор `return main` (и только `main`) из-за специального случая, введенного с C99 5.1.2.2.3, - это эквивалентно возврату 0 , что указывает на успех.

Рекомендуемая и наиболее переносимая форма `main` для размещенных систем - `int main (void)` когда программа не использует аргументы командной строки или `int main(int argc, char **argv)` когда программа использует аргументы командной строки.

C90 §5.1.2.2.3 Окончание **программы**

> Возврат от начального вызова к `main` функции эквивалентен вызову функции `exit` со значением, возвращаемым `main` функцией в качестве аргумента. Если `main` функция выполняет возврат, который не задает значения, статус завершения, возвращаемый в среду хоста, не определен.

C90 §6.6.6.4 **Оператор `return`**

> Если выполняется оператор `return` без выражения, а значение вызова функции используется вызывающим, поведение не определено. Достижение `}` которое завершает функцию, эквивалентно выполнению оператора `return` без выражения.

C99 §5.1.2.2.3 Окончание **программы**

> Если тип возврата `main` функции является типом, совместимым с `int` , возврат от начального вызова к `main` функции эквивалентен вызову функции `exit` со значением, возвращаемым `main` функцией в качестве аргумента; достигая `}` который завершает `main` функцию, возвращает значение 0. Если тип возврата несовместим с `int` , статус завершения, возвращаемый в среду хоста, не указан.

Прочитайте Начало работы с языком C онлайн: https://riptutorial.com/ru/c/topic/213/начало-работы-с-языком-с

глава 2: - классификация символов и конверсия

Examples

Классификация символов, считанных из потока

```
#include <ctype.h>
#include <stdio.h>

typedef struct {
  size_t space;
  size_t alnum;
  size_t punct;
} chartypes;

chartypes classify(FILE *f) {
  chartypes types = { 0, 0, 0 };
  int ch;

  while ((ch = fgetc(f)) != EOF) {
    types.space += !!isspace(ch);
    types.alnum += !!isalnum(ch);
    types.punct += !!ispunct(ch);
  }

  return types;
}
```

Функция `classify` считывает символы из потока и подсчитывает количество пробелов, буквенно-цифровых символов и знаков препинания. Это позволяет избежать нескольких подводных камней.

- При чтении символа из потока результат сохраняется как `int` , так как в противном случае между чтением `EOF` (маркером конца файла) и символом, имеющим один и тот же битовый шаблон, была бы двусмысленность.
- Функции классификации символов (например, `isspace`) ожидают, что их аргумент будет *либо представлен как* `unsigned char` *, либо значение макроса* `EOF` . Поскольку это именно то, что возвращает `fgetc` , здесь нет необходимости в преобразовании.
- Возвращаемое значение функций классификации символов различает только ноль (значение `false`) и ненулевой (что означает `true`). Для подсчета количества вхождений это значение нужно преобразовать в 1 или 0, что делается двойным отрицанием `!!` ,

Классификация символов из строки

```
#include <ctype.h>
```

```
#include <stddef.h>

typedef struct {
  size_t space;
  size_t alnum;
  size_t punct;
} chartypes;

chartypes classify(const char *s) {
  chartypes types = { 0, 0, 0 };
  const char *p;
  for (p= s; p != '\0'; p++) {
    types.space += !!isspace((unsigned char)*p);
    types.alnum += !!isalnum((unsigned char)*p);
    types.punct += !!ispunct((unsigned char)*p);
  }

  return types;
}
```

Функция `classify` проверяет все символы из строки и подсчитывает количество пробелов, буквенно-цифровых символов и знаков препинания. Это позволяет избежать нескольких подводных камней.

- Функции классификации символов (например, `isspace`) ожидают, что их аргумент будет *либо представлен как* `unsigned char` , *либо значение макроса* `EOF` .
- Выражение `*p` имеет тип `char` и поэтому должно быть преобразовано в соответствии с указанной выше формулировкой.
- Тип `char` определяется как эквивалентный знаку `signed char` или `unsigned char` .
- Когда `char` эквивалентен `unsigned char` , нет никакой проблемы, так как все возможные значения типа `char` представляются как `unsigned char` .
- Когда `char` эквивалентен `signed char` , он должен быть преобразован в `unsigned char` прежде чем передавать его в функции классификации символов. И хотя значение символа может измениться из-за этого преобразования, это именно то, что ожидают эти функции.
- Возвращаемое значение функций классификации символов различает только ноль (значение `false`) и ненулевой (что означает `true`). Для подсчета количества вхождений это значение нужно преобразовать в 1 или 0, что делается двойным отрицанием `!!` ,

Вступление

Заголовок `ctype.h` является частью стандартной библиотеки C. Он предоставляет функции для классификации и преобразования символов.

Все эти функции принимают один параметр, `int` который должен быть либо *EOF,* либо представлен как символ без знака.

Имена классификационных функций имеют префикс «is». Каждый возвращает целое

ненулевое значение (TRUE), если переданный ему символ удовлетворяет соответствующему условию. Если условие не выполняется, функция возвращает нулевое значение (FALSE).

Эти классифицирующие функции работают, как показано, принимая значение по умолчанию для языка C:

```
int a;
int c = 'A';
a = isalpha(c); /* Checks if c is alphabetic (A-Z, a-z), returns non-zero here. */
a = isalnum(c); /* Checks if c  is alphanumeric (A-Z, a-z, 0-9), returns non-zero here. */
a = iscntrl(c); /* Checks is c is a control character (0x00-0x1F, 0x7F), returns zero here. */
a = isdigit(c); /* Checks if c is a digit (0-9), returns zero here. */
a = isgraph(c); /* Checks if c has a graphical representation (any printing character except
space), returns non-zero here. */
a = islower(c); /* Checks if c is a lower-case letter (a-z), returns zero here. */
a = isprint(c); /* Checks if c is any printable character (including space), returns non-zero
here. */
a = isupper(c); /* Checks if c is a upper-case letter (a-z), returns zero here. */
a = ispunct(c); /* Checks if c is a punctuation character, returns zero here. */
a = isspace(c); /* Checks if c is a white-space character, returns zero here. */
a = isupper(c); /* Checks if c is an upper-case letter (A-Z), returns non-zero here. */
a = isxdigit(c); /* Checks if c is a hexadecimal digit (A-F, a-f, 0-9), returns non-zero here.
*/
```

C99

```
a = isblank(c); /* Checks if c is a blank character (space or tab), returns non-zero here. */
```

Существует две функции преобразования. Они называются с использованием префикса ' to'. Эти функции имеют тот же аргумент, что и выше. Однако возвращаемое значение не является простым нулем или ненулевым, но переданный аргумент каким-то образом изменился.

Эти функции преобразования функционируют, как показано, при условии, что язык по умолчанию C:

```
int a;
int c = 'A';

/* Converts c to a lower-case letter (a-z).
 * If conversion is not possible the unchanged value is returned.
 * Returns 'a' here.
 */
a = tolower(c);

/* Converts c to an upper-case letter (A-Z).
 * If conversion is not possible the unchanged value is returned.
 * Returns 'A' here.
 */
a = toupper(c);
```

Ниже приведена приведенная ниже информация cflusplus.com o том, как исходный 127-

символьный набор ASCII рассматривается каждой из классифицирующих функций типа (а • указывает, что функция возвращает ненулевое значение для этого символа)

Значения ASCII	персонажи	iscntrl	ISBLANK	isspace	ISUPPER	ISLOWER
0x00 .. 0x08	NUL (другие управляющие коды)	•				
0x09	tab ('\ t')	•	•	•		
0x0A .. 0x0D	(управляющие коды белого пространства: '\ f', '\ v', '\ n', '\ r')	•		•		
0x0E .. 0x1F	(другие управляющие коды)	•				
0x20	пространство (' ')		•	•		
0x21 .. 0x2F	! "# $% & '() * +, -. /					
0x30 .. 0x39	0123456789					
0x3a .. 0x40	:;? <=> @					
0x41 .. 0x46	ABCDEF				•	
0x47 .. 0x5A	GHIJKLMNOPQRSTUVWXYZ				•	
0x5B .. 0x60	[] ^ _ `					
0x61 .. 0x66	ABCDEF					•
0x67 .. 0x7A	ghijklmnopqrstuvwxyz					•
0x7B .. 0x7E	{} ~ Бар					
0x7F	(DEL)	•				

Прочитайте - классификация символов и конверсия онлайн:
https://riptutorial.com/ru/c/topic/6846/-ctype-h----классификация-символов-и-конверсия

глава 3: Interprocess Communication (IPC)

Вступление

Механизмы межпроцессной коммуникации (МПК) позволяют различным независимым процессам взаимодействовать друг с другом. Стандарт C не предусматривает каких-либо механизмов МПК. Следовательно, все такие механизмы определяются операционной системой хоста. POSIX определяет обширный набор механизмов МПК; Windows определяет другой набор; и другие системы определяют их собственные варианты.

Examples

семафоры

Семафоры используются для синхронизации операций между двумя или несколькими процессами. POSIX определяет два разных набора функций семафора:

1. «System V IPC» - `semctl()` , `semop()` , `semget()` .
2. «Семафоры POSIX» - `sem_close()` , `sem_destroy()` , `sem_getvalue()` , `sem_init()` , `sem_open()` , `sem_post()` , `sem_trywait()` , `sem_unlink()` .

В этом разделе описываются семафоры System V IPC, так называемые, потому что они возникли в Unix System V.

Во-первых, вам нужно будет включить нужные заголовки. Для старых версий POSIX требуется `#include <sys/types.h>` ; современный POSIX, и большинство систем не требуют этого.

```
#include <sys/sem.h>
```

Затем вам нужно определить ключ как у родителя, так и у ребенка.

```
#define KEY 0x1111
```

Этот ключ должен быть одинаковым в обеих программах или они не будут ссылаться на одну и ту же структуру IPC. Существуют способы создания согласованного ключа без жесткого кодирования его значения.

Далее, в зависимости от вашего компилятора, вам может понадобиться или не нужно делать этот шаг: объявить объединение для операций семафора.

```
union semun {
    int val;
```

```
    struct semid_ds *buf;
    unsigned short  *array;
};
```

Затем определите структуры *try* (`semwait`) и *raise* (`semsignal`). Имена P и V происходят от голландских

```
struct sembuf p = { 0, -1, SEM_UNDO}; # semwait
struct sembuf v = { 0, +1, SEM_UNDO}; # semsignal
```

Теперь начните с получения идентификатора для вашего семафора IPC.

```
int id;
// 2nd argument is number of semaphores
// 3rd argument is the mode (IPC_CREAT creates the semaphore set if needed)
if ((id = semget(KEY, 1, 0666 | IPC_CREAT) < 0) {
    /* error handling code */
}
```

В родителе инициализируйте семафор, чтобы иметь счетчик из 1.

```
union semun u;
u.val = 1;
if (semctl(id, 0, SETVAL, u) < 0) { // SETVAL is a macro to specify that you're setting the
value of the semaphore to that specified by the union u
    /* error handling code */
}
```

Теперь вы можете уменьшать или увеличивать семафор по мере необходимости. В начале вашего критического раздела вы уменьшаете счетчик с помощью функции `semop()` :

```
if (semop(id, &p, 1) < 0) {
    /* error handling code */
}
```

Чтобы увеличить семафор, вы используете `&v` вместо `&p` :

```
if (semop(id, &v, 1) < 0) {
    /* error handling code */
}
```

Обратите внимание, что каждая функция возвращает `0` при успехе и `-1` при сбое. Не проверять эти статусы возврата могут привести к разрушительным проблемам.

Пример 1.1: Гонки с потоками

В приведенной ниже программе будет выполняться процесс `fork` для дочернего элемента, а родительский и дочерний объекты будут пытаться печатать символы на терминале без какой-либо синхронизации.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main()
{
    int pid;
    pid =  fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork"); exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefgh";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
        }
    }
    else
    {
        char *s = "ABCDEFGH";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
        }
    }
}
```

Выход (1-й прогон):

```
aAABaBCbCbDDcEEcddeFFGGHHeffgghh
```

(2-й прогон):

```
aabbccAABddBCeeCffgDDghEEhFFGGHH
```

Компиляция и запуск этой программы должны давать вам разные результаты каждый раз.

Пример 1.2: Избегайте гонок с помощью семафоров

Изменив *пример 1.1* для использования семафоров, мы имеем:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY 0x1111

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
};

struct sembuf p = { 0, -1, SEM_UNDO};
struct sembuf v = { 0, +1, SEM_UNDO};

int main()
{
    int id = semget(KEY, 1, 0666 | IPC_CREAT);
    if(id < 0)
    {
        perror("semget"); exit(11);
    }
    union semun u;
    u.val = 1;
    if(semctl(id, 0, SETVAL, u) < 0)
    {
        perror("semctl"); exit(12);
    }
    int pid;
    pid =  fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork"); exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefgh";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(13);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(14);
            }
```

```
            sleep(rand() % 2);
        }
    }
    else
    {
        char *s = "ABCDEFGH";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(15);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(16);
            }

            sleep(rand() % 2);
        }
    }
}
```

Выход:

```
aabbAABBCCccddeeDDffEEFFGGHHgghh
```

Компиляция и запуск этой программы даст вам один и тот же результат каждый раз.

Прочитайте Interprocess Communication (IPC) онлайн:
https://riptutorial.com/ru/c/topic/10564/interprocess-communication--ipc-

глава 4: Typedef

Вступление

Механизм `typedef` позволяет создавать псевдонимы для других типов. Он не создает новые типы. Люди часто используют `typedef` чтобы улучшить переносимость кода, дать псевдонимы структуре или типам объединения или создать псевдонимы для типов функций (или указателей функций).

В стандарте C `typedef` классифицируется как «класс хранения» для удобства; это происходит синтаксически, когда могут появляться классы хранения, такие как `static` или `extern` .

Синтаксис

- typedef existing_name alias_name;

замечания

Недостатки Typedef

`typedef` может привести к загрязнению пространства имен в больших программах на C.

Недостатки Typedef Structs

Кроме того, `typedef` 'd structs без имени тега являются основной причиной ненужного наложения отношений упорядочения между заголовочными файлами.

Рассматривать:

```
#ifndef FOO_H
#define FOO_H 1

#define FOO_DEF (0xDEADBABE)

struct bar; /* forward declaration, defined in bar.h*/

struct foo {
    struct bar *bar;
};

#endif
```

С таким определением, не используя `typedefs` , блок компиляции может включать `foo.h` для определения `FOO_DEF` . Если он не пытается разыменовать элемент `bar` структуры `foo` тогда

нет необходимости включать файл `bar.h`

Typedef vs #define

`#define` - это `#define` C, которая также используется для определения псевдонимов для разных типов данных, аналогичных `typedef` но со следующими отличиями:

- `typedef` ограничивается предоставлением символических имен только для типов, где `#define` может использоваться для определения псевдонимов для значений.
- интерпретация `typedef` выполняется компилятором, тогда как операторы `#define` обрабатываются предварительным процессором.
- Обратите внимание, что `#define cptr char *` за которым следует `cptr a, b;` не делает то же самое, что `typedef char *cptr;` а затем `cptr a, b;` , C `#define` , `b` - простая переменная `char` , но она также является указателем с `typedef` .

Examples

Typedef для структур и союзов

Вы можете дать псевдоним именам для `struct` :

```
typedef struct Person {
    char name[32];
    int age;
} Person;

Person person;
```

По сравнению с традиционным способом объявления структур, программистам не требуется иметь `struct` каждый раз, когда они объявляют экземпляр этой структуры.

Обратите внимание, что имя `Person` (в отличие от `struct Person`) не определено до окончательной точки с запятой. Таким образом, для связанных списков и древовидных структур, которые должны содержать указатель на один и тот же тип структуры, вы должны использовать либо:

```
typedef struct Person {
    char name[32];
    int age;
    struct Person *next;
} Person;
```

или же:

```
typedef struct Person Person;
```

```
struct Person {
    char name[32];
    int age;
    Person *next;
};
```

Использование `typedef` для типа `union` очень похоже.

```
typedef union Float Float;

union Float
{
    float f;
    char  b[sizeof(float)];
};
```

Аналогичную структуру можно использовать для анализа байтов, которые составляют значение `float` .

Простое использование Typedef

Для предоставления коротких имен типу данных

Вместо:

```
long long int foo;
struct mystructure object;
```

можно использовать

```
/* write once */
typedef long long ll;
typedef struct mystructure mystruct;

/* use whenever needed */
ll foo;
mystruct object;
```

Это уменьшает объем ввода, если тип используется многократно в программе.

Улучшение переносимости

Атрибуты типов данных различаются для разных архитектур. Например, `int` может быть двухбайтным типом в одной реализации и 4-байтным типом в другом. Предположим, что программа должна использовать 4-байтовый тип для правильной работы.

В одной реализации пусть размер `int` равен 2 байтам, а `long` - 4 байта. В другом случае пусть размер `int` равен 4 байтам, а `long` - 8 байтов. Если программа написана с использованием второй реализации,

```
/* program expecting a 4 byte integer */
int foo; /* need to hold 4 bytes to work */
/* some code involving many more ints */
```

Для запуска программы в первой реализации все объявления `int` должны быть изменены на `long` .

```
/* program now needs long */
long foo; /*need to hold 4 bytes to work */
/* some code involving many more longs - lot to be changed */
```

Чтобы этого избежать, можно использовать `typedef`

```
/* program expecting a 4 byte integer */
typedef int myint; /* need to declare once - only one line to modify if needed */
myint foo; /* need to hold 4 bytes to work */
/* some code involving many more myints */
```

Тогда только инструкция `typedef` нужно будет менять каждый раз, а не рассматривать всю программу.

C99

Заголовок `<stdint.h>` и связанный с `<inttypes.h>` заголовок `<inttypes.h>` определяют стандартные имена типов (с использованием `typedef`) для целых чисел различного размера, и эти имена часто являются лучшим выбором в современном коде, для которого требуются целые числа фиксированного размера. Например, `uint8_t` представляет собой неподписанный 8-разрядный целочисленный тип; `int64_t` - это подписанный 64-разрядный целочисленный тип. Тип `uintptr_t` представляет собой целочисленный тип без знака, достаточно большой, чтобы содержать любой указатель на объект. Эти типы теоретически необязательны, но редко они недоступны. Существуют такие варианты, как `uint_least16_t` (наименьший беззнаковый целочисленный тип с не менее 16 битами) и `int_fast32_t` (самый быстрый тип целочисленного типа со `int_fast32_t` не менее 32 бит). Кроме того, `intmax_t` и `uintmax_t` являются наибольшими целыми типами, поддерживаемыми реализацией. Эти типы являются обязательными.

Чтобы указать использование или улучшить читаемость

Если набор данных имеет определенную цель, можно использовать `typedef` чтобы дать ему значимое имя. Более того, если свойство данных изменяется так, что базовый тип должен измениться, нужно будет изменить только оператор `typedef` , а не рассматривать всю программу.

Typedef для указателей функций

Мы можем использовать `typedef` для упрощения использования указателей на функции.

Представьте, что у нас есть некоторые функции, имеющие одну и ту же подпись, которые используют свой аргумент для распечатки чего-то по-разному:

```
#include<stdio.h>

void print_to_n(int n)
{
    for (int i = 1; i <= n; ++i)
        printf("%d\n", i);
}

void print_n(int n)
{
    printf("%d\n, n);
}
```

Теперь мы можем использовать `typedef` для создания названного типа указателя функции, называемого printer:

```
typedef void (*printer_t)(int);
```

Это создает тип с именем `printer_t` для указателя на функцию, которая принимает единственный аргумент `int` и ничего не возвращает, что соответствует сигнатуре функций, которые мы имеем выше. Чтобы использовать его, мы создаем переменную созданного типа и присваиваем ей указатель на одну из рассматриваемых функций:

```
printer_t p = &print_to_n;
void (*p)(int) = &print_to_n; // This would be required without the type
```

Затем для вызова функции, на которую указывает переменная указателя функции:

```
p(5);           // Prints 1 2 3 4 5 on separate lines
(*p)(5);        // So does this
```

Таким образом, `typedef` позволяет упростить синтаксис при работе с указателями функций. Это становится более очевидным, когда указатели на функции используются в более сложных ситуациях, таких как аргументы для функций.

Если вы используете функцию, которая принимает указатель на функцию в качестве параметра без определенного типа указателя функции, определение функции будет,

```
void foo (void (*printer)(int), int y){
    //code
    printer(y);
    //code
}
```

Однако, с `typedef` это:

```
void foo (printer_t printer, int y){
    //code
    printer(y);
    //code
}
```

Аналогично, функции могут возвращать указатели на функции и снова, использование `typedef` может упростить синтаксис при этом.

Классическим примером является `signal` функция от `<signal.h>` . Декларация для него (из стандарта C):

```
void (*signal(int sig, void (*func)(int)))(int);
```

Это функция, которая принимает два аргумента - `int` и указатель на функцию, которая принимает `int` как аргумент и ничего не возвращает - и которая возвращает указатель на функцию как свой второй аргумент.

Если мы определили тип `SigCatcher` как псевдоним для указателя на тип функции:

```
typedef void (*SigCatcher)(int);
```

то мы могли бы объявить `signal()` используя:

```
SigCatcher signal(int sig, SigCatcher func);
```

В целом это легче понять (хотя стандарт C не решил определить тип для выполнения задания). Функция `signal` принимает два аргумента: `int` и `SigCatcher` , и возвращает `SigCatcher` где `SigCatcher` является указателем на функцию, которая принимает аргумент `int` и ничего не возвращает.

Хотя использование `typedef` имен для указателей на типы функций облегчает жизнь, это также может привести к путанице для других, которые впоследствии будут поддерживать ваш код, поэтому используйте с осторожностью и правильной документацией. См. Также Указатели функций .

Прочитайте Typedef онлайн: https://riptutorial.com/ru/c/topic/2681/typedef

глава 5: Valgrind

Синтаксис

- *имя-программы* Valgrind *необязательные-аргументы <тест вход*

замечания

Valgrind - это инструмент отладки, который можно использовать для диагностики ошибок управления памятью в программах на языке C. Valgrind может использоваться для обнаружения ошибок, таких как использование недопустимого указателя, включая запись или чтение за выделенным пространством или неправильный вызов функции `free()` . Он также может использоваться для улучшения приложений посредством функций, которые выполняют профилирование памяти.

Для получения дополнительной информации см. Http://valgrind.org .

Examples

Запуск Valgrind

```
valgrind ./my-program arg1 arg2 < test-input
```

Это запустит вашу программу и опубликует отчет о любых распределениях и отчислениях, которые она сделала. Он также предупредит вас об общих ошибках, таких как использование неинициализированной памяти, указатели разыменования в странные места, списание конца блоков, выделенных с помощью malloc, или отказ от свободных блоков.

Добавление флагов

Вы также можете включить дополнительные тесты, такие как:

```
valgrind -q --tool=memcheck --leak-check=yes ./my-program arg1 arg2 < test-input
```

См. Valgrind --help для получения дополнительной информации о (многих) вариантах или просмотрите документацию по адресу http://valgrind.org/ для получения подробной информации о том, что означает выход.

Bytes lost - Забыв бесплатно

Вот программа, которая вызывает malloc, но не бесплатную:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    char *s;

    s = malloc(26); // the culprint

    return 0;
}
```

Без дополнительных аргументов valgrind не будет искать эту ошибку.

Но если мы `--leak-check=yes` или `--tool=memcheck` , он будет жаловаться и отображать строки, отвечающие за эти утечки памяти, если программа была скомпилирована в режиме отладки:

```
$ valgrind -q --leak-check=yes ./missing_free
==4776== 26 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4776==    at 0x4024F20: malloc (vg_replace_malloc.c:236)
==4776==    by 0x80483F8: main (missing_free.c:9)
==4776==
```

Если программа не скомпилирована в режиме отладки (например, с флагом `-g` в GCC), она все равно покажет нам, где утечка произошла с точки зрения соответствующей функции, но не для строк.

Это позволяет нам вернуться и посмотреть, какой блок был выделен в этой строке, и попытаться проследить вперед, чтобы понять, почему он не был освобожден.

Наиболее часто встречающиеся ошибки при использовании Valgrind

Valgrind предоставляет вам *строки, в которых произошла ошибка* в конце каждой строки в формате `(file.c:line_no)` . Ошибки в valgrind суммируются следующим образом:

```
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Наиболее распространенные ошибки:

1. **Незаконные ошибки чтения / записи**

```
==8451== Invalid read of size 2
==8451==    at 0x4E7381D: getenv (getenv.c:84)
==8451==    by 0x4EB1559: __libc_message (libc_fatal.c:80)
==8451==    by 0x4F5256B: __fortify_fail (fortify_fail.c:37)
==8451==    by 0x4F5250F: __stack_chk_fail (stack_chk_fail.c:28)
==8451==    by 0x40059C: main (valg.c:10)
==8451==  Address 0x700000007 is not stack'd, malloc'd or (recently) free'd
```

Это происходит, когда код начинает получать доступ к памяти, которая не принадлежит

программе. Размер доступной памяти также дает вам представление о том, какая переменная была использована.

2. **Использование неинициализированных переменных**

```
==8795== 1 errors in context 5 of 8:
==8795== Conditional jump or move depends on uninitialised value(s)
==8795==    at 0x4E881AF: vfprintf (vfprintf.c:1631)
==8795==    by 0x4E8F898: printf (printf.c:33)
==8795==    by 0x400548: main (valg.c:7)
```

Согласно ошибке, в строке 7 `main` of `valg.c` вызов `printf()` передал неинициализированную переменную `printf` .

3. **Незаконное освобождение памяти**

```
==8954== Invalid free() / delete / delete[] / realloc()
==8954==    at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x4005A8: main (valg.c:10)
==8954==  Address 0x5203040 is 0 bytes inside a block of size 240 free'd
==8954==    at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x40059C: main (valg.c:9)
==8954==  Block was alloc'd at
==8954==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x40058C: main (valg.c:7)
```

Согласно valgrind, код освободил память незаконно (второй раз) в *строке 10* `valg.c` , тогда как она уже была освобождена в *строке 9* , а сам блок был выделен памятью в *строке 7* .

Прочитайте Valgrind онлайн: https://riptutorial.com/ru/c/topic/2674/valgrind

глава 6: X-макросы

Вступление

X-макросы - это метод на основе препроцессора для минимизации повторяющегося кода и поддержания соответствия данных / кода. Множественные отличия макросов, основанные на общем наборе данных, поддерживаются путем представления всей группы расширений с помощью одного главного макроса с заменяющим текстом этого макроса, состоящим из последовательности расширений внутреннего макроса, по одному для каждой базы данных. Внутренний макрос традиционно называется `X()` , отсюда и название техники.

замечания

Предполагается, что пользователь основного макроса в стиле X-макроса предоставит свое собственное определение для внутреннего макроса `X()` и в пределах его возможностей для расширения основного макроса. Таким образом, внутренние макро-ссылки мастера расширяются в соответствии с определением пользователя `X()` . Таким образом, количество повторяющегося кода шаблона в исходном файле может быть уменьшено (появляется только один раз, в заменяющем тексте `X()`), как это предпочитают сторонники философии «Не повторяйте себя» (DRY).

Кроме того, переопределяя `X()` и расширяя мастер-макрос один или несколько дополнительных раз, макросы X могут облегчить сохранение соответствующих данных и кода - одно расширение макроса объявляет данные (например, как элементы массива или элементы перечисления), и другие расширения производят соответствующий код.

Хотя имя «X-macro» происходит от традиционного имени внутреннего макроса, этот метод не зависит от этого конкретного имени. Любое допустимое имя макроса можно использовать на своем месте.

Критики включают

- исходные файлы, которые полагаются на макросы X, труднее читать;
- как и все макросы, макросы X строго текстовые - они по своей сути не обеспечивают безопасность любого типа; а также
- X для *генерации кода* . По сравнению с альтернативами, основанными на вызывающих функциях, макросы X эффективно делают код более крупным.

Хорошее объяснение макросов X можно найти в статье Рэнди Майерса [X-Macros] у доктора Доббса (http://www.drdobbs.com/the-new-cx-macros/184401387) .

Examples

Тривиальное использование X-макросов для printfs

```
/* define a list of preprocessor tokens on which to call X */
#define X_123 X(1) X(2) X(3)

/* define X to use */
#define X(val) printf("X(%d) made this print\n", val);
X_123
#undef X
/* good practice to undef X to facilitate reuse later on */
```

В этом примере препроцессор генерирует следующий код:

```
printf("X(%d) made this print\n", 1);
printf("X(%d) made this print\n", 2);
printf("X(%d) made this print\n", 3);
```

Значение и идентификатор Enum

```
/* declare items of the enum */
#define FOREACH \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */

/* define the enum values */
#define X(id) MyEnum_ ## id,
enum MyEnum { FOREACH };
#undef X

/* convert an enum value to its identifier */
const char * enum2string(int enumValue)
{
    const char* stringValue = NULL;
#define X(id) if (enumValue == MyEnum_ ## id) stringValue = #id;
    FOREACH
#undef X
    return stringValue;
}
```

Затем вы можете использовать перечисленное значение в своем коде и легко распечатать его идентификатор, используя:

```
printf("%s\n", enum2string(MyEnum_item2));
```

Расширение: укажите макрос X как аргумент

Метод X-macro можно обобщить, указав имя макроса «X» как аргумент главного макроса. Это дает преимущества, позволяющие избежать конфликтов имен макросов и разрешить использование макроса общего назначения в качестве макроса «X».

Как всегда с макросами X, главный макрос представляет список элементов, значение которых специфично для этого макроса. В этом варианте такой макрос может быть определен следующим образом:

```
/* declare list of items */
#define ITEM_LIST(X) \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */
```

Затем можно сгенерировать код для печати имен элементов следующим образом:

```
/* define macro to apply */
#define PRINTSTRING(value) printf( #value "\n");

/* apply macro to the list of items */
ITEM_LIST(PRINTSTRING)
```

Это расширяется до этого кода:

```
printf( "item1" "\n"); printf( "item2" "\n"); printf( "item3" "\n");
```

В отличие от стандартных макросов X, где «X» - это встроенная характеристика главного макроса, с этим стилем может быть ненужным или даже нежелательным после этого определять неопределенный макрос, используемый в качестве аргумента (`PRINTSTRING` в этом примере).

Генерация кода

X-Macros можно использовать для генерации кода, написав повторяющийся код: перебирать список для выполнения некоторых задач или объявлять набор констант, объектов или функций.

Здесь мы используем X-макросы для объявления перечисления, содержащего 4 команды, и карту их имен в виде строк

Затем мы можем напечатать строковые значения перечисления.

```
/* All our commands */
#define COMMANDS(OP) OP(Open) OP(Close) OP(Save) OP(Quit)

/* generate the enum Commands: {cmdOpen, cmdClose, cmdSave, cmdQuit, }; */
#define ENUM_NAME(name) cmd##name,
enum Commands {
  COMMANDS(ENUM_NAME)
```

```
};
#undef ENUM_NAME

/* generate the string table */
#define COMMAND_OP(name) #name,
const char* const commandNames[] = {
  COMMANDS(COMMAND_OP)
};
#undef COMMAND_OP

/* the following prints "Quit\n": */
printf("%s\n", commandNames[cmdQuit]());
```

Аналогично, мы можем создать таблицу переходов для вызова функций по значению перечисления.

Для этого требуется, чтобы все функции имели одну и ту же подпись. Если они не принимают аргументов и возвращают int, мы помещаем это в заголовок с определением перечисления:

```
/* declare all functions as extern */
#define EXTERN_FUNC(name) extern int doCmd##name(void);
COMMANDS(EXTERN_FUNC)
#undef EXTERN_FUNC

/* declare the function pointer type and the jump table  */
typedef int (*CommandFunc)(void);
extern CommandFunc commandJumpTable[];
```

Все перечисленные ниже могут быть в разных единицах компиляции, если указанная выше часть включена в заголовок:

```
/* generate the jump table */
#define FUNC_NAME(name) doCmd##name,
CommandFunc commandJumpTable[] = {
  COMMANDS(FUNC_NAME)
};
#undef FUNC_NAME

/* call the save command like this: */
int result = commandJumpTable[cmdSave]();

/* somewhere else, we need the implementations of the commands */
int doCmdOpen(void) {/* code performing open command */}
int doCmdClose(void) {/* code performing close command */}
int doCmdSave(void) {/* code performing save command */}
int doCmdQuit(void) {/* code performing quit command */}
```

Примером этого метода, используемого в реальном коде, является диспетчеризация графических процессоров в Chromium .

Прочитайте X-макросы онлайн: https://riptutorial.com/ru/c/topic/628/x-макросы

глава 7: Аргументы командной строки

Синтаксис

- int main (int argc, char * argv [])

параметры

параметр	подробности
ARGC	аргумент count - инициализируется количеством аргументов, разделенных пробелами, заданных программе из командной строки, а также самого имени программы.
ARGV	вектор-аргумент - инициализируется массивом `char` указателей (строк), содержащих аргументы (и имя программы), которые были указаны в командной строке.

замечания

Программа AC, работающая в «размещенной среде» (нормальный тип - в отличие от «автономной среды») должна иметь `main` функцию. Это традиционно определяется как:

```
int main(int argc, char *argv[])
```

Обратите внимание, что `argv` также может быть и очень часто определяется как `char **argv` ; поведение такое же. Кроме того, имена параметров могут быть изменены, поскольку они являются только локальными переменными внутри функции, но `argc` и `argv` являются обычными, и вы должны использовать эти имена.

Для `main` функций, где код не использует никаких аргументов, используйте `int main(void)` .

Оба параметра инициализируются при запуске программы:

- `argc` инициализируется количеством аргументов, разделенных пробелом, заданных программе из командной строки, а также самого имени программы.
- `argv` - это массив `char` указателей (строк), содержащих аргументы (и имя программы), которые были указаны в командной строке.
- некоторые системы расширяют аргументы командной строки «в оболочке», другие - нет. В Unix, если пользователь вводит `myprogram *.txt` программа получит список текстовых файлов; в Windows он получит строку « `*.txt` ».

Примечание. Перед использованием `argv` вам может потребоваться проверить значение `argc` . Теоретически `argc` может быть `0` , а если `argc` равно нулю, то аргументов нет, а `argv[0]` (эквивалентный `argv[argc]`) является нулевым указателем. Если бы вы столкнулись с этой проблемой, это была бы необычная система с размещенной средой. Точно так же возможно, хотя и очень необычно, чтобы не было никакой информации о названии программы. В этом случае `argv[0][0] == '\0'` - имя программы может быть пустым.

Предположим, что мы запускаем программу следующим образом:

```
./some_program abba banana mamajam
```

Тогда `argc` равен `4` , а аргументы командной строки:

- `argv[0]` указывает на `"./some_program"` (имя программы), если имя программы доступно из среды хоста. Иначе пустая строка `""` .
- `argv[1]` указывает на `"abba"` ,
- `argv[2]` указывает на `"banana"` ,
- `argv[3]` указывает на `"mamajam"` ,
- `argv[4]` содержит значение `NULL` .

См. Также Что должно `main()` возвращать в C и C ++ для полных кавычек из стандарта.

Examples

Печать аргументов командной строки

После получения аргументов вы можете распечатать их следующим образом:

```
int main(int argc, char **argv)
{
    for (int i = 1; i < argc; i++)
    {
        printf("Argument %d: [%s]\n", i, argv[i]);
    }
}
```

Заметки

1. Параметр `argv` может быть определен как `char *argv[]` .
2. `argv[0]` *может* содержать имя самой программы (в зависимости от того, как была выполнена программа). Первый «реальный» аргумент командной строки находится в `argv[1]` , и именно поэтому переменная цикла `i` инициализируется 1.
3. В заявлении print вы можете использовать `*(argv + i)` вместо `argv[i]` - он оценивает одно и то же, но более подробен.
4. Квадратные скобки вокруг значения аргумента помогают идентифицировать начало и конец. Это может быть неоценимо, если в аргументе есть завершающие пробелы,

символы новой строки, возврат каретки или другие нечетные символы. Некоторые варианты этой программы являются полезным инструментом для отладки сценариев оболочки, где вам нужно понять, что на самом деле содержит список аргументов (хотя есть простые альтернативы оболочки, которые почти эквивалентны).

Печать аргументов в программу и преобразование в целые значения

Следующий код будет печатать аргументы в программе, и код попытается преобразовать каждый аргумент в число (до `long`):

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <limits.h>

int main(int argc, char* argv[]) {

    for (int i = 1; i < argc; i++) {
        printf("Argument %d is: %s\n", i, argv[i]);

        errno = 0;
        char *p;
        long argument_numValue = strtol(argv[i], &p, 10);

        if (p == argv[i]) {
            fprintf(stderr, "Argument %d is not a number.\n", i);
        }
        else if ((argument_numValue == LONG_MIN || argument_numValue == LONG_MAX) && errno ==
ERANGE) {
            fprintf(stderr, "Argument %d is out of range.\n", i);
        }
        else {
            printf("Argument %d is a number, and the value is: %ld\n",
                   i, argument_numValue);
        }
    }
    return 0;
}
```

РЕКОМЕНДАЦИИ:

- strtol () возвращает неверное значение
- Правильное использование strtol

Использование инструментов GNU getopt

Параметры командной строки для приложений не обрабатываются иначе, чем аргументы командной строки на языке C. Это просто аргументы, которые в среде Linux или Unix традиционно начинаются с тире (`-`).

С glibc в среде Linux или Unix вы можете использовать инструменты getopt, чтобы легко определять, проверять и анализировать параметры командной строки из остальных ваших

аргументов.

Эти инструменты ожидают, что ваши параметры будут отформатированы в соответствии со стандартами кодирования GNU , что является расширением того, что POSIX задает для формата параметров командной строки.

В приведенном ниже примере демонстрируется обработка параметров командной строки с помощью инструментов GNU getopt.

```
#include <stdio.h>
#include <getopt.h>
#include <string.h>

/* print a description of all supported options */
void usage (FILE *fp, const char *path)
{
    /* take only the last portion of the path */
    const char *basename = strrchr(path, '/');
    basename = basename ? basename + 1 : path;

    fprintf (fp, "usage: %s [OPTION]\n", basename);
    fprintf (fp, "  -h, --help\t\t"
                 "Print this help and exit.\n");
    fprintf (fp, "  -f, --file[=FILENAME]\t"
                 "Write all output to a file (defaults to out.txt).\n");
    fprintf (fp, "  -m, --msg=STRING\t"
                 "Output a particular message rather than 'Hello world'.\n");
}

/* parse command-line options and print message */
int main(int argc, char *argv[])
{
    /* for code brevity this example just uses fixed buffer sizes for strings */
    char filename[256] = { 0 };
    char message[256] = "Hello world";
    FILE *fp;
    int help_flag = 0;
    int opt;

    /* table of all supported options in their long form.
     * fields: name, has_arg, flag, val
     * `has_arg` specifies whether the associated long-form option can (or, in
     * some cases, must) have an argument. the valid values for `has_arg` are
     * `no_argument`, `optional_argument`, and `required_argument`.
     * if `flag` points to a variable, then the variable will be given a value
     * of `val` when the associated long-form option is present at the command
     * line.
     * if `flag` is NULL, then `val` is returned by `getopt_long` (see below)
     * when the associated long-form option is found amongst the command-line
     * arguments.
     */
    struct option longopts[] = {
        { "help", no_argument, &help_flag, 1 },
        { "file", optional_argument, NULL, 'f' },
        { "msg", required_argument, NULL, 'm' },
        { 0 }
    };
```

```
/* infinite loop, to be broken when we are done parsing options */
while (1) {
    /* getopt_long supports GNU-style full-word "long" options in addition
     * to the single-character "short" options which are supported by
     * getopt.
     * the third argument is a collection of supported short-form options.
     * these do not necessarily have to correlate to the long-form options.
     * one colon after an option indicates that it has an argument, two
     * indicates that the argument is optional. order is unimportant.
     */
    opt = getopt_long (argc, argv, "hf::m:", longopts, 0);

    if (opt == -1) {
        /* a return value of -1 indicates that there are no more options */
        break;
    }

    switch (opt) {
    case 'h':
        /* the help_flag and value are specified in the longopts table,
         * which means that when the --help option is specified (in its long
         * form), the help_flag variable will be automatically set.
         * however, the parser for short-form options does not support the
         * automatic setting of flags, so we still need this code to set the
         * help_flag manually when the -h option is specified.
         */
        help_flag = 1;
        break;
    case 'f':
        /* optarg is a global variable in getopt.h. it contains the argument
         * for this option. it is null if there was no argument.
         */
        printf ("outarg: '%s'\n", optarg);
        strncpy (filename, optarg ? optarg : "out.txt", sizeof (filename));
        /* strncpy does not fully guarantee null-termination */
        filename[sizeof (filename) - 1] = '\0';
        break;
    case 'm':
        /* since the argument for this option is required, getopt guarantees
         * that aptarg is non-null.
         */
        strncpy (message, optarg, sizeof (message));
        message[sizeof (message) - 1] = '\0';
        break;
    case '?':
        /* a return value of '?' indicates that an option was malformed.
         * this could mean that an unrecognized option was given, or that an
         * option which requires an argument did not include an argument.
         */
        usage (stderr, argv[0]);
        return 1;
    default:
        break;
    }
}

if (help_flag) {
    usage (stdout, argv[0]);
    return 0;
}
```

```
    if (filename[0]) {
        fp = fopen (filename, "w");
    } else {
        fp = stdout;
    }

    if (!fp) {
        fprintf(stderr, "Failed to open file.\n");
        return 1;
    }

    fprintf (fp, "%s\n", message);

    fclose (fp);
    return 0;
}
```

Он может быть скомпилирован с помощью `gcc` :

```
gcc example.c -o example
```

Он поддерживает три параметра командной строки (`--help` , `--file` и `--msg`). У всех также есть «короткая форма» (`-h` , `-f` и `-m`). Опции «file» и «msg» принимают аргументы. Если вы укажете опцию «msg», ее аргумент необходим.

Аргументы для параметров форматируются как:

- `--option=value` (для вариантов длинной формы)
- `-ovalue` или `-o"value"` (для вариантов короткой формы)

Прочитайте Аргументы командной строки онлайн: https://riptutorial.com/ru/c/topic/1285/аргументы-командной-строки

глава 8: атомная энергетика

Синтаксис

- `#ifdef __STDC_NO_ATOMICS__`
- `# error this implementation needs atomics`
- `#endif`
- `#include <stdatomic.h>`
- unsigned _Atomic counter = ATOMIC_VAR_INIT (0);

замечания

Atomics как часть языка C является дополнительной функцией, доступной с C11.

Их цель - обеспечить беспроблемный доступ к переменным, которые разделяются между различными потоками. Без атомной квалификации состояние общей переменной не будет определено, если два потока обращаются к нему одновременно. Например, операцию приращения (`++`) можно разделить на несколько команд ассемблера, чтение, само добавление и инструкцию хранилища. Если другой поток будет выполнять одну и ту же операцию, их две последовательности команд могут быть переплетены и привести к несогласованному результату.

- **Типы.** Все типы объектов, за исключением типов массивов, могут быть квалифицированы с помощью `_Atomic` .
- **Операторы:** все операции чтения-изменения-записи (например, `++` или `*=`) на них гарантированно являются атомарными.
- **Операции:** Существуют и другие операции, которые задаются как общие функции типа, например, `atomic_compare_exchange` .
- **Темы:** Доступ к ним гарантированно не приведет к сбору данных, когда к ним обращаются разные потоки.
- **Обработчики сигналов:** Атомные типы называются *блокируемыми,* если все операции с ними не имеют состояния. В этом случае они также могут использоваться для обработки изменений состояния между нормальным потоком управления и обработчиком сигналов.
- Существует только один тип данных, который гарантированно блокируется: `atomic_flag` . Это минимальный тип операций, которые предназначены для сопоставления с эффективными тестовыми и установленными аппаратными инструкциями.

Другие способы избежать условий гонки доступны в интерфейсе потоков C11, в частности, mutex-тип `mtx_t` чтобы взаимно исключить потоки от доступа к критическим данным или критическим разделам кода. Если атомы недоступны, они должны использоваться для предотвращения рас.

Examples

атомы и операторы

Доступ к атомным переменным возможен одновременно между различными потоками без создания условий гонки.

```
/* a global static variable that is visible by all threads */
static unsigned _Atomic active = ATOMIC_VAR_INIT(0);


int myThread(void* a) {
  ++active;         // increment active race free
  // do something
  --active;         // decrement active race free
  return 0;
}
```

Все операции lvalue (операции, которые изменяют объект), разрешенные для базового типа, разрешены и не приводят к условиям гонки между различными потоками, которые обращаются к ним.

- Операции над атомными объектами обычно на порядок медленнее, чем обычные арифметические операции. Это также включает в себя простые операции загрузки или хранения. Поэтому вы должны использовать их только для критических задач.
- Обычные арифметические операции и присвоение, такие как `a = a+1;` фактически три операции на : первые нагрузки, а затем сложение и , наконец, магазин. `a` Это *не* гонка бесплатно. Только операция `a += 1;` и `a++;` являются.

Прочитайте атомная энергетика онлайн: https://riptutorial.com/ru/c/topic/4924/атомная-энергетика

глава 9: Битовые поля

Вступление

Большинство переменных в C имеют размер, который является целым числом байтов. Бит-поля являются частью структуры, которая не обязательно занимает целое число байтов; они могут принимать любое количество бит. Несколько бит-полей могут быть упакованы в единый блок хранения. Они являются частью стандарта C, но есть много аспектов, которые определяются реализацией. Они являются одной из наименее переносимых частей C.

Синтаксис

- Идентификатор спецификатора типа: размер;

параметры

параметр	Описание
Тип Спецификатор	`signed` , `unsigned` , `int` или `_Bool`
идентификатор	Имя этого поля в структуре
размер	Количество бит для использования в этом поле

замечания

Единственные переносные типы для бит-полей - это `signed` , `unsigned` или `_Bool` . Можно использовать простой тип `int` , но стандарт говорит (§6.7.2¶5) *... для бит-полей, он определяется реализацией, специфицирует ли спецификатор* `int` *тот же тип, что и* `signed int` *или тот же тип, что и* `unsigned int` *.*

Другие целые типы могут быть разрешены конкретной реализацией, но использование их не переносимо.

Examples

Битовые поля

Простое битовое поле может использоваться для описания вещей, которые могут иметь определенное количество бит.

```
struct encoderPosition {
   unsigned int encoderCounts : 23;
   unsigned int encoderTurns  : 4;
   unsigned int _reserved     : 5;
};
```

В этом примере мы рассмотрим кодировщик с 23 битами одинарной точности и 4 битами для описания многооборотных. Бит-поля часто используются при взаимодействии с оборудованием, которое выводит данные, связанные с определенным количеством бит. Другим примером может быть связь с FPGA, где FPGA записывает данные в вашу память в 32-разрядных разделах, что позволяет использовать аппаратное обеспечение:

```
struct FPGAInfo {
    union {
        struct bits {
            unsigned int bulb1On  : 1;
            unsigned int bulb2On  : 1;
            unsigned int bulb1Off : 1;
            unsigned int bulb2Off : 1;
            unsigned int jetOn    : 1;
        };
        unsigned int data;
   };
};
```

В этом примере мы показали широко используемую конструкцию, чтобы иметь возможность доступа к данным в своих отдельных битах или для записи пакета данных в целом (эмулируя то, что может сделать FPGA). Затем мы могли бы получить доступ к битам следующим образом:

```
FPGAInfo fInfo;
fInfo.data = 0xFF34F;
if (fInfo.bits.bulb1On) {
    printf("Bulb 1 is on\n");
}
```

Это действительно, но согласно стандарту C99 6.7.2.1, пункт 10:

> Порядок распределения бит-полей внутри единицы (от высокого порядка до младшего или низкого порядка) определяется реализацией.

Таким образом, при определении бит-полей вы должны знать об истинности. Таким образом, может потребоваться использовать директиву препроцессора для проверки подлинности машины. Пример этого следует:

```
typedef union {
    struct bits {
#if defined(WIN32) || defined(LITTLE_ENDIAN)
    uint8_t commFailure :1;
    uint8_t hardwareFailure :1;
    uint8_t _reserved :6;
#else
```

```
    uint8_t _reserved :6;
    uint8_t hardwareFailure :1;
    uint8_t commFailure :1;
#endif
    };
    uint8_t data;
} hardwareStatus;
```

Использование битовых полей в виде небольших целых чисел

```
#include <stdio.h>

int main(void)
{
    /* define a small bit-field that can hold values from 0 .. 7 */
    struct
    {
        unsigned int uint3: 3;
    } small;

    /* extract the right 3 bits from a value */
    unsigned int value = 255 - 2; /* Binary 11111101 */
    small.uint3 = value;          /* Binary      101 */
    printf("%d", small.uint3);

    /* This is in effect an infinite loop */
    for (small.uint3 = 0; small.uint3 < 8; small.uint3++)
    {
        printf("%d\n", small.uint3);
    }

    return 0;
}
```

Выравнивание битового поля

Бит-поля дают возможность объявлять поля структуры, которые меньше ширины символов. Бит-поля реализуются с помощью маски уровня байта или уровня слова. Следующий пример приводит к структуре из 8 байтов.

```
struct C
{
    short s;            /* 2 bytes */
    char  c;            /* 1 byte */
    int   bit1 : 1;     /* 1 bit */
    int   nib  : 4;     /* 4 bits padded up to boundary of 8 bits. Thus 3 bits are padded */
    int   sept : 7;     /* 7 Bits septet, padded up to boundary of 32 bits. */
};
```

Комментарии описывают один возможный макет, но поскольку стандарт говорит, *что выравнивание адресного блока хранения не указано* , возможны и другие макеты.

Безымянное битовое поле может иметь любой размер, но их нельзя инициализировать или ссылаться.

Битовому полю с нулевой шириной нельзя присвоить имя и выровнять следующее поле с границей, определяемой типом данных бит-поля. Это достигается путем заполнения битов между битовыми полями.

Размер структуры «A» равен 1 байт.

```
struct A
{
    unsigned char c1 : 3;
    unsigned char c2 : 4;
    unsigned char c3 : 1;
};
```

В структуре B первое неназванное битовое поле пропускает 2 бита; бит-поле нулевой ширины после `c2` заставляет `c3` начинаться с границы символа (так что 3 бита пропускаются между `c2` и `c3` . После `c4` есть 3 бита заполнения. Таким образом, размер структуры составляет 2 байта.

```
struct B
{
    unsigned char c1 : 1;
    unsigned char    : 2;    /* Skips 2 bits in the layout */
    unsigned char c2 : 2;
    unsigned char    : 0;    /* Causes padding up to next container boundary */
    unsigned char c3 : 4;
    unsigned char c4 : 1;
};
```

Когда поле бит полезно?

Битовое поле используется для объединения многих переменных в один объект, аналогичный структуре. Это позволяет сократить использование памяти и особенно полезно во встроенной среде.

```
 e.g. consider the following variables having the ranges as given below.
 a --> range 0 - 3
 b --> range 0 - 1
 c --> range 0 - 7
 d --> range 0 - 1
 e --> range 0 - 1
```

Если мы объявляем эти переменные отдельно, каждый из них должен быть как минимум 8-битным целым числом, а требуемое общее пространство будет 5 байтов. Кроме того, переменные не будут использовать весь диапазон 8-битного беззнакового целого числа (0-255). Здесь мы можем использовать бит-поля.

```
typedef struct {
   unsigned int a:2;
   unsigned int b:1;
   unsigned int c:3;
```

```
   unsigned int d:1;
   unsigned int e:1;
} bit_a;
```

Битовые поля в структуре доступны так же, как и любая другая структура. Программисту необходимо следить за тем, чтобы переменные записывались в диапазоне. Если вне диапазона, поведение не определено.

```
int main(void)
{
   bit_a bita_var;
   bita_var.a = 2;              // to write into element a
   printf ("%d",bita_var.a);    // to read from element a.
   return 0;
}
```

Часто программист хочет обнулить набор бит-полей. Это можно сделать по элементам, но есть второй метод. Просто создайте объединение структуры выше с неподписанным типом, который больше или равен размеру структуры. Тогда весь набор бит-полей может быть обнулен путем обнуления этого целого без знака.

```
typedef union {
    struct {
       unsigned int a:2;
       unsigned int b:1;
       unsigned int c:3;
       unsigned int d:1;
       unsigned int e:1;
    };
    uint8_t data;
} union_bit;
```

Использование выглядит следующим образом

```
int main(void)
{
   union_bit un_bit;
   un_bit.data = 0x00;         // clear the whole bit-field
   un_bit.a = 2;               // write into element a
   printf ("%d",un_bit.a);     // read from element a.
   return 0;
}
```

В заключение, бит-поля обычно используются в ситуациях с ограниченной памятью, где у вас много переменных, которые могут принимать ограниченные диапазоны.

Не требуется для бит-полей

1. Массивы бит-полей, указатели на бит-поля и функции, возвращающие бит-поля, недопустимы.
2. Оператор адреса (&) не может применяться к элементам битового поля.

3. Тип данных битового поля должен быть достаточно широким, чтобы содержать размер поля.
4. Оператор `sizeof()` не может быть применен к битовому полю.
5. Невозможно создать `typedef` для отдельного битового поля (хотя вы, безусловно, можете создать `typedef` для структуры, содержащей бит-поля).

```
typedef struct mybitfield
{
    unsigned char c1 : 20;   /* incorrect, see point 3 */
    unsigned char c2 : 4;    /* correct */
    unsigned char c3 : 1;
    unsigned int x[10]: 5;   /* incorrect, see point 1 */
} A;

int SomeFunction(void)
{
    // Somewhere in the code
    A a = { … };
    printf("Address of a.c2 is %p\n", &a.c2);      /* incorrect, see point 2 */
    printf("Size of a.c2 is %zu\n", sizeof(a.c2)); /* incorrect, see point 4 */
}
```

Прочитайте Битовые поля онлайн: https://riptutorial.com/ru/c/topic/1930/битовые-поля

глава 10: Встраивание

Examples

Функции вложения, используемые в более чем одном исходном файле

Для небольших функций, вызываемых часто, служебные данные, связанные с вызовом функции, могут быть значительной частью общего времени выполнения этой функции. Таким образом, одним из способов повышения производительности является устранение накладных расходов.

В этом примере мы используем четыре функции (плюс `main()`) в трех исходных файлах. Два из них (`plusfive()` и `timestwo()`) каждый вызываются двумя другими, расположенными в «source1.c» и «source2.c». `main()` включен, поэтому у нас есть рабочий пример.

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int main(void) {
    int start = 3;
    int intermediate = complicated1(start);
    printf("First result is %d\n", intermediate);
    intermediate = complicated2(start);
    printf("Second result is %d\n", intermediate);
    return 0;
}
```

source1.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated1(int input) {
    int tmp = timestwo(input);
    tmp = plusfive(tmp);
    return tmp;
}
```

source2.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"
```

```
int complicated2(int input) {
    int tmp = plusfive(input);
    tmp = timestwo(tmp);
    return tmp;
}
```

headerfile.h:

```
#ifndef HEADERFILE_H
#define HEADERFILE_H

int complicated1(int input);
int complicated2(int input);

inline int timestwo(int input) {
  return input * 2;
}
inline int plusfive(int input) {
  return input + 5;
}

#endif
```

Функции `timestwo` и `plusfive` вызываются как `complicated1` и `complicated2` , которые находятся в разных «единицах перевода» или исходных файлах. Чтобы использовать их таким образом, мы должны определить их в заголовке.

Скомпилируйте это, предположив gcc:

```
cc -O2 -std=c99 -c -o main.o main.c
cc -O2 -std=c99 -c -o source1.o source1.c
cc -O2 -std=c99 -c -o source2.o source2.c
cc main.o source1.o source2.o -o main
```

Мы используем опцию оптимизации -O2, потому что некоторые компиляторы не встроены без оптимизации.

Эффект ключевого слова `inline` заключается в том, что соответствующий символ функции не попадает в объектный файл. В противном случае в последней строке будет возникать ошибка, когда мы связываем объектные файлы с окончательным исполняемым файлом. Если бы у нас не было `inline` , один и тот же символ был бы определен в обоих файлах `.o` , и возникла бы ошибка «с несколькими символами».

В ситуациях, когда символ действительно необходим, это имеет тот недостаток, что символ вообще не создается. Есть две возможности справиться с этим. Первое заключается в добавлении дополнительного выражения `extern` для встроенных функций в один из файлов `.c` . Поэтому добавьте следующее в `source1.c` :

```
extern int timestwo(int input);
extern int plusfive(int input);
```

Другая возможность - определить функцию со `static inline` а не `inline` . Этот метод имеет недостаток, что в конечном итоге копия рассматриваемой функции может быть создана в **каждом** объектном файле, который создается с этим заголовком.

Прочитайте Встраивание онлайн: https://riptutorial.com/ru/c/topic/7427/встраивание

глава 11: Встроенная сборка

замечания

Встроенная сборка - это практика добавления инструкций по сборке в середине исходного кода C. Стандарт ISO C не требует поддержки встроенной сборки. Поскольку это не требуется, синтаксис для встроенной сборки зависит от компилятора и компилятора. Несмотря на то, что он обычно поддерживается, существует очень мало причин использовать встроенную сборку и многие причины этого не делать.

Pros

1. **Производительность.** Написав конкретные инструкции по сборке для операции, вы можете добиться большей производительности, чем код сборки, сгенерированный компилятором. Обратите внимание, что эти приросты производительности редки. В большинстве случаев вы можете добиться повышения производительности, просто переставив свой код C, чтобы оптимизатор мог выполнять свою работу.
2. **Аппаратный интерфейс** Драйвер устройства или код запуска процессора может потребоваться некоторый код сборки для доступа к правильным регистрам и гарантировать, что определенные операции выполняются в определенном порядке с определенной задержкой между операциями.

Cons

1. Синтаксис **компилятора** для встроенной сборки не гарантируется одинаковым от одного компилятора к другому. Если вы пишете код с встроенной сборкой, которая должна поддерживаться разными компиляторами, используйте макросы препроцессора (`#ifdef`), чтобы проверить, какой компилятор используется. Затем напишите отдельный сборный раздел для каждого поддерживаемого компилятора.
2. **Переносимость процессора.** Вы не можете писать встроенную сборку для процессора x86 и ожидать, что она будет работать на процессоре ARM. Встроенная сборка предназначена для записи для конкретного семейства процессоров или процессоров. Если у вас встроенная сборка, которую вы хотите поддерживать на разных процессорах, используйте макросы препроцессора, чтобы проверить, на какой процессор компилируется код, и выбрать соответствующий раздел кода сборки.
3. **Будущие изменения производительности** Встраиваемая сборка может быть написана в ожидании задержек на основе определенной тактовой частоты процессора. Если программа скомпилирована для процессора с более быстрыми часами, код сборки может работать не так, как ожидалось.

Examples

Поддержка gcc Basic asm

Базовая поддержка сборки с gcc имеет следующий синтаксис:

```
asm [ volatile ] ( AssemblerInstructions )
```

где `AssemblerInstructions` - это код прямой сборки для данного процессора. Ключевое слово volatile является необязательным и не имеет никакого эффекта, поскольку gcc не оптимизирует код в базовом выражении asm. `AssemblerInstructions` может содержать несколько инструкций по сборке. Базовый оператор asm используется, если у вас есть asm-процедура, которая должна существовать вне функции C. Следующий пример приведен в руководстве GCC:

```
/* Note that this code will not compile with -masm=intel */
#define DebugBreak() asm("int $3")
```

В этом примере вы могли бы использовать `DebugBreak()` в других местах вашего кода, и он выполнит команду сборки `int $3` . Обратите внимание, что даже если gcc не будет изменять какой-либо код в базовом выражении asm, оптимизатор может по-прежнему перемещать последовательные операторы asm. Если у вас есть несколько инструкций по сборке, которые должны выполняться в определенном порядке, включите их в один оператор asm.

Поддержка gcc Extended asm

Расширенная поддержка asm в gcc имеет следующий синтаксис:

```
asm [volatile] ( AssemblerTemplate
                  : OutputOperands
                  [ : InputOperands
                  [ : Clobbers ] ])

 asm [volatile] goto ( AssemblerTemplate
                       :
                       : InputOperands
                       : Clobbers
                       : GotoLabels)
```

где `AssemblerTemplate` является шаблоном для команды ассемблера, `OutputOperands` - это любые переменные C, которые могут быть изменены кодом сборки, `InputOperands` - это любые переменные C, используемые в качестве входных параметров, `Clobbers` - это список или регистры, которые изменены кодом сборки, и `GotoLabels` - любые метки операторов goto, которые могут использоваться в коде сборки.

Расширенный формат используется в функциях C и является более типичным использованием встроенной сборки. Ниже приведен пример ядра Linux для байтового обмена 16-битными и 32-разрядными номерами для ARM-процессора:

```
/* From arch/arm/include/asm/swab.h in Linux kernel version 4.6.4 */
#if __LINUX_ARM_ARCH__ >= 6

static inline __attribute_const__ __u32 __arch_swahb32(__u32 x)
{
    __asm__ ("rev16 %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swahb32 __arch_swahb32
#define __arch_swab16(x) ((__u16)__arch_swahb32(x))

static inline __attribute_const__ __u32 __arch_swab32(__u32 x)
{
    __asm__ ("rev %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swab32 __arch_swab32

#endif
```

Каждая секция asm использует переменную `x` качестве ее входного и выходного параметров. Затем функция C возвращает обработанный результат.

В расширенном формате asm gcc может оптимизировать инструкции сборки в блоке asm, следуя тем же правилам, которые он использует для оптимизации кода C. Если вы хотите, чтобы ваш раздел asm остался нетронутым, используйте ключевое слово `volatile` для раздела asm.

gcc Встроенная сборка в макросах

Мы можем поместить инструкции сборки внутри макроса и использовать макрос, как вы бы назвали функцию.

```
#define mov(x,y) \
{ \
    __asm__ ("l.cmov %0,%1,%2" : "=r" (x) : "r" (y), "r" (0x0000000F)); \
}

/// some definition and assignment
unsigned char sbox[size][size];
unsigned char sbox[size][size];

///Using
mov(state[0][1], sbox[si][sj]);
```

Использование встроенных команд сборки, встроенных в код C, может улучшить время работы программы. Это очень полезно в критичных по времени ситуациях, таких как криптографические алгоритмы, такие как AES. Например, для простой операции сдвига, которая необходима в алгоритме AES, мы можем заменить прямую инструкцию по `Rotate Right` с помощью оператора сдвига C `>>` .

В реализации «AES256» в функции «AddRoundKey ()» мы имеем следующие утверждения:

```
unsigned int w;           // 32-bit
unsigned char subkey[4]; // 8-bit, 4*8 = 32

subkey[0] = w >> 24;      // hold 8 bit, MSB, leftmost group of 8-bits
subkey[1] = w >> 16;      // hold 8 bit, second group of 8-bit from left
subkey[2] = w >> 8;       // hold 8 bit, second group of 8-bit from right
subkey[3] = w;            // hold 8 bit, LSB, rightmost group of 8-bits

/// subkey <- w
```

Они просто присваивают значение бит `w` массиву `subkey` .

Мы можем изменить три смещения + присваивать и назначить выражение C только с одной командой « `Rotate Right` .

```
__asm__ ("l.ror  %0,%1,%2" : "=r" (* (unsigned int *) subkey)  : "r" (w), "r" (0x10));
```

Конечный результат точно такой же.

Прочитайте Встроенная сборка онлайн: https://riptutorial.com/ru/c/topic/4263/встроенная-сборка

глава 12: Выборочные заявления

Examples

if () Заявления

Один из самых простых способов контролировать ход выполнения программы заключается в использовании , `if` заявления выбора. Независимо от того, должен ли исполняться блок кода или не быть выполнен, это утверждение можно решить.

Синтаксис `if` оператор выбора в C может быть следующим:

```
if(cond)
{
  statement(s);  /*to be executed, on condition being true*/
}
```

Например,

```
if (a > 1) {
    puts("a is larger than 1");
}
```

Где `a > 1` - это *условие* , которое должно оцениваться в `true` для выполнения операторов внутри блока `if` . В этом примере «a больше 1» печатается только, если `a > 1` истинно.

`if` операторы выбора могут опускать скобки для переноски `{` и `}` если в блоке есть только один оператор. Вышеприведенный пример можно переписать в

```
if (a > 1)
    puts("a is larger than 1");
```

Однако для выполнения нескольких операторов в блоке необходимо использовать фигурные скобки.

Условие `if` может включать несколько выражений. `if` будет выполняться только действие, если конечный результат выражения равен true.

Например

```
if ((a > 1) && (b > 1)) {
    puts("a is larger than 1");
    a++;
}
```

будет выполнять только `printf` и `a++` если **и** `a` и `b` больше `1` .

if () ... else и синтаксис

Если `if` выполняет действие только тогда, когда его условие оценивается как `true` , `if` / `else` позволяет вам указать разные действия, когда условие `true` и когда условие `false` .

Пример:

```
if (a > 1)
    puts("a is larger than 1");
else
    puts("a is not larger than 1");
```

Так же, как и оператор `if` , когда блок внутри `if` или `else` состоит только из одного оператора, то фигурные скобки могут быть опущены (но делать это не рекомендуется, так как это может легко ввести проблемы непроизвольно). Однако, если в блоке `if` или `else` имеется более одного оператора, то на этом конкретном блоке должны использоваться фигурные скобки.

```
if (a > 1)
{
    puts("a is larger than 1");
    a--;
}
else
{
    puts("a is not larger than 1");
    a++;
}
```

switch () Заявления

Операторы `switch` полезны, когда вы хотите, чтобы ваша программа делала много разных вещей в соответствии со значением конкретной тестовой переменной.

Пример использования оператора `switch` выглядит следующим образом:

```
int a = 1;

switch (a) {
case 1:
    puts("a is 1");
    break;
case 2:
    puts("a is 2");
    break;
default:
    puts("a is neither 1 nor 2");
    break;
}
```

Этот пример эквивалентен

```
int a = 1;

if (a == 1) {
    puts("a is 1");
} else if (a == 2) {
    puts("a is 2");
} else {
    puts("a is neither 1 nor 2");
}
```

Если значение равно 1 , когда `a switch` используется оператор, `a is 1` будет печататься. Если значение `a` равно 2, тогда будет напечатан `a is 2` . В противном случае `a is neither 1 nor 2` будет напечатано `a is neither 1 nor 2` .

`case n:` используется для описания того, где будет выполняться поток выполнения, когда значение, переданное в оператор `switch` равно *n* . *n* должна быть константой времени компиляции, и одно и то же *n* может существовать не более одного раза в одном операторе `switch` .

`default:` используется для описания того, когда значение не соответствует ни одному из вариантов для `case n:` Хорошей практикой является включение случая по `default` в каждый оператор switch для обнаружения неожиданного поведения.

`break;` оператор должен выпрыгнуть из блока `switch` .

Примечание. Если вы случайно забыли добавить `break` после окончания `case` , компилятор предположит, что вы намереваетесь «провалиться», и все последующие операторы case, если таковые имеются, будут выполнены (если оператор *break* не найден в любой из последующих случаев), независимо от того, соответствуют ли последующие утверждения (-ы) случая. Это конкретное свойство используется для реализации устройства Даффа . Такое поведение часто считается недостатком в спецификации языка C.

Ниже приведен пример, показывающий эффекты отсутствия `break;` :

```
int a = 1;

switch (a) {
case 1:
case 2:
    puts("a is 1 or 2");
case 3:
    puts("a is 1, 2 or 3");
    break;
default:
    puts("a is neither 1, 2 nor 3");
    break;
}
```

Когда значение `a` равно 1 или 2, `a is 1 or 2` а `a is 1, 2 or 3` , оба будут напечатаны. Когда `a` равно 3, будет напечатано только `a is 1, 2 or 3` . В противном случае `a is neither 1, 2 nor 3` будет напечатано `a is neither 1, 2 nor 3` .

Обратите внимание, что случай по `default` не нужен, особенно когда набор значений, которые вы получаете в `switch` , завершен и известен во время компиляции.

Лучшим примером является использование `switch enum` .

```
enum msg_type { ACK, PING, ERROR };
void f(enum msg_type t)
{
  switch (t) {
  case ACK:
    // do nothing
    break;
  case PING:
    // do something
    break;
  case ERROR:
    // do something else
    break;
  }
}
```

Для этого есть несколько преимуществ:

- большинство компиляторов сообщают о предупреждении, если вы не обрабатываете значение (это не будет сообщаться, если присутствует случай по `default`)
- по той же причине, если вы добавите новое значение в `enum` , вы будете уведомлены обо всех местах, где вы забыли обработать новое значение (в случае по `default` вам нужно будет вручную изучить ваш код для поиска таких случаев)
- Читателю не нужно вычислять «то, что скрыто по `default:` есть ли другие значения `enum` или это защита «на всякий случай». И если есть другие значения `enum` , кодер намеренно использовал случай по `default` для них или есть ошибка, которая была введена, когда он добавил значение?
- обработка каждого значения `enum` делает код самоочевидным, поскольку вы не можете спрятаться за дикой картой, вы должны явно обращаться с каждым из них.

Тем не менее, вы не можете запретить кому-то писать злой код, например:

```
enum msg_type t = (enum msg_type)666; // I'm evil
```

Таким образом, вы можете добавить дополнительную проверку перед своим коммутатором, чтобы обнаружить ее, если она вам действительно нужна.

```
void f(enum msg_type t)
{
   if (!is_msg_type_valid(t)) {
      // Handle this unlikely error
   }

   switch(t) {
    // Same code than before
   }
```

```
}
```

if () ... else Лестница Цепочка двух или более операторов if () ... else

Хотя оператор `if ()... else` позволяет определить только одно (по умолчанию) поведение, которое возникает, когда условие внутри `if ()` не выполняется, объединение двух или более операторов `if () ... else` позволяет определить пару больше поведения , прежде чем идти до последнего `else` филиала , действующего в качестве « по умолчанию», если таковые имеются.

Пример:

```
int a = ... /* initialise to some value. */

if (a >= 1)
{
    printf("a is greater than or equals 1.\n");
}
else if (a == 0) //we already know that a is smaller than 1
{
    printf("a equals 0.\n");
}
else /* a is smaller than 1 and not equals 0, hence: */
{
    printf("a is negative.\n");
}
```

Вложенное if () ... else VS if () .. else Лестница

Вложенные операторы `if()...else` принимают большее время выполнения (они медленнее) по сравнению с лестницей `if()...else` потому что вложенные операторы `if()...else` проверяют все внутренние условные операторы после внешнего условное выполнение `if()` выполняется, тогда как лестница `if()..else` прекратит тестирование условий, как только истинные условные утверждения `if()` или `else if()` верны.

Линия `if()...else` :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a, b, c;
  printf("\nEnter Three numbers = ");
  scanf("%d%d%d", &a, &b, &c);
  if ((a < b) && (a < c))
  {
    printf("\na = %d is the smallest.", a);
  }
  else if ((b < a) && (b < c))
  {
    printf("\nb = %d is the smallest.", b);
  }
```

```
  else if ((c < a) && (c < b))
  {
    printf("\nc = %d is the smallest.", c);
  }
  else
  {
    printf("\nImprove your coding logic");
  }
  return 0;
}
```

В общем случае считается лучшим, чем эквивалентное вложенное `if()...else` :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a, b, c;
  printf("\nEnter Three numbers = ");
  scanf("%d%d%d", &a, &b, &c);
  if (a < b)
  {
    if (a < c)
      {
        printf("\na = %d is the smallest.", a);
      }
    else
      {
        printf("\nc = %d is the smallest.", c);
      }
  }
  else
  {
    if(b < c)
    {
      printf("\nb = %d is the smallest.", b);
    }
    else
    {
      printf("\nc = %d is the smallest.", c);
    }
  }
  return 0;
}
```

Прочитайте Выборочные заявления онлайн: https://riptutorial.com/ru/c/topic/3073/выборочные-заявления

глава 13: Генерация случайных чисел

замечания

Из-за недостатков `rand()` многие годы появилось много других реализаций по умолчанию. Среди них:

- `arc4random()` (доступно для OS X и BSD)
- `random()` (доступно в Linux)
- `drand48()` (доступно на POSIX)

Examples

Генерация случайных чисел

Функция `rand()` может использоваться для генерации псевдослучайного целочисленного значения между `0` и `RAND_MAX` (`RAND_MAX` `0` и `RAND_MAX`).

`srand(int)` используется для подсчета генератора псевдослучайных чисел. Каждый раз, когда `rand()` засевается одним и тем же семенем, он должен вызывать одну и ту же последовательность значений. Он должен быть посеян только один раз перед вызовом `rand()` . Его не следует многократно посеять или пересаживать каждый раз, когда вы хотите создать новую партию псевдослучайных чисел.

Стандартная практика заключается в использовании результата `time(NULL)` в качестве семени. Если генератор случайных чисел требует детерминированной последовательности, вы можете засеять генератор с тем же значением при каждом запуске программы. Обычно это не требуется для кода выпуска, но полезно в отладочных запусках, чтобы сделать ошибки воспроизводимыми.

Рекомендуется всегда засевать генератор, если он не посеян, он ведет себя так, как если бы он был засеян с помощью `srand(1)` .

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    int i;
    srand(time(NULL));
    i = rand();

    printf("Random value between [0, %d]: %d\n", RAND_MAX, i);
    return 0;
}
```

Возможный выход:

```
Random value between [0, 2147483647]: 823321433
```

Заметки:

Стандарт C не гарантирует качество созданной случайной последовательности. Раньше некоторые реализации `rand()` имели серьезные проблемы в распределении и случайности генерируемых чисел. **Использование `rand()` не рекомендуется для серьезных потребностей генерации случайных чисел, таких как криптография.**

Преобразованный конгруэнтный генератор

Вот автономный генератор случайных чисел, который не полагается на функции `rand()` или аналогичные библиотеки.

Зачем вам это нужно? Возможно, вы не доверяете встроенному генератору случайных чисел вашей платформы, или, возможно, вам нужен воспроизводимый источник случайности, не зависящий от какой-либо конкретной реализации библиотеки.

Этот код является PCG32 от pcg-random.org , современного, быстрого, универсального RNG с отличными статистическими свойствами. Это не криптографически безопасно, поэтому не используйте его для криптографии.

```
#include <stdint.h>

/* *Really* minimal PCG32 code / (c) 2014 M.E. O'Neill / pcg-random.org
 * Licensed under Apache License 2.0 (NO WARRANTY, etc. see website) */

typedef struct { uint64_t state;  uint64_t inc; } pcg32_random_t;

uint32_t pcg32_random_r(pcg32_random_t* rng) {
    uint64_t oldstate = rng->state;
    /* Advance internal state */
    rng->state = oldstate * 6364136223846793005ULL + (rng->inc | 1);
    /* Calculate output function (XSH RR), uses old state for max ILP */
    uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
}

void pcg32_srandom_r(pcg32_random_t* rng, uint64_t initstate, uint64_t initseq) {
    rng->state = 0U;
    rng->inc = (initseq << 1u) | 1u;
    pcg32_random_r(rng);
    rng->state += initstate;
    pcg32_random_r(rng);
}
```

И вот как это назвать:

```
#include <stdio.h>
int main(void) {
    pcg32_random_t rng; /* RNG state */
    int i;

    /* Seed the RNG */
    pcg32_srandom_r(&rng, 42u, 54u);

    /* Print some random 32-bit integers */
    for (i = 0; i < 6; i++)
        printf("0x%08x\n", pcg32_random_r(&rng));

    return 0;
}
```

Ограничить генерацию до заданного диапазона

Обычно при генерации случайных чисел полезно генерировать целые числа в пределах диапазона или значение ap между 0.0 и 1.0. В то время как операция модуляции может использоваться для уменьшения количества семян до низкого целого, это использует низкие биты, которые часто проходят короткий цикл, что приводит к небольшому перекосу распределения, если N велико пропорционально RAND_MAX.

Макрос

```
#define uniform() (rand() / (RAND_MAX + 1.0))
```

дает значение ap от 0,0 до 1,0 - epsilon, поэтому

```
i = (int)(uniform() * N)
```

будет устанавливать `i` равномерное случайное число в диапазоне от 0 до N - 1.

К сожалению, существует технический недостаток, поскольку RAND_MAX разрешено быть большим, чем переменная типа `double` может точно представлять. Это означает, что `RAND_MAX + 1.0` оценивается как RAND_MAX, и функция иногда возвращает единицу. Однако это маловероятно.

Поколение Xorshift

Хорошей и простой альтернативой ошибочным процедурам `rand()` является *xorshift* , класс генераторов псевдослучайных чисел, открытых Джорджем Марсалья . Генератор xorshift является одним из самых быстрых некриптографически безопасных генераторов случайных чисел. Более подробная информация и другие примеры реализации доступны на странице Wikipedia xorshift

Пример реализации

```
#include <stdint.h>

/* These state variables must be initialised so that they are not all zero. */
uint32_t w, x, y, z;

uint32_t xorshift128(void)
{
    uint32_t t = x;
    t ^= t << 11U;
    t ^= t >> 8U;
    x = y; y = z; z = w;
    w ^= w >> 19U;
    w ^= t;
    return w;
}
```

Прочитайте Генерация случайных чисел онлайн: https://riptutorial.com/ru/c/topic/365/генерация-случайных-чисел

глава 14: Декларация против определения

замечания

Источник: Какая разница между определением и декларацией?

Источник (для слабых и сильных символов): https://www.amazon.com/Computer-Systems-Programmers-Perspective-2nd/dp/0136108040/

Examples

Понимание Декларации и Определение

Объявление вводит идентификатор и описывает его тип, будь то тип, объект или функция. Декларация - это то, что компилятор должен принимать ссылки на этот идентификатор. Это декларации:

```
extern int bar;
extern int g(int, int);
double f(int, double); /* extern can be omitted for function declarations */
double h1();           /* declaration without prototype */
double h2();           /* ditto                         */
```

Определение фактически создает / реализует этот идентификатор. Это то, что требуется компоновщику, чтобы связать ссылки на эти объекты. Это определения, соответствующие приведенным выше объявлениям:

```
int bar;
int g(int lhs, int rhs) {return lhs*rhs;}
double f(int i, double d) {return i+d;}
double h1(int a, int b) {return -1.5;}
double h2() {}  /* prototype is implied in definition, same as double h2(void) */
```

Определение может быть использовано в месте объявления.

Однако он должен быть определен ровно один раз. Если вы забыли определить что-то, что было объявлено и где-то указано, то компоновщик не знает, на что ссылаться ссылки и жалуется на недостающие символы. Если вы определяете что-то более одного раза, то компоновщик не знает, какое из определений ссылается на ссылки и жалуется на дублированные символы.

Исключение:

```
extern int i = 0;  /* defines i */
extern int j;  /* declares j */
```

Это исключение можно объяснить с помощью понятий «Сильные символы против слабых символов» (с точки зрения компоновщика). Подробнее см. Здесь (Слайд 22).

```
/* All are definitions. */
struct S { int a; int b; };             /* defines S */
struct X {                              /* defines X */
    int x;                              /* defines non-static data member x */
};
struct X anX;                                   /* defines anX */
```

Прочитайте Декларация против определения онлайн: https://riptutorial.com/ru/c/topic/3104/декларация-против-определения

глава 15: инициализация

Examples

Инициализация переменных в C

При отсутствии явной инициализации внешние и `static` переменные гарантированно инициализируются до нуля; автоматические переменные (включая `register` переменные) имеют *неопределенные* начальные значения [1] (т. е. мусор).

Скалярные переменные могут быть инициализированы, когда они определены, следуя имени с знаком равенства и выражением:

```
int x = 1;
char squota = '\'';
long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */
```

Для внешних и `static` переменных инициализатор должен быть *константным выражением* [2] ; инициализация выполняется один раз, концептуально до начала выполнения программы.

Для автоматических и `register` переменных инициализатор не ограничивается константой: это может быть любое выражение, включающее ранее определенные значения, даже вызовы функций.

Например, см. Фрагмент кода ниже

```
int binsearch(int x, int v[], int n)
{
   int low = 0;
   int high = n - 1;
   int mid;
   ...
}
```

вместо

```
   int low, high, mid;

   low = 0;
   high = n - 1;
```

По сути, инициализация автоматических переменных является просто сокращением для операторов присваивания. Какая форма предпочтительнее во многом зависит от вкуса. Обычно мы используем явные присваивания, потому что инициализаторы в объявлениях сложнее видеть и удаляться от точки использования. С другой стороны, переменные должны быть объявлены только тогда, когда они будут использоваться, когда это

возможно.

Инициализация массива:

Массив может быть инициализирован, следуя его объявлению с помощью списка инициализаторов, заключенных в фигурные скобки и разделенных запятыми.

Например, чтобы инициализировать дни массива с количеством дней в каждом месяце:

```
int days_of_month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

(Обратите внимание, что январь в этой структуре закодирован как нулевой месяц.)

Когда размер массива опущен, компилятор будет вычислять длину, подсчитывая инициализаторы, из которых в этом случае 12.

Если для массива меньше инициализаторов, чем указанный размер, остальные будут равны нулю для всех типов переменных.

Ошибка слишком много инициализаторов. Нет стандартного способа указать повторение инициализатора, но GCC имеет расширение для этого.

C99

В C89 / C90 или более ранних версиях C не было возможности инициализировать элемент в середине массива без предоставления всех предыдущих значений.

C99

С C99 и выше назначенные инициализаторы позволяют инициализировать произвольные элементы массива, оставляя любые неинициализированные значения нулями.

Инициализация массивов символов:

Массивы символов - это особый случай инициализации; вместо привязок и запятой можно использовать строку:

```
char chr_array[] = "hello";
```

является сокращением длинного, но эквивалентного:

```
char chr_array[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

В этом случае размер массива составляет шесть (пять символов плюс завершающий `'\0'`).

[1] Что происходит с объявленной, неинициализированной переменной в C? Имеет ли это значение?

[2] Обратите внимание, что *константное выражение* определяется как то, что может быть оценено во время компиляции. Итак, `int global_var = f();` является недействительным. Другое распространенное заблуждение думает о `const` квалифицированного переменных в качестве *постоянного выражения.* В C, `const` означает «только для чтения», а не «время компиляции постоянная». Итак, глобальные определения, такие как `const int SIZE = 10; int global_arr[SIZE];` и `const int SIZE = 10; int global_var = SIZE;` не являются законными в C.

Инициализация структур и массивов структур

Структуры и массивы структур могут быть инициализированы рядом значений, заключенных в фигурные скобки, по одному значению на члена структуры.

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { 1776, 7, 4 };

struct Date uk_battles[] =
{
    { 1066, 10, 14 },    // Battle of Hastings
    { 1815,  6, 18 },    // Battle of Waterloo
    { 1805, 10, 21 },    // Battle of Trafalgar
};
```

Обратите внимание, что инициализация массива может быть записана без внутренних фигурных скобок, а в прошлом (до 1990 года, скажем, часто) было бы написано без них:

```
struct Date uk_battles[] =
{
    1066, 10, 14,    // Battle of Hastings
    1815,  6, 18,    // Battle of Waterloo
    1805, 10, 21,    // Battle of Trafalgar
};
```

Хотя это работает, это не очень хороший современный стиль - вы не должны пытаться использовать эту нотацию в новом коде и должны исправлять предупреждения компилятора, которые он обычно дает.

См. Также назначенные инициализаторы .

Использование назначенных инициализаторов

C99

В C99 была введена концепция *назначенных инициализаторов.* Они позволяют указать, какие элементы массива, структуры или объединения должны быть инициализированы

следующими значениями.

Обозначенные инициализаторы для элементов массива

Для простого типа, такого как обычный `int` :

```
int array[] = { [4] = 29, [5] = 31, [17] = 101, [18] = 103, [19] = 107, [20] = 109 };
```

Термин в квадратных скобках, который может быть любым константным целочисленным выражением, указывает, какой элемент массива должен быть инициализирован значением члена после знака `=` . Необычные элементы инициализируются по умолчанию, что означает, что определены нули. В этом примере показаны назначенные инициализаторы по порядку; они не должны быть в порядке. В примере показаны пробелы; они являются законными. В примере не показаны две разные инициализации для одного и того же элемента; что также допускается (ИСО / МЭК 9899: 2011, §6.7.9 Инициализация, ¶19 *Инициализация должна* выполняться *в порядке списка инициализаторов, причем каждый инициализатор предоставил конкретный подобъект, переопределяющий любой ранее указанный инициализатор для того же подобъекта*).

В этом примере размер массива не определен явно, поэтому максимальный индекс, указанный в назначенных инициализаторах, определяет размер массива - который будет 21 элементом в примере. Если размер был определен, инициализация записи за пределами конца массива была бы ошибкой, как обычно.

Назначенные инициализаторы для структур

Вы можете указать, какие элементы структуры инициализируются с помощью `.` обозначение *`element`* :

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { .day = 4, .month = 7, .year = 1776 };
```

Если элементы не указаны, они по умолчанию инициализируются (обнуляются).

Назначенный инициализатор для объединений

Вы можете указать, какой элемент объединения инициализируется назначенным инициализатором.

C89

До стандарта C не было возможности инициализировать `union` . Стандарт C89 / C90 позволяет вам инициализировать первый член `union` - поэтому выбор, который перечисляется в списке, имеет первостепенное значение.

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    } du;
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du = { .du_int = 1 } };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du = { .du_double = 3.14159 }
};
```

C11

Обратите внимание, что C11 позволяет использовать анонимные члены объединения внутри структуры, так что вам не нужно имя `du` в предыдущем примере:

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    };
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du_int = 1 };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du_double = 3.14159 };
```

Назначенные инициализаторы для массивов структур и т. Д.

Эти конструкции могут быть объединены для массивов структур, содержащих элементы, которые являются массивами и т. Д. Использование полных наборов фигурных скобок гарантирует, что обозначение однозначно.

```
typedef struct Date Date;  // See earlier in this example

struct date_range
{
    Date    dr_from;
    Date    dr_to;
    char    dr_what[80];
};

struct date_range ranges[] =
{
    [3] = { .dr_from = { .year = 1066, .month = 10, .day = 14 },
```

```
            .dr_to   = { .year = 1066, .month = 12, .day = 25 },
            .dr_what = "Battle of Hastings to Coronation of William the Conqueror",
          },
    [2] = { .dr_from = { .month = 7, .day =  4, .year = 1776 },
            .dr_to   = { .month = 5, .day = 14, .year = 1787 },
            .dr_what = "US Declaration of Independence to Constitutional Convention",
          }
};
```

Определение диапазонов в инициализаторах массива

GCC предоставляет расширение , позволяющее указать диапазон элементов в массиве, которому должен быть присвоен одинаковый инициализатор:

```
int array[] = { [3 ... 7] = 29, 19 = 107 };
```

Тройные точки должны быть отделены от чисел, чтобы одна из точек не интерпретировалась как часть числа с плавающей запятой (*максимальное* правило *munch*).

Прочитайте инициализация онлайн: https://riptutorial.com/ru/c/topic/4547/инициализация

глава 16: Итерационные выражения / Циклы: для, пока, делать-пока

Синтаксис

- / * все версии * /
- for ([выражение]; [выражение]; [выражение]) one_statement
- for ([выражение]; [выражение]; [выражение]) {нуль или несколько операторов}
- while (выражение) one_statement
- while (выражение) {нуль или несколько утверждений}
- делать one_statement while (выражение);
- делать {одно или несколько утверждений} while (выражение);
- // с C99 в дополнение к форме выше
- for (declaration; [expression]; [expression]) one_statement;
- for (declaration; [expression]; [expression]) {zero или несколько операторов}

замечания

Итерационное заявление / Петли подразделяются на две категории:

- управляемый итерацией оператор / петли
- управление итерацией с педалью / петлями

Заглавие итерации с управляемой головой / петли

```
for ([<expression>]; [<expression>]; [<expression>]) <statement>
while (<expression>) <statement>
```

C99

```
for ([declaration expression]; [expression] [; [expression]]) statement
```

Операция итерации с педальным управлением / петли

```
do <statement> while (<expression>);
```

Examples

Для цикла

Чтобы снова выполнить блок кода заново, на изображение попадают петли. Цикл `for` должен использоваться, когда блок кода должен выполняться фиксированное количество раз. Например, чтобы заполнить массив размером `n` с помощью пользовательских входов, нам нужно выполнить `scanf()` для `n` раз.

C99

```
#include <stddef.h>          // for size_t

int array[10];               // array of 10 int

for (size_t i = 0; i < 10; i++)    // i starts at 0 and finishes with 9
{
   scanf("%d", &array[i]);
}
```

Таким образом, вызов функции `scanf()` выполняется `n` раз (10 раз в нашем примере), но записывается только один раз.

Здесь переменная `i` является индексом цикла, и ее лучше всего объявить как представленную. Тип `size_t` (*тип размера*) должен использоваться для всего, что подсчитывает или пересекает объекты данных.

Этот способ объявления переменных внутри `for` доступен только для компиляторов, которые были обновлены до стандарта C99. Если по какой-то причине вы все еще придерживаетесь более старого компилятора, вы можете объявить индекс цикла перед циклом `for` :

C99

```
#include <stddef.h>        /* for size_t */
size_t i;
int array[10];             /* array of 10 int */

for (i = 0; i < 10; i++)       /* i starts at 0 and finishes at 9 */
{
   scanf("%d", &array[i]);
}
```

Пока цикл

В `while` цикл используется для выполнения фрагмента кода в то время как условие истинно. В `while` цикл должен быть использован , когда блок кода должен быть выполнен переменное число раз. Например, показанный код получает пользовательский ввод, если пользователь вставляет числа, которые не равны `0` . Если пользователь вставляет `0` , условие while больше не является истинным, поэтому выполнение завершает цикл и переходит к следующему коду:

```
int num = 1;
```

```
while (num != 0)
{
    scanf("%d", &num);
}
```

Цикл Do-While

В отличии от `for` и в `while` петли, `do-while` петли проверить истинность условия в конце цикла, что означает , что `do` блок будет выполняться один раз, а затем проверить состояние `while` в нижней части блока. Это означает, что цикл `do-while` while *всегда* запускается хотя бы один раз.

Например, этот цикл `do-while` while будет получать числа от пользователя, пока сумма этих значений больше или равна `50` :

```
int num, sum;
num = sum = 0;

do
{
  scanf("%d", &num);
  sum += num;

} while (sum < 50);
```

петли `do-while` относительно редки в большинстве стилей программирования.

Структура и поток управления в цикле for

```
for ([declaration-or-expression]; [expression2]; [expression3])
{
    /* body of the loop */
}
```

В `for` цикла, условие цикла имеет три выражения, все по желанию.

- Первое выражение, выражение `declaration-or-expression` , *инициализирует* цикл. Он выполняется ровно один раз в начале цикла.

C99

Это может быть либо объявление, либо инициализация переменной цикла, либо общее выражение. Если это объявление, область действия объявленной переменной ограничена оператором `for` .

C99

Исторические версии C допускают только выражение, здесь и объявление переменной цикла должно быть помещено перед `for` .

- Второе выражение, `expression2` , является *условием проверки* . Он выполняется сначала после инициализации. Если условие `true` , тогда элемент управления входит в тело цикла. Если нет, то он сдвигается за пределы тела цикла в конце цикла. Впоследствии этот conditon проверяется после каждого выполнения тела, а также оператора обновления. Когда `true` , управление возвращается к началу тела цикла. Условие обычно предназначено для проверки количества циклов, выполняемых телом цикла. Это основной способ выхода из цикла, другой способ использования операторов перехода .
- Третье выражение, `expression3` , является *оператором обновления* . Он выполняется после каждого выполнения тела цикла. Он часто используется для увеличения переменной, поддерживающей подсчет количества раз, когда тело цикла выполнялось, и эта переменная называется *итератором* .

Каждый экземпляр выполнения тела цикла называется *итерацией* .

Пример:

C99

```
for(int i = 0; i < 10 ; i++)
{
    printf("%d", i);
}
```

Выход:

```
0123456789
```

В приведенном выше примере выполняется первое `i = 0` , инициализирующее `i` . Затем проверяется условие `i < 10` , которое считается `true` . Элемент управления входит в тело цикла и печатается значение `i` . Затем управление переходит к `i++` , обновляя значение `i` от 0 до 1. Затем условие снова проверяется, и процесс продолжается. Это продолжается до тех пор, пока значение `i` станет равным 10. Тогда условие `i < 10` оценивает значение `false` , после чего элемент управления выходит из цикла.

Бесконечные петли

Цикл называется *бесконечным циклом,* если элемент управления входит, но никогда не покидает тело цикла. Это происходит, когда тестовое условие цикла никогда не оценивается как `false` .

Пример:

C99

```
for (int i = 0; i >= 0; )
{
```

```
    /* body of the loop where i is not changed*/
}
```

В приведенном выше примере переменная `i` , итератор инициализируется равным 0. Условие испытания первоначально `true` . Однако `i` не изменяется нигде в теле, и выражение update пусто. Следовательно, `i` останется 0, и условие теста никогда не будет оцениваться как `false` , что приведет к бесконечному циклу.

Предполагая, что нет инструкций перехода, другой способ, которым может быть сформирован бесконечный цикл, заключается в явном сохранении условия true:

```
while (true)
{
    /* body of the loop */
}
```

В цикле `for` оператор условия необязателен. В этом случае условие всегда `true` пусто, что приводит к бесконечной петле.

```
for (;;)
{
    /* body of the loop */
}
```

Однако в некоторых случаях условие может сохраняться `true` намеренно, с намерением выйти из цикла с помощью инструкции перехода, например `break` .

```
while (true)
{
    /* statements */
    if (condition)
    {
         /* more statements */
         break;
    }
}
```

Прокрутка Loop и устройство Duff

Иногда прямая петля не может полностью содержаться внутри тела цикла. Это связано с тем, что цикл должен быть загружен некоторыми операторами ***B.*** Затем итерация начинается с некоторых операторов ***A*** , которые затем следуют за ***B*** перед циклом.

```
do_B();
while (condition) {
    do_A();
    do_B();
}
```

Для того, чтобы избежать возможных проблем вырезать / вставить с повторяющимися ***B***

дважды в коде, устройство Дафф может быть применен , чтобы начать цикл от середины в `while` тело, с помощью переключателя заявление и провалиться поведения.

```
switch (true) while (condition) {
case false: do_A(); /* FALL THROUGH */
default:    do_B(); /* FALL THROUGH */
}
```

Устройство Duff было изобретено для развертывания цикла. Представьте, что вы применяете маску к блоку памяти, где `n` - тип подписанного интеграла с положительным значением.

```
do {
    *ptr++ ^= mask;
} while (--n > 0);
```

Если `n` всегда делится на 4, вы можете легко развернуть это:

```
do {
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
} while ((n -= 4) > 0);
```

Но, с помощью устройства Даффа, код может следовать за этой разворачивающейся идиомой, которая прыгает в нужное место в середине цикла, если `n` не делится на 4.

```
switch (n % 4) do {
case 0: *ptr++ ^= mask; /* FALL THROUGH */
case 3: *ptr++ ^= mask; /* FALL THROUGH */
case 2: *ptr++ ^= mask; /* FALL THROUGH */
case 1: *ptr++ ^= mask; /* FALL THROUGH */
} while ((n -= 4) > 0);
```

Такой ручный разворот редко требуется с современными компиляторами, поскольку механизм оптимизации компилятора может разворачивать петли от имени программиста.

Прочитайте Итерационные выражения / Циклы: для, пока, делать-пока онлайн: https://riptutorial.com/ru/c/topic/5151/итерационные-выражения---циклы--для--пока--делать-пока

глава 17: Классы хранения

Вступление

Класс хранения используется для задания области действия переменной или функции. Зная класс хранения переменной, мы можем определить время жизни этой переменной в течение времени выполнения программы.

Синтаксис

- [auto | register | static | extern] <Тип данных> <Имя переменной> [= <Значение>];
- [static _Thread_local | extern _Thread_local | _Thread_local] <Тип данных> <Имя переменной> [= <Значение>]; / *, так как> = C11 * /
- Примеры:
- typedef int *foo* ;
- extern int *foo* [2];

замечания

Спецификаторы класса хранения - это ключевые слова, которые могут отображаться рядом с типом объявления верхнего уровня. Использование этих ключевых слов влияет на продолжительность хранения и привязку объявленного объекта в зависимости от того, объявлено ли оно в области файла или в области блока:

Ключевое слово	Продолжительность хранения	связь	замечания
`static`	статический	внутренний	Устанавливает внутреннюю привязку для объектов в области файлов; устанавливает статическую длительность хранения для объектов в области блока.
`extern`	статический	внешний	Подразумевается и поэтому избыточно для объектов, определенных в области файлов, которые также имеют инициализатор. При

Ключевое слово	Продолжительность хранения	связь	замечания
			использовании в объявлении в области файла без инициализатора намеки на то, что определение должно быть найдено в другой единицы перевода и будет разрешено во время ссылки.
`auto`	автоматическая	Ненужные	Подразумевается и поэтому избыточно для объектов, объявленных в области блока.
`register`	автоматическая	Ненужные	Релевантно относится только к объектам с автоматическим временем хранения. Предоставляет подсказку о том, что переменная должна храниться в регистре. Наложенное ограничение состоит в том, что нельзя использовать унарный `&` «адрес» оператора на таком объекте, и поэтому объект нельзя сгладить.
`typedef`	Ненужные	Ненужные	Не спецификатор класса хранения на практике, но работает как один с синтаксической точки зрения. Единственное отличие состоит в том, что объявленный идентификатор является типом, а не объектом.
`_Thread_local`	Нить	Внутренний / внешний	Представлен в C11 для представления *продолжительности хранения потоков* . Если используется в области блока, он также должен включать в себя `extern` или `static` .

Каждый объект имеет связанную длительность хранения (независимо от области действия) и привязку (относится только к объявлениям только в области файлов), даже если эти ключевые слова опущены.

Упорядочение спецификаторов класса хранения по отношению к спецификаторам типа верхнего уровня (`int` , `unsigned` , `short` и т. Д.) И квалификаторам верхнего уровня (`const` , `volatile`) не применяется, поэтому оба этих объявления действительны:

```
int static const unsigned a = 5; /* bad practice */
static const unsigned int b = 5; /* good practice */
```

Однако считается хорошей практикой сначала поставить спецификаторы класса хранения, затем квалификаторы типов, а затем спецификатор типа (`void` , `char` , `int` , `signed long` , `unsigned long long` , `long double` ...).

Не все спецификаторы класса хранения являются законными в определенной области:

```
register int x; /* legal at block scope, illegal at file scope */
auto int y; /* same */

static int z; /* legal at both file and block scope */
extern int a; /* same */

extern int b = 5; /* legal and redundant at file scope, illegal at block scope */

/* legal because typedef is treated like a storage class specifier syntactically */
int typedef new_type_name;
```

Продолжительность хранения

Длительность хранения может быть как статической, так и автоматической. Для объявленного объекта он определяется в зависимости от его области действия и спецификаторов класса хранения.

Статическая продолжительность хранения

Переменные со статическим хранением сохраняются в течение всего выполнения программы и могут быть объявлены как в области файлов (с или без `static`), так и в области блока (путем `static` ввода). Они обычно выделяются и инициализируются операционной системой при запуске программы и возвращаются, когда процесс завершается. На практике исполняемые форматы имеют выделенные разделы для таких переменных (`data` , `bss` и `rodata`), и все эти разделы из файла отображаются в памяти в определенных диапазонах.

Продолжительность хранения резьбы

C11

Эта продолжительность хранения была введена в C11. Это было недоступно в более ранних стандартах C. Некоторые компиляторы предоставляют нестандартное расширение с аналогичной семантикой. Например, gcc поддерживает `__thread` который может использоваться в более ранних стандартах C, у которых не было `_Thread_local` .

Переменные с продолжительностью хранения потоков могут быть объявлены как в области файлов, так и в области блоков. Если объявлено в области блока, оно также должно использовать `static` или `extern` хранитель. Его время жизни - это полное выполнение *потока,* в котором он создан. Это единственный спецификатор хранилища, который может появляться рядом с другим спецификатором хранилища.

Автоматическая продолжительность хранения

Переменные с автоматической продолжительностью хранения могут быть объявлены только в области блока (непосредственно внутри функции или внутри блока в этой функции). Они могут использоваться только в период между входом и выходом из функции или блока. Как только переменная выходит из области видимости (либо путем возврата из функции, либо путем выхода из блока), ее память автоматически освобождается. Любые дополнительные ссылки на одну и ту же переменную из указателей являются недопустимыми и приводят к неопределенному поведению.

В типичных реализациях автоматические переменные располагаются при определенных смещениях в кадре стека функции или в регистрах.

Внешняя и внутренняя связь

Связывание относится только к объектам (функциям и переменным), объявленным в области файлов, и влияет на их видимость в разных единицах перевода. Объекты с внешней связью видны во всех других единицах перевода (при условии, что включена соответствующая декларация). Объекты с внутренней связью не подвергаются воздействию других единиц перевода и могут использоваться только в блоке перевода, где они определены.

Examples

ЬурейеЕ

Определяет новый тип на основе существующего типа. Его синтаксис является зеркальным отображением объявления переменной.

```
/* Byte can be used wherever `unsigned char` is needed */
typedef unsigned char Byte;
```

```
/* Integer is the type used to declare an array consisting of a single int */
typedef int Integer[1];

/* NodeRef is a type used for pointers to a structure type with the tag "node" */
typedef struct node *NodeRef;

/* SigHandler is the function pointer type that gets passed to the signal function. */
typedef void (*SigHandler)(int);
```

Хотя технически не класс хранения, компилятор будет рассматривать его как один, поскольку ни один из других классов хранения не разрешен, если используется ключевое слово `typedef` .

`typedef` s важны и не должны заменяться макросом `#define` .

```
typedef int newType;
newType *ptr;        // ptr is pointer to variable of type 'newType' aka int
```

Тем не мение,

```
#define int newType
newType *ptr;        // Even though macros are exact replacements to words, this doesn't
result to a pointer to variable of type 'newType' aka int
```

авто

Этот класс хранения означает, что идентификатор имеет автоматическую продолжительность хранения. Это означает, что когда область, в которой был определен идентификатор, заканчивается, объект, обозначенный идентификатором, более недействителен.

Поскольку все объекты, не проживающие в глобальном масштабе или объявленные `static` , имеют автоматическую продолжительность хранения по умолчанию при определении, это ключевое слово в основном представляет собой исторический интерес и не должно использоваться:

```
int foo(void)
{
    /* An integer with automatic storage duration. */
    auto int i = 3;

    /* Same */
    int j = 5;

    return 0;
} /* The values of i and j are no longer able to be used. */
```

статический

`static` класс хранения используется в разных целях, в зависимости от местоположения объявления в файле:

1. Ограничить идентификатор только этой единицей перевода (scope = file).

```
/* No other translation unit can use this variable. */
static int i;

/* Same; static is attached to the function type of f, not the return type int. */
static int f(int n);
```

2. Чтобы сохранить данные для использования со следующим вызовом функции (scope = block):

```
void foo()
{
    static int a = 0; /* has static storage duration and its lifetime is the
                       * entire execution of the program; initialized to 0 on
                       * first function call */
    int b = 0; /* b has block scope and has automatic storage duration and
                * only "exists" within function */

    a += 10;
    b += 10;

    printf("static int a = %d, int b = %d\n", a, b);
}

int main(void)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        foo();
    }

    return 0;
}
```

Этот код печатает:

```
static int a = 10, int b = 10
static int a = 20, int b = 10
static int a = 30, int b = 10
static int a = 40, int b = 10
static int a = 50, int b = 10
```

Статические переменные сохраняют свое значение даже при вызове из нескольких разных потоков.

C99

3. Ожидается, что в функциональных параметрах для обозначения массива будет иметься постоянное минимальное количество элементов и ненулевой параметр:

```
/* a is expected to have at least 512 elements. */
void printInts(int a[static 512])
{
    size_t i;
    for (i = 0; i < 512; ++i)
        printf("%d\n", a[i]);
}
```

Требуемое количество элементов (или даже ненулевой указатель) не обязательно проверяется компилятором, и компиляторы не обязаны уведомлять вас каким-либо образом, если у вас недостаточно элементов. Если программист передает менее 512 элементов или нулевой указатель, результатом является неопределенное поведение. Так как это невозможно обеспечить, необходимо использовать особую осторожность при передаче значения для этого параметра такой функции.

внешний

Используется для **объявления объекта или функции,** которая определена в другом месте (и имеет *внешнюю связь*). В общем случае он используется для объявления объекта или функции, которые будут использоваться в модуле, который не является тем, в котором определен соответствующий объект или функция:

```
/* file1.c */
int foo = 2;  /* Has external linkage since it is declared at file scope. */
```

```
/* file2.c */
#include <stdio.h>
int main(void)
{
    /* `extern` keyword refers to external definition of `foo`. */
    extern int foo;
    printf("%d\n", foo);
    return 0;
}
```

C99

Вещи немного интереснее с введением ключевого слова `inline` в C99:

```
/* Should usually be place in a header file such that all users see the definition */
/* Hints to the compiler that the function `bar` might be inlined */
/* and suppresses the generation of an external symbol, unless stated otherwise. */
inline void bar(int drink)
{
    printf("You ordered drink no.%d\n", drink);
}

/* To be found in just one .c file.
   Creates an external function definition of `bar` for use by other files.
   The compiler is allowed to choose between the inline version and the external
   definition when `bar` is called. Without this line, `bar` would only be an inline
   function, and other files would not be able to call it. */
```

```
extern void bar(int);
```

регистр

Подсказки к компилятору, что доступ к объекту должен быть как можно быстрее. Независимо от того, использует ли данный компилятор подсказку, определяется ли реализация; он может просто рассматривать его как эквивалент `auto` .

Единственное свойство, которое окончательно отличается для всех объектов, объявленных с помощью `register` состоит в том, что они не могут вычислить свой адрес. Таким образом, `register` может быть хорошим инструментом для обеспечения определенных оптимизаций:

```
register size_t size = 467;
```

это объект, который никогда не может быть *псевдонимом,* потому что ни один код не может передать свой адрес другой функции, где он может быть неожиданно изменен.

Это свойство также означает, что массив

```
register int array[5];
```

не может распадаться в указатель на его первый элемент (т.е. `array` превращается в `&array[0]`). Это означает, что элементы такого массива не могут быть доступны, и сам массив не может быть передан функции.

Фактически единственным законным использованием массива, объявленного с классом хранения `register` является оператор `sizeof` ; любой другой оператор потребует адрес первого элемента массива. По этой причине массивы обычно не должны быть объявлены с ключевым словом `register` поскольку они делают их бесполезными для чего-либо, кроме вычисления размера всего массива, что можно сделать так же легко без ключевого слова `register` .

Класс хранения `register` более подходит для переменных, которые определены внутри блока, и к ним обращаются с высокой частотой. Например,

```
/* prints the sum of the first 5 integers*/
/* code assumed to be part of a function body*/
{
    register int k, sum;
    for(k = 1, sum = 0; k < 6; sum += k, k++);
        printf("\t%d\n",sum);
}
```

C11

Оператору `_Alignof` также разрешено использовать с `register` массивами.

_Thread_local

C11

Это был новый спецификатор хранилища, введенный в C11, а также многопоточность. Это не доступно в более ранних стандартах C.

Обозначает *продолжительность хранения потоков* . Переменная, объявленная с помощью `_Thread_local` хранилища `_Thread_local` означает, что объект является *локальным для этого потока,* а его время жизни - это полное выполнение потока, в котором он создан. Он также может появляться вместе со `static` или `extern` .

```
#include <threads.h>
#include <stdio.h>
#define SIZE 5

int thread_func(void *id)
{
    /* thread local variable i. */
    static _Thread_local int i;

    /* Prints the ID passed from main() and the address of the i.
     * Running this program will print different addresses for i, showing
     * that they are all distinct objects. */
    printf("From thread:[%d], Address of i (thread local): %p\n", *(int*)id, (void*)&i);

    return 0;
}

int main(void)
{
    thrd_t id[SIZE];
    int arr[SIZE] = {1, 2, 3, 4, 5};

    /* create 5 threads. */
    for(int i = 0; i < SIZE; i++) {
        thrd_create(&id[i], thread_func, &arr[i]);
    }

    /* wait for threads to complete. */
    for(int i = 0; i < SIZE; i++) {
        thrd_join(id[i], NULL);
    }
}
```

Прочитайте Классы хранения онлайн: https://riptutorial.com/ru/c/topic/3597/классы-хранения

глава 18: Комментарии

Вступление

Комментарии используются, чтобы указать что-то человеку, читающему код. Комментарии обрабатываются как пустые компилятором и ничего не меняют в фактическом значении кода. Есть два синтаксиса, используемые для комментариев в C, оригинал `/* */` и немного более новый `//` . Некоторые системы документации используют специально отформатированные комментарии, чтобы помочь создать документацию для кода.

Синтаксис

- `/*...*/`
- `//...` (только с C99 и позже)

Examples

/ * * / разделили комментарии

Комментарий начинается с прямой косой черты, за которой сразу следует звездочка (`/*`), и заканчивается, как только появляется звездочка, сразу же за которой следует косая черта (`*/`). Все, что находится между этими комбинациями символов, является комментарием и рассматривается как пустой (в основном игнорируемый) компилятором.

```
/* this is a comment */
```

Комментарий выше - комментарий в одной строке. Комментарии этого `/*` типа могут охватывать несколько строк, например:

```
/* this is a
multi-line
comment */
```

Хотя это не является строго необходимым, соглашение об общем стиле с многострочными комментариями заключается в том, чтобы помещать ведущие пробелы и звездочки в строки после первого и `/*` и `*/` на новые строки, чтобы все они выстроились в очередь:

```
/*
 * this is a
 * multi-line
 * comment
 */
```

Дополнительные звездочки не имеют никакого функционального влияния на комментарий,

так как ни один из них не имеет связанной косой черты.

Эти `/*` типы комментариев могут использоваться в их собственной строке, в конце строки кода или даже в строках кода:

```
/* this comment is on its own line */
if (x && y) { /*this comment is at the end of a line */
    if ((complexCondition1) /* this comment is within a line of code */
            && (complexCondition2)) {
        /* this comment is within an if, on its own line */
    }
}
```

Комментарии не могут быть вложенными. Это связано с тем, что любой последующий `/*` будет проигнорирован (как часть комментария), а первый `*/` достиг будет рассматриваться как окончание комментария. Комментарий в следующем примере *не будет работать* :

```
/* outer comment, means this is ignored => /* attempted inner comment */ <= ends the comment,
not this one => */
```

Чтобы прокомментировать блоки кода, содержащие комментарии этого типа, которые в противном случае были бы вложенными, см. Комментарий, используя пример препроцессора ниже

// Ограниченные комментарии

C99

C99 представил использование однострочных комментариев в стиле C ++. Этот тип комментариев начинается с двух косых черт и проходит до конца строки:

```
// this is a comment
```

Этот комментарий не позволяет многострочные комментарии, хотя можно сделать блок комментариев, добавив несколько комментариев одной строки один за другим:

```
// each of these lines are a single-line comment
// note how each must start with
// the double forward-slash
```

Этот тип комментариев может использоваться в отдельной строке или в конце строки кода. Однако, поскольку они работают *до конца строки* , они *не* могут использоваться в пределах строки кода

```
// this comment is on its own line
if (x && y) { // this comment is at the end of a line
    // this comment is within an if, on its own line
}
```

Комментирование с использованием препроцессора

Большие куски кода также можно «прокомментировать», используя директивы препроцессора `#if 0` и `#endif` . Это полезно, когда код содержит многострочные комментарии, которые в противном случае не будут вложены.

```
#if 0 /* Starts the "comment", anything from here on is removed by preprocessor */

/* A large amount of code with multi-line comments */
int foo()
{
    /* lots of code */
    ...

    /* ... some comment describing the if statement ... */
    if (someTest) {
        /* some more comments */
        return 1;
    }

    return 0;
}

#endif /* 0 */

/* code from here on is "uncommented" (included in compiled executable) */
...
```

Возможная ловушка из-за триграфов

C99

При написании `//` разделительных комментариев можно сделать типографскую ошибку, которая влияет на их ожидаемую работу. Если один тип:

```
int x = 20;  // Why did I do this??/
```

`/` В конце была опечатка, но теперь она будет интерпретирована в `\` . Это потому, что `??/` Образует триграф .

`??/` триграф на самом деле обыкновенное письмо обозначение `\` , который является символом строки продолжения. Это означает, что компилятор считает, что следующая строка является продолжением текущей строки, т. Е. Продолжением комментария, что может и не быть тем, что предназначено.

```
int foo = 20; // Start at 20 ??/
int bar = 0;

// The following will cause a compilation error (undeclared variable 'bar')
// because 'int bar = 0;' is part of the comment on the preceding line
bar += foo;
```

Прочитайте Комментарии онлайн: https://riptutorial.com/ru/c/topic/10670/комментарии

глава 19: компиляция

Вступление

Язык C традиционно является скомпилированным языком (в отличие от интерпретации). Стандарт C определяет ***фазы перевода*** , а продукт их применения представляет собой программный образ (или скомпилированную программу). В c11 фазы перечислены в п. 5.1.1.2.

замечания

Расширение имени файла	Описание
`.c`	Исходный файл. Обычно содержит определения и код.
`.h`	Файл заголовка. Обычно содержит декларации.
`.o`	Файл объекта. Скомпилированный код в машинном языке.
`.obj`	Альтернативное расширение для объектных файлов.
`.a`	Файл библиотеки. Пакет объектных файлов.
`.dll`	Динамическая библиотека ссылок в Windows.
`.so`	Общий объект (библиотека) во многих Unix-подобных системах.
`.dylib`	Библиотека динамических ссылок в OSX (вариант Unix).
`.exe` , `.com`	Исполняемый файл Windows. Создано путем связывания объектных файлов и файлов библиотек. В Unix-подобных системах для исполняемого файла нет специального расширения имени файла.

Флаги компилятора POSIX c99	Описание
`-o filename`	Имя выходного файла, например. (`bin/program.exe` , `program`)
`-I directory`	поиск заголовков в `direrctory` .
`-D name`	определить `name` макроса

Флаги компилятора POSIX c99	Описание
`-L directory`	поиск библиотек в `directory` .
`-l name`	link library `libname` .

Компиляторы на платформах POSIX (Linux, мейнфреймы, Mac) обычно принимают эти параметры, даже если они не называются `c99` .

- См. Также c99 - скомпилируйте стандартные программы C

Флаги GCC (сборник компиляторов GNU)	Описание
`-Wall`	Позволяет использовать все предупреждающие сообщения, которые обычно считаются полезными.
`-Wextra`	Включает больше предупреждающих сообщений, может быть слишком шумным.
`-pedantic`	Принудительные предупреждения, когда код нарушает выбранный стандарт.
`-Wconversion`	Включить предупреждения о неявной конверсии, использовать с осторожностью.
`-c`	Компилирует исходные файлы без ссылки.
`-v`	Распечатывает информацию о компиляции.

- `gcc` принимает флаги POSIX и многие другие.
- Многие другие компиляторы на платформах POSIX (`clang` , компиляторы конкретных поставщиков) также используют флаги, перечисленные выше.
- См. Также Вызов GCC для многих других опций.

Флаги TCC (Tiny C Compiler)	Описание
`-Wimplicit-function-declaration`	Предупреждать о объявлении неявной функции.
`-Wunsupported`	Предупредите о неподдерживаемых функциях GCC, которые TCC игнорируют.
`-Wwrite-strings`	Сделать строковые константы типа const char * вместо char *.

Флаги TCC (Tiny C Compiler)	Описание
`-Werror`	Прерывание компиляции, если выдается предупреждение.
`-Wall`	Активируйте все предупреждения, кроме `-Werror` , `-Wunusupported` и `-Wwrite strings` .

Examples

Линкер

Задача компоновщика заключается в том, чтобы связать кучу объектных файлов (файлы `.o`) в двоичном исполняемом файле. Процесс *связывания в* основном включает в себя *разрешение символических адресов на числовые адреса* . Результатом процесса ссылки обычно является исполняемая программа.

Во время процесса компоновки компоновщик будет отображать все объектные модули, указанные в командной строке, добавить перед собой системный *код запуска* и попытаться разрешить все *внешние* ссылки в объектном модуле с *внешними определениями* в других объектных файлах (объектные файлы могут быть указаны непосредственно в командной строке или могут быть неявно добавлены через библиотеки). Затем он назначит *адреса загрузки* для объектных файлов, то есть определяет, где код и данные будут попадать в адресное пространство готовой программы. Как только он получит адреса загрузки, он может заменить все символические адреса в объектном коде «реальными», числовыми адресами в адресном пространстве целевого объекта. Программа готова к выполнению сейчас.

Сюда входят как объектные файлы, созданные компилятором из файлов исходного кода, так и файлы объектов, которые были предварительно скомпилированы для вас и собраны в файлы библиотек. Эти файлы имеют имена, которые заканчиваются на `.a` или `.so` , и вам обычно не нужно знать о них, поскольку компоновщик знает, где большинство из них находится, и свяжет их автоматически по мере необходимости.

Неявный вызов компоновщика

Подобно препроцессору, компоновщик представляет собой отдельную программу, часто называемую `ld` (но Linux использует `collect2` , например). Также как и предварительный процессор, компоновщик автоматически запускается для вас, когда вы используете компилятор. Таким образом, обычным способом использования компоновщика является следующее:

```
% gcc foo.o bar.o baz.o -o myprog
```

Эта строка сообщает компилятору связать три объектных файла (`foo.o` , `bar.o` и `baz.o`) в двоичный исполняемый файл с именем `myprog` . Теперь у вас есть файл под названием `myprog` который вы можете запустить, и который, надеюсь, сделает что-нибудь классное и / или полезное.

Явный вызов компоновщика

Можно напрямую ссылаться на компоновщик, но это редко бывает целесообразным и, как правило, очень специфично для платформы. То есть опции, которые работают в Linux, не обязательно будут работать на Solaris, AIX, MacOS, Windows и аналогично для любой другой платформы. Если вы работаете с GCC, вы можете использовать `gcc -v` для просмотра того, что выполняется от вашего имени.

Варианты компоновщика

Линкер также принимает некоторые аргументы, чтобы изменить его поведение. Следующая команда сообщила бы gcc, чтобы связать `foo.o` и `bar.o` , но также включить библиотеку `ncurses` .

```
% gcc foo.o bar.o -o foo -lncurses
```

Это фактически (более или менее) эквивалентно

```
% gcc foo.o bar.o /usr/lib/libncurses.so -o foo
```

(хотя `libncurses.so` может быть `libncurses.a` , который является просто архивом, созданным с помощью `ar`). Обратите внимание, что вы должны перечислить библиотеки (либо по имени пути, либо через параметры `-lname`) после файлов объектов. С статическими библиотеками имеет значение порядок, в котором они указаны; часто, с общими библиотеками, порядок не имеет значения.

Обратите внимание, что во многих системах, если вы используете математические функции (из `<math.h>`), вам нужно указать `-lm` для загрузки библиотеки математики, но Mac OS X и macOS Sierra этого не требуют. Существуют и другие библиотеки, которые являются отдельными библиотеками в Linux и других Unix-системах, но не для потоков macOS-POSIX и POSIX в реальном времени, а также для сетевых библиотек. Следовательно, процесс связывания варьируется между платформами.

Другие варианты компиляции

Это все, что вам нужно знать, чтобы начать компиляцию ваших собственных программ на C. Как правило, мы также рекомендуем использовать `-Wall` командной строки `-Wall` :

```
% gcc -Wall -c foo.cc
```

Опция `-Wall` заставляет компилятор предупреждать вас о юридических, но сомнительных конструкциях кода, и поможет вам поймать множество ошибок очень рано.

Если вы хотите, чтобы компилятор выдавал вам больше предупреждений (включая переменные, объявленные, но не используемые, забывая вернуть значение и т. Д.), Вы можете использовать этот набор параметров, поскольку `-Wall` , несмотря на имя, не поворачивается *все возможные предупреждения* :

```
% gcc -Wall -Wextra -Wfloat-equal -Wundef -Wcast-align -Wwrite-strings -Wlogical-op \
>     -Wmissing-declarations -Wredundant-decls -Wshadow …
```

Обратите внимание, что у `clang` есть опция `-Weverything` которая действительно включает все предупреждения в `clang` .

Типы файлов

Для компиляции программ C вам необходимо работать с пятью типами файлов:

1. **Исходные файлы** : эти файлы содержат определения функций и имеют имена, которые заканчиваются на `.c` по соглашению. Примечание: `.cc` и `.cpp` - файлы C ++; *а не* файлы C.
   например, `foo.c`

2. **Заголовочные файлы** . Эти файлы содержат прототипы функций и различные предпроцессорные инструкции (см. Ниже). Они используются, чтобы файлы исходного кода могли получать доступ к внешним функциям. Заголовочные файлы заканчиваются на `.h` по соглашению.
   например, `foo.h`

3. **Объектные файлы** : эти файлы создаются как выходные данные компилятора. Они состоят из определений функций в двоичной форме, но сами по себе они не исполняются. Файлы объектов заканчиваются на `.o` по соглашению, хотя в некоторых операционных системах (например, Windows, MS-DOS) они часто заканчиваются на `.obj` .
   например, `foo.o foo.obj`

4. **Бинарные исполняемые файлы** : они создаются в качестве выхода программы, называемой «компоновщик». Компонент связывает вместе множество объектных файлов для создания двоичного файла, который может быть выполнен непосредственно. Бинарные исполняемые файлы не имеют специального суффикса в операционных системах Unix, хотя обычно они заканчиваются на `.exe` в Windows.
   например `foo foo.exe`

5. **Библиотеки** : библиотека является скомпилированным двоичным кодом, но сама по себе не является исполняемым (т. **Е.** В библиотеке нет функции `main()`). Библиотека

содержит функции, которые могут использоваться более чем одной программой. Библиотека должна поставляться с файлами заголовков, которые содержат прототипы для всех функций в библиотеке; эти файлы заголовков должны быть указаны (например, `#include <library.h>`) в любом исходном файле, который использует библиотеку. Затем компоновщик необходимо передать в библиотеку, чтобы программа могла успешно скомпилироваться. Существует два типа библиотек: статический и динамический.

- **Статическая библиотека** : статическая библиотека (`.a` файлы для POSIX-систем и `.lib` файлов для Windows - не путать с файлами библиотеки DLL-импорта , которые также используют расширение `.lib`) статически встроена в программу. Статические библиотеки имеют то преимущество, что программа точно знает, какая версия библиотеки используется. С другой стороны, размеры исполняемых файлов больше, поскольку все используемые функции библиотеки включены.
  например, `libfoo.a foo.lib`
- **Динамическая библиотека** : динамическая библиотека (`.so` файлы для большинства POSIX-систем, `.dylib` для OSX и `.dll` файлов для Windows) динамически связана во время выполнения программой. Они также иногда называются разделяемыми библиотеками, потому что одно из изображений библиотеки может использоваться многими программами. У динамических библиотек преимущество состоит в том, что они занимают меньше места на диске, если несколько библиотек используют более одного приложения. Кроме того, они позволяют обновлять библиотеки (исправления ошибок) без необходимости пересоздавать исполняемые файлы.
  например `foo.so foo.dylib foo.dll`

Препроцессор

Прежде чем компилятор C начнет компилировать файл исходного кода, файл обрабатывается на этапе предварительной обработки. Эта фаза может быть выполнена отдельной программой или полностью интегрирована в один исполняемый файл. В любом случае он автоматически запускается компилятором до начала собственно компиляции. Фаза предварительной обработки преобразует исходный код в другой исходный код или блок перевода, применяя текстовые замены. Вы можете думать об этом как о «модифицированном» или «расширенном» исходном коде. Этот расширенный источник может существовать как реальный файл в файловой системе, или он может храниться только в памяти в течение короткого времени перед дальнейшей обработкой.

Команды препроцессора начинаются с знака фунта («#»). Существует несколько команд препроцессора; два из наиболее важных:

1. **Определяет** :

`#define` в основном используется для определения констант. Например,

```
#define BIGNUM 1000000
int a = BIGNUM;
```

становится

```
int a = 1000000;
```

`#define` используется таким образом, чтобы избежать необходимости явно выписывать некоторое постоянное значение во многих разных местах файла исходного кода. Это важно, если вам нужно впоследствии изменить значение константы; он гораздо менее подвержен ошибкам, чтобы изменить его один раз в `#define` , чем изменять его в нескольких местах, разбросанных по всему коду.

Поскольку `#define` просто выполняет расширенный поиск и заменяет, вы также можете объявить макросы. Например:

```
#define ISTRUE(stm) do{stm = stm ? 1 : 0;}while(0)
// in the function:
a = x;
ISTRUE(a);
```

будет выглядеть так:

```
// in the function:
a = x;
do {
    a = a ? 1 : 0;
} while(0);
```

В первом приближении этот эффект примерно такой же, как и для встроенных функций, но препроцессор не обеспечивает проверку типов для макросов `#define` . Это, как известно, подвержено ошибкам, и их использование требует большой осторожности.

Также обратите внимание на то, что препроцессор также заменит комментарии пробелами, как описано ниже.

2. **Включает** :

`#include` используется для доступа к определениям функций, определенным за пределами файла исходного кода. Например:

```
 #include <stdio.h>
```

заставляет препроцессор вставлять содержимое `<stdio.h>` в файл исходного кода по адресу оператора `#include` перед его компиляцией. `#include` почти всегда используется

для включения файлов заголовков, которые являются файлами, которые в основном содержат объявления функций и инструкции `#define` . В этом случае мы используем `#include` , чтобы иметь возможность использовать такие функции, как `printf` и `scanf` , чьи объявления находятся в файле `stdio.h` . Компиляторы C не позволяют использовать функцию, если она ранее не была объявлена или не определена в этом файле; Операторы `#include` , таким образом, являются способом повторного использования ранее написанного кода в ваших программах на C.

3. **Логические операции** :

```
#if defined A || defined B
variable = another_variable + 1;
#else
variable = another_variable * 2;
#endif
```

будет изменено на:

```
variable = another_variable + 1;
```

если A или B были определены где-то в проекте раньше. Если это не так, конечно, препроцессор сделает это:

```
variable = another_variable * 2;
```

Это часто используется для кода, который работает на разных системах или компилируется на разных компиляторах. Поскольку существуют глобальные определения, которые являются специфическими для компилятора / системы, вы можете протестировать эти определения и всегда позволять компилятору просто использовать код, который он будет компилировать наверняка.

4. **Комментарии**

Препроцессор заменяет все комментарии в исходном файле на отдельные пробелы. Комментарии обозначаются `//` до конца строки или комбинация открывания `/*` и закрытия `*/` комментариев.

Компилятор

После того, как предварительный процессор C включил все файлы заголовков и расширил все макросы, компилятор может скомпилировать программу. Он делает это, превращая исходный код C в файл объектного кода, который является файлом, заканчивающимся на `.o` который содержит двоичную версию исходного кода. Тем не менее, код объекта не является исполняемым. Чтобы сделать исполняемый файл, вам также нужно добавить код для всех функций библиотеки, которые были `#include` d в файл (это не то же самое, что и включение объявлений, что и означает `#include`). Это работа линкера .

В общем, точная последовательность вызова C-компилятора во многом зависит от используемой вами системы. Здесь мы используем компилятор GCC, хотя следует отметить, что существует еще много компиляторов:

```
% gcc -Wall -c foo.c
```

`%` - командная строка ОС. Это говорит компилятору запустить предварительный процессор в файле `foo.c` а затем скомпилировать его в файл объектного кода `foo.o` Параметр `-c` означает компиляцию файла исходного кода в объектный файл, но не для вызова компоновщика. Эта опция `-c` доступна в системах POSIX, таких как Linux или macOS; другие системы могут использовать различный синтаксис.

Если вся ваша программа находится в одном исходном коде, вы можете сделать это:

```
% gcc -Wall foo.c -o foo
```

Это говорит компилятору, чтобы запустить предварительный процессор на `foo.c` , скомпилировать его, а затем связать его, чтобы создать исполняемый файл с именем `foo` . Опция `-o` указывает, что следующее слово в строке - это имя исполняемого бинарного файла (программы). Если вы не укажете `-o` , (если вы просто `gcc foo.c`), исполняемый файл будет называться `a.out` по историческим причинам.

В общем случае компилятор выполняет четыре шага при преобразовании файла `.c` в исполняемый файл:

1. **pre-processing** - текстовое расширение директив `#include` и макросов `#define` в вашем `.c` файле
2. **compilation** - преобразует программу в сборку (вы можете остановить компилятор на этом шаге, добавив параметр `-S`)
3. **сборка** - преобразует сборку в машинный код
4. **linkage** - связывает объектный код с внешними библиотеками для создания исполняемого файла

Также обратите внимание, что имя используемого компилятора - это GCC, что означает «компилятор GNU C» и «сборник компилятора GNU», в зависимости от контекста. Существуют другие компиляторы C. Для Unix-подобных операционных систем многие из них имеют имя `cc` для «компилятора C», что часто является символической ссылкой на какой-либо другой компилятор. В системах Linux `cc` часто является псевдонимом для GCC. На macOS или OS-X он указывает на clang.

Стандарты POSIX в настоящее время определяют `c99` как имя компилятора C - он по умолчанию поддерживает стандарт C99. Более ранние версии POSIX были `c89` как `c89` компилятором. POSIX также требует, чтобы этот компилятор понимал опции `-c` и `-o` которые мы использовали выше.

Примечание. Параметр `-Wall` присутствующий в обоих примерах `gcc` указывает компилятору печатать предупреждения о сомнительных конструкциях, что настоятельно рекомендуется. Также неплохо добавить другие варианты предупреждений , например `-Wextra` .

Фазы перевода

В соответствии со стандартом C 2011, перечисленным в *разделе 5.1.1.2 «Фазы перевода»* , перевод исходного кода на программный образ (например, исполняемый файл) перечислены в 8 упорядоченных шагах.

1. Ввод исходного файла сопоставляется с исходным набором символов (при необходимости). На этом шаге заменены триграфы.
2. Строки продолжения (строки, заканчивающиеся на `\`) соединяются следующей строкой.
3. Исходный код анализируется на пробельные и препроцессорные маркеры.
4. Препроцессор применяется, который выполняет директивы, расширяет макросы и применяет прагмы. Каждый исходный файл, набранный с помощью `#include` претерпевает фазы перевода с 1 по 4 (при необходимости рекурсивно). Затем все директивы, связанные с препроцессором, удаляются.
5. Значения значений исходного символа в символьных константах и строковых литералах сопоставляются с набором символов выполнения.
6. Строковые литералы, смежные друг с другом, объединяются.
7. Исходный код анализируется на токены, которые составляют блок перевода.
8. Внешние ссылки разрешаются, и формируется изображение программы.

Реализация компилятора C может объединять несколько шагов вместе, но результирующее изображение должно вести себя так, как если бы вышеупомянутые шаги выполнялись отдельно в порядке, указанном выше.

Прочитайте компиляция онлайн: https://riptutorial.com/ru/c/topic/1337/компиляция

глава 20: Литералы для чисел, символов и строк

замечания

Термин « **литерал**» обычно используется для описания последовательности символов в C-коде, который обозначает постоянное значение, например число (например, `0`) или строку (например, `"C"`). Строго говоря, стандарт использует термин **константа** для целочисленных констант, плавающих констант, констант перечисления и символьных констант, резервируя термин «литерал» для строковых литералов, но это не является обычным использованием.

Литералы могут иметь **префиксы** или **суффиксы** (но не оба), которые являются дополнительными символами, которые могут начинать или заканчивать литерал, чтобы изменить его тип по умолчанию или его представление.

Examples

Целочисленные литералы

Целочисленные литералы используются для обеспечения интегральных значений. Поддерживаются три числовые базы, обозначенные префиксами:

База	Префикс	пример
Десятичный	Никто	`5`
восьмеричный	`0`	`0345`
шестнадцатеричный	`0x` или `0X`	`0x12AB` , `0X12AB` , `0x12ab` , `0x12Ab`

Обратите внимание, что это письмо не содержит никаких признаков, поэтому целые литералы всегда положительны. Кое-что вроде `-1` рассматривается как выражение, которое имеет один целочисленный литерал (`1`), который отрицается с помощью а `-`

Тип десятичного целочисленного литерала - это первый тип данных, который может соответствовать значению от `int` и `long` . Поскольку C99, `long long` также поддерживается для очень больших литералов.

Тип восьмеричного или шестнадцатеричного целочисленного литерала - это первый тип данных, который может соответствовать значению от `int` , `unsigned` , `long` и `unsigned long` . Поскольку C99, `long long` и `unsigned long long` также поддерживаются для очень больших литералов.

Используя различные суффиксы, можно изменить тип литерала по умолчанию.

Суффикс	объяснение
`L` , `l`	`long int`
`LL` , `ll` (начиная с C99)	`long long int`
`U` , `u`	`unsigned`

Суффикс U и L / LL можно комбинировать в любом порядке и в любом случае. Ошибка дублирования суффиксов (например, предоставление двух `U` суффиксов), даже если они имеют разные случаи.

Строковые литералы

Строковые литералы используются для указания массивов символов. Это последовательности символов, заключенные в двойные кавычки (например, `"abcd"` и имеющие тип `char*`).

Префикс `L` делает литерал широким массивом символов типа `wchar_t*` . Например, `L"abcd"` .

Начиная с C11, существуют и другие префиксы кодирования, аналогичные `L` :

префикс	базовый тип	кодирование
никто	`char`	зависит от платформы
`L`	`wchar_t`	зависит от платформы
`u8`	`char`	UTF-8,
`u`	`char16_t`	обычно UTF-16
`U`	`char32_t`	обычно UTF-32

Для последних двух он может быть запрошен с помощью макросов функций, если кодирование является фактически соответствующей кодировкой UTF.

Литералы с плавающей точкой

Литералы с плавающей точкой используются для представления подписанных действительных чисел. Следующие суффиксы могут использоваться для указания типа литерала:

Суффикс	Тип	Примеры
никто	double	3.1415926 -3E6
f , F	float	3.1415926f 2.1E-6F
l , L	long double	3.1415926L 1E126L

Чтобы использовать эти суффиксы, литерал *должен* быть литералом с плавающей запятой. Например, `3f` является ошибкой, так как `3` является целым литералом, а `3.f` или `3.0f` являются правильными. Для `long double` рекомендуется всегда использовать капитал `L` для удобства чтения.

Литералы символов

Символьные литералы - это особый тип целочисленных литералов, которые используются для обозначения одного символа. Они заключены в одинарные кавычки, например `'a'` и имеют тип `int` . Значение литерала представляет собой целочисленное значение в соответствии с набором символов машины. Они не допускают суффиксов.

Префикс `L` перед символьным литералом делает его широким символом типа `wchar_t` . Аналогично, поскольку префиксы C11 `u` и `U` делают его широкими символами типа `char16_t` и `char32_t` , соответственно.

При намеревании представлять определенные специальные символы, такие как непечатаемый символ, используются escape-последовательности. Последовательности Escape используют последовательность символов, которые переводятся на другой символ. Все escape-последовательности состоят из двух или более символов, первая из которых - обратная косая черта `\` . Символы, следующие за обратной косой чертой, определяют, какой символ буквально интерпретируется как последовательность.

Escape Sequence	Представленный персонаж
\b	возврат на одну позицию
\f	Подача формы
\n	Линейный канал (новая строка)
\r	Возврат каретки
\t	Горизонтальная вкладка
\v	Вертикальная вкладка
\\	бэкслэш

Escape Sequence	Представленный персонаж
`\'`	Одиночная кавычка
`\"`	Двойная кавычка
`\?`	Вопросительный знак
`\nnn`	Октальное значение
`\xnn ...`	Шестнадцатеричное значение

C89

Escape Sequence	Представленный персонаж
`\a`	Предупреждение (звуковой сигнал, звонок)

C99

Escape Sequence	Представленный персонаж
`\unnnn`	Имя универсального символа
`\Unnnnnnnn`	Имя универсального символа

Универсальное имя символа - это кодовая точка Юникода. Имя универсального символа может отображаться более чем на один символ. Цифры `n` интерпретируются как шестнадцатеричные цифры. В зависимости от используемой кодировки UTF универсальная последовательность имен символов может приводить к тому, что кодовая точка состоит из нескольких символов, а не одного обычного `char` символа.

При использовании escape-последовательности линии в текстовом режиме ввода-вывода он преобразуется в байтовую или байтовую последовательность новой строки.

Для избежания триграфов используется escape-последовательность вопросительного знака. Например, `??/` скомпилирован как триграф, представляющий символ обратной косой черты `'\'` , но используя `?\?/` Приведет к *строке* `"??/"` .

В восьмеричной последовательности выхода может быть одна, две или три восьмеричные цифры `n` .

Прочитайте Литералы для чисел, символов и строк онлайн:
https://riptutorial.com/ru/c/topic/3455/литералы-для-чисел--символов-и-строк

глава 21: логический

замечания

Чтобы использовать предопределенный тип `_Bool` и заголовок `<stdbool.h>` , вы должны использовать версии C99 / C11 C.

Чтобы избежать предупреждений компилятора и, возможно, ошибок, вы должны использовать только пример `typedef` / `define` если используете C89 и предыдущие версии языка.

Examples

Использование stdbool.h

C99

Используя файл заголовка системы `stdbool.h` вы можете использовать `bool` как тип данных Boolean. `true` оценивает значение `1` а `false` - `0` .

```
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    bool x = true;  /* equivalent to bool x = 1; */
    bool y = false; /* equivalent to bool y = 0; */
    if (x)  /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

`bool` - просто хорошее написание для типа данных `_Bool` . Он имеет специальные правила, когда числа или указатели преобразуются в него.

Использование #define

С всех версий эффективно обрабатывает любое целочисленное значение, отличное от `0` как `true` для операторов сравнения, а целочисленное значение `0` как `false` . Если у вас нет `_Bool` или `bool` с C99, вы можете имитировать тип данных Boolean на C с помощью макросов `#define` , и вы все равно можете найти такие вещи в устаревшем коде.

```
#include <stdio.h>

#define bool int
#define true 1
#define false 0

int main(void) {
    bool x = true;  /* Equivalent to int x = 1; */
    bool y = false; /* Equivalent to int y = 0; */
    if (x) /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

Не `<stdbool.h>` это в новый код, так как определение этих макросов может столкнуться с современным использованием `<stdbool.h>` .

Использование встроенного (встроенного) типа _Bool

C99

`_Bool` также добавлен в стандартную C-версию C99, а также является родным типом данных C. Он способен удерживать значения `0` (для *false*) и `1` (для *true*).

```
#include <stdio.h>

int main(void) {
    _Bool x = 1;
    _Bool y = 0;
    if(x) /* Equivalent to if (x == 1) */
    {
        puts("This will print!");
    }
    if (!y) /* Equivalent to if (y == 0) */
    {
        puts("This will also print!");
    }
}
```

`_Bool` - целочисленный тип, но имеет специальные правила для конверсий из других типов. Результат аналогичен использованию других типов в выражениях `if` . В следующих

```
_Bool z = X;
```

- Если `X` имеет арифметический тип (любой номер), `z` становится `0` если `X == 0` . В противном случае `z` становится равным `1` .
- Если `X` имеет тип указателя, `z` становится `0` если `X` является нулевым указателем и `1` противном случае.

Чтобы использовать более приятные варианты написания `bool` , `false` и `true` вам нужно использовать `<stdbool.h>` .

Целые и указатели в булевых выражениях.

Все целые числа или указатели могут использоваться в выражении, которое интерпретируется как «значение истины».

```
int main(int argc, char* argv[]) {
  if (argc % 4) {
    puts("arguments number is not divisible by 4");
  } else {
    puts("argument number is divisible by 4");
  }
...
```

Выражение `argc % 4` оценивается и приводит к одному из значений `0` , `1` , `2` или `3` . Первое, `0` - единственное значение, которое является «ложным» и приносит исполнение в часть `else` . Все остальные значения являются «истинными» и переходят в часть `if` .

```
double* A = malloc(n*sizeof *A);
if (!A) {
   perror("allocation problems");
   exit(EXIT_FAILURE);
}
```

Здесь указатель `A` оценивается и, если он является нулевым указателем, обнаруживается ошибка и программа завершается.

Многие люди предпочитают писать что-то как `A == NULL` , но если у вас есть такие сравнения указателей как часть других сложных выражений, вещи становятся очень трудночитаемыми.

```
char const* s = ....;   /* some pointer that we receive */
if (s != NULL && s[0] != '\0' && isalpha(s[0])) {
   printf("this starts well, %c is alphabetic\n", s[0]);
}
```

Для этого вам нужно будет сканировать сложный код в выражении и быть уверенным в предпочтении оператора.

```
char const* s = ....;   /* some pointer that we receive */
if (s && s[0] && isalpha(s[0])) {
   printf("this starts well, %c is alphabetic\n", s[0]);
}
```

относительно легко захватить: если указатель действителен, мы проверяем, отличен ли первый символ от нуля, а затем проверьте, является ли это буквой.

Определение типа bool с использованием typedef

Учитывая, что большинство отладчиков не знают макросов `#define` , но могут проверять константы `enum` , может быть желательно сделать что-то вроде этого:

```
#if __STDC_VERSION__ < 199900L
typedef enum { false, true } bool;
/* Modern C code might expect these to be macros. */
# ifndef bool
#  define bool bool
# endif
# ifndef true
#  define true true
# endif
# ifndef false
#  define false false
# endif
#else
# include <stdbool.h>
#endif

/* Somewhere later in the code ... */
bool b = true;
```

Это позволяет компиляторам для исторических версий C функционировать, но остается передовым, если код скомпилирован с использованием современного компилятора C.

Для получения дополнительной информации о `typedef` см. Typedef , для получения дополнительной информации о `enum` см. Перечисления

Прочитайте логический онлайн: https://riptutorial.com/ru/c/topic/3336/логический

глава 22: Массивы

Вступление

Массивы представляют собой производные типы данных, представляющие упорядоченный набор значений («элементов») другого типа. Большинство массивов в C имеют фиксированное количество элементов любого одного типа, и его представление хранит элементы смежно в памяти без пробелов или отступов. C допускает многомерные массивы, элементами которых являются другие массивы, а также массивы указателей.

C поддерживает динамически распределенные массивы, размер которых определяется во время выполнения. C99 и более поздние версии поддерживают массивы переменной длины или VLA.

Синтаксис

- имя типа [длина]; / * Определить массив «type» с именем «name» и длиной «length». * /
- int arr [10] = {0}; / * Определить массив и инициализировать ВСЕ элементы в 0. * /
- int arr [10] = {42}; / * Определить массив и инициализировать 1-й элемент до 42, а остаток - 0. * /
- int arr [] = {4, 2, 3, 1}; / * Определить и инициализировать массив длиной 4. * /
- arr [n] = значение; / * Установленное значение при индексе n. * /
- значение = arr [n]; / * Получить значение по индексу n. * /

замечания

Зачем нам нужны массивы?

Массивы обеспечивают способ организации объектов в совокупность с его собственным значением. Например, строки C представляют собой массивы символов (`char` s) и строку, такую как «Hello, World!». имеет значение как совокупность, которая не присуща персонажам индивидуально. Аналогично, массивы обычно используются для представления математических векторов и матриц, а также списков многих видов. Более того, без какого-либо элемента для группировки элементов нужно будет решать каждый отдельно, например, через отдельные переменные. Мало того, что это громоздко, он не легко вмещает коллекции разной длины.

Массивы неявно преобразуются в указатели в большинстве контекстов .

За исключением случаев, когда он является операндом оператора `sizeof` оператором `_Alignof` (C2011) или оператором unary `&` (address-of) или как строковый литерал, используемый для инициализации (другого) массива, массив неявно преобразуется в («

decays to») указатель на свой первый элемент. Это неявное преобразование тесно связано с определением оператора субтипирования массива (`[]`): выражение `arr[idx]` определяется как эквивалентное `*(arr + idx)` . Кроме того, поскольку арифметика указателя коммутативна, `*(arr + idx)` также эквивалентна `*(idx + arr)` , что, в свою очередь, эквивалентно `idx[arr]` . Все эти выражения действительны и оцениваются с одинаковым значением при условии, что либо `idx` либо `arr` является указателем (или массивом, который распадается на указатель), а другой является целым числом, а целое число является допустимым индексом в массив на который указывает указатель.

В качестве частного случая заметим, что `&(arr[0])` эквивалентно `&*(arr + 0)` , что упрощается до `arr` . Все эти выражения взаимозаменяемы везде, где последний разпад указателя. Это просто снова выражает, что массив распадается на указатель на его первый элемент.

Напротив, если адрес-оператор применяется к массиву типа `T[N]` (*т.е.* `&arr`), тогда результат имеет тип `T (*)[N]` и указывает на весь массив. Это отличается от указателя на первый элемент массива, по крайней мере, относительно арифметики указателя, которая определяется в терминах размера заостренного типа.

Функциональные параметры не являются массивами .

```
void foo(int a[], int n);
void foo(int *a, int n);
```

Хотя первое объявление `foo` использует синтаксис типа массива для параметра `a` , такой синтаксис используется для объявления параметра функции, объявляющего этот параметр как *указатель* на тип элемента массива. Таким образом, вторая сигнатура для `foo()` семантически идентична первой. Это соответствует распаду значений массива указателям, где они отображаются в качестве аргументов для *вызова* функции, так что если переменная и параметр функции объявлены с тем же типом массива, то значение этой переменной подходит для использования в вызове функции как аргумент, связанный с параметром.

Examples

Объявление и инициализация массива

Общий синтаксис объявления одномерного массива

```
type arrName[size];
```

где `type` может быть любым встроенным типом или определяемыми пользователем типами, такими как структуры, `arrName` является определяемым пользователем идентификатором, а `size` является целочисленной константой.

Объявление массива (массив из 10 переменных int в этом случае) выполняется следующим образом:

```
int array[10];
```

теперь он имеет неопределенные значения. Чтобы гарантировать, что при объявлении он имеет нулевые значения, вы можете сделать это:

```
int array[10] = {0};
```

В массивах также могут быть инициализаторы, в этом примере объявляется массив из 10 `int` , где первые 3 `int` будут содержать значения `1` , `2` , `3` , все остальные значения будут равны нулю:

```
int array[10] = {1, 2, 3};
```

В приведенном выше методе инициализации первое значение в списке будет присвоено первому члену массива, второе значение будет присвоено второму элементу массива и так далее. Если размер списка меньше размера массива, то, как и в предыдущем примере, остальные члены массива будут инициализированы нулями. С назначенной инициализацией списка (ISO C99) возможна явная инициализация элементов массива. Например,

```
int array[5] = {[2] = 5, [1] = 2, [4] = 9}; /* array is {0, 2, 5, 0, 9} */
```

В большинстве случаев компилятор может вывести длину массива для вас, этого можно добиться, оставив квадратные скобки пустыми:

```
int array[] = {1, 2, 3}; /* an array of 3 int's */
int array[] = {[3] = 8, [0] = 9}; /* size is 4 */
```

Объявление массива нулевой длины недопустимо.

C99 C11

В C99 были добавлены массивы переменной длины (VLA для краткости) и были добавлены в C11. Они равны нормальным массивам, с одной, важной, разницей: длина не обязательно должна быть известна во время компиляции. У VLA есть время автоматического хранения. Только указатели на VLA могут иметь статическую продолжительность хранения.

```
size_t m = calc_length(); /* calculate array length at runtime */
int vla[m];               /* create array with calculated length */
```

Важный:

VLA потенциально опасны. Если для массива `vla` в приведенном выше примере требуется

больше места в стеке, чем доступно, стек будет переполняться. Поэтому использование VLA часто не поощряется в руководствах по стилю, а также книгами и упражнениями.

Очистка содержимого массива (обнуление)

Иногда необходимо установить массив в ноль после завершения инициализации.

```
#include <stdlib.h> /* for EXIT_SUCCESS */

#define ARRLEN (10)

int main(void)
{
  int array[ARRLEN]; /* Allocated but not initialised, as not defined static or global. */

  size_t i;
  for(i = 0; i < ARRLEN; ++i)
  {
    array[i] = 0;
  }

  return EXIT_SUCCESS;
}
```

Общим сокращением к вышеуказанному циклу является использование `memset()` из `<string.h>` . Проходящий `array` как показано ниже, заставляет его распадаться на указатель на его 1-й элемент.

```
memset(array, 0, ARRLEN * sizeof (int)); /* Use size explicitly provided type (int here). */
```

или же

```
memset(array, 0, ARRLEN * sizeof *array); /* Use size of type the pointer is pointing to. */
```

Как и в этом примере, `array` *представляет* собой массив, а не только указатель на 1-й элемент массива (см. Длину массива на том, почему это важно), возможно третий вариант для вывода из массива:

```
 memset(array, 0, sizeof array); /* Use size of the array itself. */
```

Длина массива

Массивы имеют фиксированные длины, которые известны в рамках их деклараций. Тем не менее, возможно и иногда удобно рассчитать длину массива. В частности, это может сделать код более гибким, когда длина массива определяется автоматически из инициализатора:

```
int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
/* size of `array` in bytes */
size_t size = sizeof(array);

/* number of elements in `array` */
size_t length = sizeof(array) / sizeof(array[0]);
```

Однако в большинстве контекстов, где массив появляется в выражении, он автоматически преобразуется в указатель («decays to») на свой первый элемент. Случай, когда массив является операндом оператора `sizeof` является одним из небольшого числа исключений. Результирующий указатель сам по себе не является массивом, и он не несет никакой информации о длине массива, из которого он был получен. Поэтому, если эта длина необходима в сочетании с указателем, например, когда указатель передается функции, он должен передаваться отдельно.

Например, предположим, что мы хотим написать функцию для возврата последнего элемента массива из `int` . Продолжая вышеизложенное, мы можем назвать это так:

```
/* array will decay to a pointer, so the length must be passed separately */
int last = get_last(array, length);
```

Функция может быть реализована следующим образом:

```
int get_last(int input[], size_t length) {
    return input[length - 1];
}
```

Обратите внимание, в частности, что хотя объявление `input` параметра похоже на объявление массива, **оно фактически объявляет `input` как указатель** (для `int`). Это точно эквивалентно объявлению `input` как `int *input` . То же самое было бы верно, даже если бы было дано измерение. Это возможно, потому что массивы никогда не могут быть фактическими аргументами для функций (они распадаются на указатели, когда они появляются в выражениях вызова функций), и их можно рассматривать как мнемонические.

Это очень распространенная ошибка, чтобы попытаться определить размер массива из указателя, который не может работать. НЕ ДЕЛАЙТЕ ЭТОГО:

```
int BAD_get_last(int input[]) {
    /* INCORRECTLY COMPUTES THE LENGTH OF THE ARRAY INTO WHICH input POINTS: */
    size_t length = sizeof(input) / sizeof(input[0]));

    return input[length - 1];  /* Oops -- not the droid we are looking for */
}
```

На самом деле эта конкретная ошибка настолько распространена, что некоторые компиляторы ее распознают и предупреждают об этом. `clang` , например, выдает следующее предупреждение:

```
warning: sizeof on array function parameter will return size of 'int *' instead of 'int []' [-
Wsizeof-array-argument]
        int length = sizeof(input) / sizeof(input[0]);
                             ^
note: declared here
int BAD_get_last(int input[])
                     ^
```

Установка значений в массивах

Доступ к значениям массива обычно выполняется с помощью квадратных скобок:

```
int val;
int array[10];

/* Setting the value of the fifth element to 5: */
array[4] = 5;

/* The above is equal to: */
*(array + 4) = 5;

/* Reading the value of the fifth element: */
val = array[4];
```

В качестве побочного эффекта операндов к оператору `+` заменить (-> коммутативный закон) следующее эквивалентно:

```
*(array + 4) = 5;
*(4 + array) = 5;
```

так что следующие утверждения эквивалентны:

```
array[4] = 5;
4[array] = 5; /* Weird but valid C ... */
```

и эти два:

```
val = array[4];
val = 4[array]; /* Weird but valid C ... */
```

C не выполняет никаких пограничных проверок, доступ к содержимому вне объявленного массива не определен (доступ к памяти за пределами выделенного фрагмента):

```
int val;
int array[10];

array[4] = 5;    /* ok */
val = array[4];  /* ok */
array[19] = 20;  /* undefined behavior */
val = array[15]; /* undefined behavior */
```

Определить массив и элемент массива доступа

```
#include <stdio.h>

#define ARRLEN (10)

int main (void)
{

   int n[ ARRLEN ]; /* n is an array of 10 integers */
   size_t i, j; /* Use size_t to address memory, that is to index arrays, as its guaranteed to

                   be wide enough to address all of the possible available memory.
                   Using signed integers to do so should be considered a special use case,
                   and should be restricted to the uncommon case of being in the need of
                   negative indexes. */

   /* Initialize elements of array n. */
   for ( i = 0; i < ARRLEN ; i++ )
   {
      n[ i ] = i + 100; /* Set element at location i to i + 100. */
   }

   /* Output each array element's value. */
   for (j = 0; j < ARRLEN ; j++ )
   {
      printf("Element[%zu] = %d\n", j, n[j] );
   }

   return 0;
}
```

Выделить и нуль инициализировать массив с заданным пользователем размером

```
#include <stdio.h>
#include <stdlib.h>


int main (void)
{
  int * pdata;
  size_t n;

  printf ("Enter the size of the array: ");
  fflush(stdout); /* Make sure the prompt gets printed to buffered stdout. */

  if (1 != scanf("%zu", &n)) /* If zu is not supported (Windows?) use lu. */
  {
    fprintf("scanf() did not read a in proper value.\n");
    exit(EXIT_FAILURE);
  }

  pdata = calloc(n, sizeof *pdata);
  if (NULL == pdata)
  {
    perror("calloc() failed"); /* Print error. */
```

```
    exit(EXIT_FAILURE);
  }

  free(pdata); /* Clean up. */

  return EXIT_SUCCESS;
}
```

Эта программа пытается сканировать в значении без знака со стандартного ввода, выделяет блок памяти для массива из `n` элементов типа `int` , вызывая функцию `calloc()` . Память инициализируется всеми нулями последней.

В случае успеха память освобождается вызовом `free()` .

Итерация через массив эффективно и порядок строк

Массивы в C можно рассматривать как непрерывный кусок памяти. Точнее, последнее измерение массива - это смежная часть. Мы называем это *строковым порядком* . Понимая это и тот факт, что ошибка кэша загружает полную кеш-строку в кеш при доступе к нераскрытым данным, чтобы предотвратить последующие ошибки кэша, мы видим, почему доступ к массиву размера 10000x10000 с `array[0][0]` **потенциально может быть** загружен в `array[0][1]` в кеше, но доступ к `array[1][0]` сразу же сгенерировал бы вторую ошибку кэша, так как это `sizeof(type)*10000` bytes from `array[0][0]` , и, следовательно, в той же строке кэша. Вот почему итерация таким образом неэффективна:

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[j][i] = 0;
    }
}
```

Итерация таким образом более эффективна:

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[i][j] = 0;
    }
}
```

В том же духе, поэтому при работе с массивом с одним измерением и несколькими

индексами (скажем, 2 измерения здесь для простоты с индексами i и j) важно выполнить итерацию по массиву следующим образом:

```
#define DIM_X 10
#define DIM_Y 20

int array[DIM_X*DIM_Y];

size_t i, j;
for (i = 0; i < DIM_X; ++i)
{
    for(j = 0; j < DIM_Y; ++j)
    {
        array[i*DIM_Y+j] = 0;
    }
}
```

Или с 3 измерениями и индексами i, j и k:

```
#define DIM_X 10
#define DIM_Y 20
#define DIM_Z 30

int array[DIM_X*DIM_Y*DIM_Z];

size_t i, j, k;
for (i = 0; i < DIM_X; ++i)
{
    for(j = 0; j < DIM_Y; ++j)
    {
        for (k = 0; k < DIM_Z; ++k)
        {
            array[i*DIM_Y*DIM_Z+j*DIM_Z+k] = 0;
        }
    }
}
```

Или более общим образом, когда у нас есть массив с элементами **N1 x N2 x ... x Nd** , **d** измерениями и индексами, отмеченными как **n1, n2, ..., nd,** смещение рассчитывается так

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots + N_2 n_1) \cdots)) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k$$

Изображение / формула взято из: https://en.wikipedia.org/wiki/Row-major_order

Многомерные массивы

Язык программирования C позволяет использовать многомерные массивы . Вот общая форма объявления многомерного массива -

```
type name[size1][size2]...[sizeN];
```

Например, следующее объявление создает трехмерный (5 x 10 x 4) целочисленный массив:

```
int arr[5][10][4];
```

Двумерные массивы

Простейшей формой многомерного массива является двумерный массив. Двумерный массив представляет собой, по существу, список одномерных массивов. Чтобы объявить двумерный целочисленный массив размеров mxn, мы можем написать следующее:

```
type arrayName[m][n];
```

Где `type` может быть любым допустимым типом данных C (`int` , `float` и т. Д.), А `arrayName` может быть любым допустимым идентификатором C. Двумерный массив можно визуализировать как таблицу с `m` строками и `n` столбцами. **Примечание** : порядок *имеет* значение в C. Массив `int a[4][3]` не совпадает с массивом `int a[3][4]` . Количество строк приходит сначала в качестве C является *строка* -Майора языка.

Двумерный массив `a` , содержащий три строки и четыре столбца, можно показать следующим образом:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Таким образом, каждый элемент в массиве `a` идентифицируется именем элемента формы `a[i][j]` , где `a` - это имя массива, `i` представляет, какую строку и `j` представляет собой какой столбец. Напомним, что строки и столбцы нулевые индексируются. Это очень похоже на математическое обозначение для подписи двухмерных матриц.

Инициализация двумерных массивов

Многомерные массивы могут быть инициализированы путем задания скобок для каждой строки. Следующие определяют массив с 3 строками, где каждая строка имеет 4 столбца.

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

Вложенные фигурные скобки, которые указывают предполагаемую строку, являются необязательными. Следующая инициализация эквивалентна предыдущему примеру:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Хотя метод создания массивов с вложенными фигурными скобками является необязательным, он настоятельно рекомендуется, поскольку он более читабельным и понятным.

Доступ к двумерным элементам массива

Доступ к элементу в двумерном массиве осуществляется с помощью индексов, то есть индекса строки и индекса столбца массива. Например,

```
int val = a[2][3];
```

Вышеприведенный оператор берет 4-й элемент из 3-й строки массива. Давайте проверим следующую программу, в которой мы использовали вложенный цикл для обработки двумерного массива:

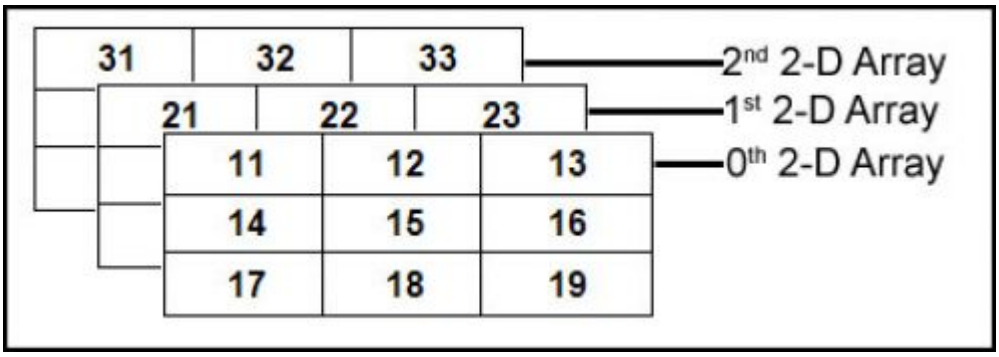
```
#include <stdio.h>

int main () {

   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   int i, j;

   /* output each array element's value */
   for ( i = 0; i < 5; i++ ) {

      for ( j = 0; j < 2; j++ ) {
         printf("a[%d][%d] = %d\n", i,j, a[i][j] );
      }
   }

   return 0;
}
```

Когда приведенный выше код компилируется и выполняется, он производит следующий результат:

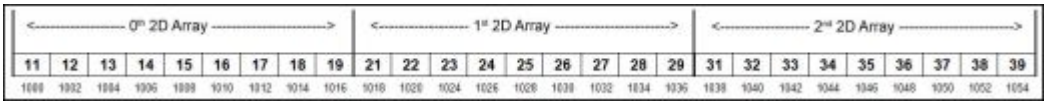
```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

Трехмерный массив:

3D-массив по существу представляет собой массив массивов массивов: это массив или набор 2D-массивов, а 2D-массив - массив из 1-го массива.



Карта памяти 3D-массива:



Инициализация 3D-массива:

```
double cprogram[3][2][4]={
{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},
 {{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
 {{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
};
```

Мы можем иметь массивы с любым количеством измерений, хотя вполне вероятно, что большинство создаваемых массивов будут иметь один или два измерения.

Итерация через массив с помощью указателей

```
#include <stdio.h>
#define SIZE (10)
int main()
{
    size_t i = 0;
    int *p = NULL;
    int a[SIZE];

    /* Setting up the values to be i*i */
    for(i = 0; i < SIZE; ++i)
    {
        a[i] = i * i;
    }

    /* Reading the values using pointers */
    for(p = a; p < a + SIZE; ++p)
    {
        printf("%d\n", *p);
    }

    return 0;
}
```

Здесь, при инициализации `p` в первом `for` цикла условиях, массив `a` *распадается* на указатель на его первый элемент, как и во всех местах, где используется такая переменная массива.

Затем `++p` выполняет арифметику указателя на указателе `p` и идет один за другим через элементы массива и ссылается на них путем разыменования их с помощью `*p` .

Передача многомерных массивов в функцию

Многомерные массивы следуют тем же правилам, что и одномерные массивы при передаче их функции. Однако комбинация распада на указатель, приоритет оператора и два разных способа объявления многомерного массива (массив массивов против массива указателей) могут сделать объявление таких функций неинтуитивным. В следующем примере показаны правильные способы передачи многомерных массивов.

```
#include <assert.h>
#include <stdlib.h>

/* When passing a multidimensional array (i.e. an array of arrays) to a
   function, it decays into a pointer to the first element as usual.  But only
   the top level decays, so what is passed is a pointer to an array of some fixed
   size (4 in this case). */
void f(int x[][4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* This prototype is equivalent to f(int x[][4]).
   The parentheses around *x are required because [index] has a higher
   precedence than *expr, thus int *x[4] would normally be equivalent to int
   *(x[4]), i.e. an array of 4 pointers to int.  But if it's declared as a
   function parameter, it decays into a pointer and becomes int **x,
   which is not compatable with x[2][4]. */
void g(int (*x)[4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* An array of pointers may be passed to this, since it'll decay into a pointer
   to pointer, but an array of arrays may not. */
void h(int **x) {
    assert(sizeof(*x) == sizeof(int*));
}

int main(void) {
    int foo[2][4];
    f(foo);
    g(foo);

    /* Here we're dynamically creating an array of pointers.  Note that the
       size of each dimension is not part of the datatype, and so the type
       system just treats it as a pointer to pointer, not a pointer to array
       or array of arrays. */
    int **bar = malloc(sizeof(*bar) * 2);
    assert(bar);
    for (size_t i = 0; i < 2; i++) {
        bar[i] = malloc(sizeof(*bar[i]) * 4);
        assert(bar[i]);
```

```
    }

    h(bar);

    for (size_t i = 0; i < 2; i++) {
        free(bar[i]);
    }
    free(bar);
}
```

Смотрите также

Передача массивов в функции

Прочитайте Массивы онлайн: https://riptutorial.com/ru/c/topic/322/массивы

глава 23: Многопоточность

Вступление

В C11 существует стандартная библиотека потоков, `<threads.h>` , но не известный компилятор, который ее реализует. Таким образом, чтобы использовать многопоточность в C, вы должны использовать специфичные для платформы реализации, такие как библиотека потоков POSIX (часто называемая pthreads), используя заголовок `pthread.h` .

Синтаксис

- thrd_t // Определенный реализацией полный тип объекта, определяющий поток
- int thrd_create (thrd_t * thr, thrd_start_t func, void * arg); // Создает поток
- int thrd_equal (thrd_t thr0, thrd_t thr1); // Проверяем, ссылаются ли аргументы на тот же поток
- thr_t thrd_current (void); // Возвращает идентификатор потока, который его вызывает
- int thrd_sleep (const struct timespec * duration, struct timespec * остается); // Приостановить выполнение потока вызовов в течение как минимум заданного времени
- void thrd_yield (void); // Разрешить запуск других потоков вместо потока, который его вызывает
- _Noreturn void thrd_exit (int res); // Завершает поток поток, который его вызывает
- int thrd_detatch (thrd_t thr; // Отделяет данный поток от текущей среды
- int thrd_join (thrd_t thr, int * res); // Блокирует текущий поток до тех пор, пока данный поток не завершится

замечания

Использование потоков может привести к дополнительным неопределенным поведением, таким как http://www.riptutorial.com/c/example/2622/data-race . Для расы , свободной от доступа к переменным , которые являются общими между различными потоками C11 обеспечивает `mtx_lock()` функциональность мьютекса или (опционально) http://www.riptutorial.com/c/topic/4924/atomics типов данных и связанных с ними функций в `stdatomic.h` .

Examples

Простой пример C11 Threads

```
#include <threads.h>
#include <stdio.h>
```

```
int run(void *arg)
{
    printf("Hello world of C11 threads.");

    return 0;
}

int main(int argc, const char *argv[])
{
    thrd_t thread;
    int result;

    thrd_create(&thread, run, NULL);

    thrd_join(&thread, &result);

    printf("Thread return %d at the end\n", result);
}
```

Прочитайте Многопоточность онлайн: https://riptutorial.com/ru/c/topic/10489/многопоточность

глава 24: Многосимвольная последовательность символов

замечания

Не все препроцессоры поддерживают обработку последовательности триграмм. Некоторые компиляторы предоставляют дополнительную опцию или переключатель для их обработки. Другие используют отдельную программу для преобразования триграфов.

Компилятор GCC не распознает их, если вы явно не попросите его сделать это (используйте `-trigraphs` чтобы включить их, используйте `-Wtrigraphs` , часть `-Wall` , чтобы получать предупреждения о триграфах).

Поскольку большинство используемых сегодня платформ поддерживают весь спектр одиночных символов, используемых в C, орграфы предпочтительнее, чем триграфы, но использование любых многосимвольных последовательностей символов обычно не рекомендуется.

Кроме того, остерегайтесь случайного использования триграфа (`puts("What happened??!!");` , например).

Examples

триграфы

Символы `[ ] { } ^ \ | ~ #` Часто используются в программах C, но в конце 1980 - х годов, были кодовые наборы в использовании (ISO 646 вариантов, например, в скандинавских странах) , где позиции ASCII символов для них были использованы для национального языка вариантных символов (например , `£` для `#` в Великобритании; `Æ Å æ å ø Ø` для `{ } { } | \` в Дании не было `~` в EBCDIC). Это означало, что было сложно написать код на машинах, которые использовали эти наборы.

Чтобы решить эту проблему, в стандарте C было предложено использовать комбинации из трех символов для создания одного символа, называемого триграфом. Триграф представляет собой последовательность из трех символов, первые две из которых являются вопросительными знаками.

Ниже приведен простой пример, который использует последовательности триграмм вместо `#` , `{` и `}` :

```
??=include <stdio.h>
```

```
int main()
??<
    printf("Hello World!\n");
??>
```

Это будет изменено препроцессором C, заменив триграфы их односимвольными эквивалентами, как если бы код был написан:

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
}
```

триграф	эквивалент
знак равно	#
?? /	\
??»	^
?? (	[
??)	]
??!	|
?? <	{
??>	}
?? -	~

Обратите внимание, что триграфы являются проблематичными, потому что, например `??/` Является обратным слэшем и может влиять на значение строк продолжения в комментариях и должно быть распознано внутри строк и символьных литералов (например, `'??/??/'` - это один символ, обратная косая черта).

орграфах

C99

В 1994 году были предоставлены более читаемые альтернативы пяти триграфам. Они используют только два символа и называются орграфами. В отличие от триграфов, орграфы - это жетоны. Если орграф встречается в другом токене (например, строковые литералы или символьные константы), то он не будет рассматриваться как орграф, но останется таким, каким он есть.

Ниже показано различие до и после обработки орграфа последовательности.

```
#include <stdio.h>

int main()
<%
    printf("Hello %> World!\n"); /* Note that the string contains a digraph */
%>
```

Которые будут обрабатываться так же, как:

```
#include <stdio.h>

int main()
{
    printf("Hello %> World!\n"); /* Note the unchanged digraph within the string. */
}
```

дограф	эквивалент
<:	[
:>	]
<%	{
%>	}
%:	#

Прочитайте Многосимвольная последовательность символов онлайн:
https://riptutorial.com/ru/c/topic/7111/многосимвольная-последовательность-символов

глава 25: Неопределенное поведение

Вступление

В C некоторые выражения дают *неопределенное поведение* . Стандарт явно не определяет, как должен себя вести компилятор, если он сталкивается с таким выражением. В результате компилятор может делать все, что сочтет нужным, и может давать полезные результаты, неожиданные результаты или даже сбой.

Код, который вызывает UB, может работать как предполагалось в конкретной системе с конкретным компилятором, но, скорее всего, не будет работать в другой системе или с другим компилятором, версией компилятора или компилятором.

замечания

Что такое Undefined Behavior (UB)?

Неопределенное поведение - это термин, используемый в стандарте C. Стандарт C11 (ISO / IEC 9899: 2011) определяет термин неопределенное поведение как

> поведение при использовании непереносимой или ошибочной программной конструкции или ошибочных данных, для которых настоящий международный стандарт не налагает никаких требований

Что произойдет, если в моем коде есть UB?

Это результаты, которые могут произойти из-за неопределенного поведения в соответствии со стандартом:

> ПРИМЕЧАНИЕ. Возможное неопределенное поведение варьируется от полного игнорирования ситуации с непредсказуемыми результатами, ведения во время перевода или выполнения программы документированным образом, характерным для среды (с выдачей диагностического сообщения или без него), до прекращения перевода или выполнения (с выдача диагностического сообщения).

Следующая цитата часто используется для описания (менее формально, хотя) результатов, происходящих от неопределенного поведения:

> «Когда компилятор сталкивается с [определенной неопределенной конструкцией], это законно для того, чтобы заставить демонов вылететь из вашего носа» (подразумевается, что компилятор может выбрать любой произвольно причудливый способ интерпретации кода без нарушения стандарта ANSI C)

Почему существует UB?

Если это так плохо, почему они просто не определяли его или не определяли его реализацию?

Неопределенное поведение позволяет больше возможностей для оптимизации; Компилятор может с полным основанием предположить, что любой код не содержит неопределенного поведения, что может позволить ему избежать проверок во время выполнения и выполнять оптимизации, срок действия которых был бы дорогостоящим или невозможным доказать иначе.

Почему UB трудно отслеживать?

Существует по крайней мере две причины, по которым неопределенное поведение создает ошибки, которые трудно обнаружить:

- Компилятор не обязан - и, как правило, не может надежно - предупреждать вас о неопределенном поведении. Фактически требуя, чтобы это сделало это, явилось бы прямо против причины существования неопределенного поведения.
- Непредсказуемые результаты могут не начинаться разворачиваться в точную точку операции, где происходит конструкция, поведение которой не определено; Неопределенное поведение окрашивает все исполнение и его последствия могут произойти в любое время: во время, после или даже *до* неопределенной конструкции.

Рассмотрим разворот нулевого указателя: компилятор не требуется для диагностики разыменования нулевых указателей и даже не мог, так как во время выполнения любой указатель, переданный в функцию, или в глобальной переменной может иметь значение NULL. *И когда происходит разыменование нулевого указателя, стандарт не требует, чтобы программа была повреждена.* Скорее, программа может произойти сбой раньше, позже или вообще не сбой; он может даже вести себя так, как если бы нулевой указатель указывал на действительный объект и вел себя нормально, только для того, чтобы сбой при других обстоятельствах.

В случае нулевого указателя разыменования, C язык отличается от управляемых языков , таких как Java или C #, где *определяется* поведение нуль-указатель разыменования: генерируется исключение, в точное время (`NullPointerException` в Java, `NullReferenceException` в C #) , поэтому те, которые приходят с Java или C #, могут *ошибочно полагать, что в этом случае программа C должна сбой, с выдачей диагностического сообщения или без него* .

Дополнительная информация

Существует несколько таких ситуаций, которые следует четко различать:

- Явно неопределенное поведение, то есть где стандарт C явно сообщает вам, что вы не в курсе.
- Неявно неопределенное поведение, когда в стандарте нет текста, который

предвидит поведение для ситуации, в которую вы ввели вашу программу.

Также имейте в виду, что во многих местах поведение определенных конструкций сознательно не определено стандартом C, чтобы оставить место для разработчиков компилятора и библиотек, чтобы придумать свои собственные определения. Хорошим примером являются сигналы и обработчики сигналов, где расширения C, такие как стандарт операционной системы POSIX, определяют гораздо более сложные правила. В таких случаях вам просто нужно проверить документацию своей платформы; стандарт C не может вам ничего сказать.

Также обратите внимание, что если неопределенное поведение происходит в программе, это не означает, что просто точка, где произошло неопределенное поведение, является проблематичной, а целая программа становится бессмысленной.

Из-за таких проблем важно (тем более, что компиляторы не всегда предупреждают нас о UB), чтобы программирование на языке C было, по крайней мере, знакомым с вещами, которые вызывают неопределенное поведение.

Следует отметить, что есть некоторые инструменты (например, инструменты статического анализа, такие как PC-Lint), которые помогают обнаруживать неопределенное поведение, но опять же, они не могут обнаружить все случаи неопределенного поведения.

Examples

Вызов нулевого указателя

Это пример разыменования указателя NULL, вызывающего неопределенное поведение.

```
int * pointer = NULL;
int value = *pointer; /* Dereferencing happens here */
```

Указатель `NULL` гарантируется стандартом C для сравнения неравного с любым указателем на действительный объект, а разыменование его вызывает неопределенное поведение.

Изменение любого объекта более одного раза между двумя точками последовательности

```
int i = 42;
i = i++; /* Assignment changes variable, post-increment as well */
int a = i++ + i--;
```

Подобный код часто приводит к размышлениям о «результирующем значении» `i` . Однако, вместо того, чтобы указывать результат, стандарты C указывают, что оценка такого выражения приводит к *неопределенному поведению* . До C2011 стандарт формализовал эти правила в терминах так называемых *точек последовательности* :

> Между предыдущей и следующей точкой последовательности скалярный объект должен иметь значение, которое его хранимое значение изменялось не более одного раза путем оценки выражения. Кроме того, предыдущее значение должно быть считано только для определения значения, которое необходимо сохранить.

(Стандарт C99, раздел 6.5, пункт 2)

Эта схема оказалась слишком грубой, что привело к появлению некоторых выражений, демонстрирующих неопределенное поведение в отношении C99, которые, вероятно, не должны делать. C2011 сохраняет точки последовательности, но вводит более тонкий подход к этой области на основе *последовательности* и отношений, которые он называет «секвенированными до»:

> Если побочный эффект скалярного объекта не зависит от другого побочного эффекта для одного и того же скалярного объекта или вычисления значения с использованием значения одного и того же скалярного объекта, поведение не определено. Если существует несколько допустимых порядков подвыражений выражения, поведение не определено, если такой побочный эффект без последствий происходит в любом из упорядочений.

(Стандарт C2011, раздел 6.5, пункт 2)

Полные детали отношения «sequenced before» слишком длинны для описания здесь, но они дополняют последовательности, а не вытесняют их, поэтому они влияют на определение поведения для некоторых оценок, поведение которых ранее не было определено. В частности, если есть точка последовательности между двумя оценками, то перед точкой последовательности «секвенируется до» после.

Следующий пример имеет четкое поведение:

```
int i = 42;
i = (i++, i+42); /* The comma-operator creates a sequence point */
```

Следующий пример имеет неопределенное поведение:

```
int i = 42;
printf("%d %d\n", i++, i++); /* commas as separator of function arguments are not comma-
operators */
```

Как и в любой форме неопределенного поведения, наблюдение за фактическим поведением оценки выражений, нарушающих правила секвенирования, не является информативным, кроме как в ретроспективном смысле. Стандарт языка не дает оснований ожидать, что такие наблюдения будут прогностическими даже в отношении будущего поведения той же программы.

Отсутствует оператор возврата в функции возврата значения

```
int foo(void) {
  /* do stuff */
  /* no return here */
}

int main(void) {
  /* Trying to use the (not) returned value causes UB */
  int value = foo();
  return 0;
}
```

Когда функция объявляется для возврата значения, она должна делать это на каждом возможном пути кода через нее. Неопределенное поведение возникает, как только вызывающий (который ожидает возвращаемое значение) пытается использовать возвращаемое значение [1] .

Обратите внимание, что неопределенное поведение происходит *только в том случае, если* вызывающий пытается использовать / получить доступ к значению функции. Например,

```
int foo(void) {
  /* do stuff */
  /* no return here */
}

int main(void) {
  /* The value (not) returned from foo() is unused. So, this program
   * doesn't cause *undefined behaviour*. */
  foo();
  return 0;
}
```

C99

`main()` функция является исключением из этого правила в том , что это возможно для того , чтобы быть прекращено без оператора возврата , потому что предполагается , возвращаемое значение `0` автоматически будет использоваться в данном случае [2].

---

[1] (*ISO / IEC 9899: 201x* , 6.9.1 / 12)

> Если функция}, которая завершает функцию, будет достигнута, а значение вызова функции используется вызывающим, поведение не определено.

[2] (*ISO / IEC 9899: 201x* , 5.1.2.2.3 / 1)

> достигая}, который завершает основную функцию, возвращает значение 0.

Подписанное целочисленное переполнение

В параграфе 6.5 / 5 как C99, так и C11 оценка выражения создает неопределенное поведение, если результат не является представимым значением типа выражения. Для арифметических типов это называется *переполнением* . Беззнаковая целочисленная арифметика не переполняется, поскольку применяется параграф 6.2.5 / 9, в результате чего любой результат без знака, который в противном случае был бы вне диапазона, был бы уменьшен до значения внутри диапазона. Однако не существует аналогичного положения для *знаковых* целочисленных типов; они могут и переполняются, производя неопределенное поведение. Например,

```
#include <limits.h>      /* to get INT_MAX */

int main(void) {
    int i = INT_MAX + 1; /* Overflow happens here */
    return 0;
}
```

Большинство случаев такого типа неопределенного поведения более трудно распознать или предсказать. Переполнение может в принципе возникать из-за любой операции сложения, вычитания или умножения на целые числа со знаком (с учетом обычных арифметических преобразований), где нет эффективных границ или отношений между операндами для предотвращения этого. Например, эта функция:

```
int square(int x) {
    return x * x;  /* overflows for some values of x */
}
```

разумно, и он делает правильные вещи для достаточно малых значений аргументов, но его поведение не определено для больших значений аргументов. Вы не можете судить по этой функции только в том случае, если программы, которые ее называют, демонстрируют неопределенное поведение. Это зависит от того, какие аргументы он передает.

С другой стороны, рассмотрим этот тривиальный пример переполненной цепочки со знаком целочисленной арифметики:

```
int zero(int x) {
    return x - x;  /* Cannot overflow */
}
```

Связь между операндами оператора вычитания гарантирует, что вычитание никогда не переполняется. Или рассмотрим этот несколько более практичный пример:

```
int sizeDelta(FILE *f1, FILE *f2) {
    int count1 = 0;
    int count2 = 0;
    while (fgetc(f1) != EOF) count1++;  /* might overflow */
    while (fgetc(f2) != EOF) count2++;  /* might overflow */

    return count1 - count2; /* provided no UB to this point, will not overflow */
}
```

Пока счетчики не переполняются индивидуально, операнды конечного вычитания будут неотрицательными. Все различия между любыми двумя такими значениями представляются как `int` .

Использование неинициализированной переменной

```
int a;
printf("%d", a);
```

Переменная `a` - это `int` с автоматическим временем хранения. В приведенном выше примере пример пытается напечатать значение неинициализированной переменной (`a` никогда не инициализировалось). Автоматические переменные, которые не инициализированы, имеют неопределенные значения; доступ к ним может привести к неопределенному поведению.

Примечание. Переменные со статическим или потоковым локальным хранилищем, включая глобальные переменные без `static` ключевого слова, инициализируются либо нулем, либо их инициализированным значением. Следовательно, законно.

```
static int b;
printf("%d", b);
```

Очень распространенная ошибка состоит в том, чтобы не инициализировать переменные, которые служат счетчиками равными 0. Вы добавляете к ним значения, но поскольку начальное значение является мусором, вы будете вызывать **Undefined Behavior** , например, в вопросе Компиляция на терминале выдает предупреждение указателя и странные символы .

Пример:

```
#include <stdio.h>

int main(void) {
    int i, counter;
    for(i = 0; i < 10; ++i)
        counter += i;
    printf("%d\n", counter);
    return 0;
}
```

Выход:

```
C02QT2UBFVH6-lm:~ gsamaras$ gcc main.c -Wall -o main
main.c:6:9: warning: variable 'counter' is uninitialized when used here [-Wuninitialized]
        counter += i;
        ^~~~~~~
main.c:4:19: note: initialize the variable 'counter' to silence this warning
    int i, counter;
                  ^
                   = 0
```

```
1 warning generated.
C02QT2UBFVH6-lm:~ gsamaras$ ./main
32812
```

Вышеуказанные правила применимы и для указателей. Например, следующие результаты в неопределенном поведении

```
int main(void)
{
    int *p;
    p++; // Trying to increment an uninitialized pointer.
}
```

Обратите внимание, что приведенный выше код сам по себе может не вызвать ошибку или ошибку сегментации, но попытка разыменования этого указателя позже приведет к неопределенному поведению.

Вывод указателя на переменную за пределами ее срока службы

```
int* foo(int bar)
{
    int baz = 6;
    baz += bar;
    return &baz; /* (&baz) copied to new memory location outside of foo. */
} /* (1) The lifetime of baz and bar end here as they have automatic storage
   * duration (local variables), thus the returned pointer is not valid! */

int main (void)
{
    int* p;

    p = foo(5);  /* (2) this expression's behavior is undefined */
    *p = *p - 6; /* (3) Undefined behaviour here */

    return 0;
}
```

Некоторые компиляторы с благодарностью указывают на это. Например, `gcc` предупреждает:

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

и `clang` предупреждает:

```
warning: address of stack memory associated with local variable 'baz' returned
[-Wreturn-stack-address]
```

для вышеуказанного кода. Но компиляторы, возможно, не смогут помочь в сложном коде.

(1) Возвращаемая ссылка на переменную объявленную `static` определяется поведением, поскольку переменная не уничтожается после выхода из текущей области.

(2) Согласно ISO / IEC 9899: 2011 6.2.4 §2 «Значение указателя становится неопределенным, когда объект, на который он указывает, достигает конца своего срока службы».

(3) Разыменование указателя, возвращаемого функцией `foo` является неопределенным поведением, поскольку память, на которую он ссылается, содержит неопределенное значение.

Деление на ноль

```
int x = 0;
int y = 5 / x;  /* integer division */
```

или же

```
double x = 0.0;
double y = 5.0 / x;  /* floating point division */
```

или же

```
int x = 0;
int y = 5 % x;  /* modulo operation */
```

Для второй строки в каждом примере, где значение второго операнда (x) равно нулю, поведение не определено.

Обратите внимание, что большинство реализаций математики с плавающей запятой будут следовать стандарту (например, IEEE 754), и в этом случае операции, такие как деление на ноль, будут иметь согласованные результаты (например, `INFINITY`), даже если в стандарте C указано, что операция не определена.

Доступ к памяти за пределами выделенного фрагмента

А указатель на кусок памяти, содержащий `n` элементов, может быть разыменован только в том случае, если он находится в `memory` диапазона и `memory + (n - 1)` . Выделение указателя за пределами этого диапазона приводит к неопределенному поведению. В качестве примера рассмотрим следующий код:

```
int array[3];
int *beyond_array = array + 3;
*beyond_array = 0; /* Accesses memory that has not been allocated. */
```

Третья строка обращается к четвертому элементу в массиве, длина которого составляет всего 3 элемента, что приводит к неопределенному поведению. Точно так же поведение второй строки в следующем фрагменте кода также недостаточно четко определено:

```
int array[3];
array[3] = 0;
```

Обратите внимание, что указание последнего элемента массива не является неопределенным поведением (`beyond_array = array + 3` здесь четко определено), но разыменование его (`*beyond_array` - неопределенное поведение). Это правило также выполняется для динамически распределенной памяти (например, буферов, созданных через `malloc`).

Копирование перекрывающейся памяти

Среди их эффектов множество разнообразных стандартных библиотечных функций - копирование последовательностей байтов из одной области памяти в другую. Большинство из этих функций имеют неопределенное поведение, когда области источника и получателя перекрываются.

Например, это ...

```
#include <string.h> /* for memcpy() */

char str[19] = "This is an example";
memcpy(str + 7, str, 10);
```

... пытается скопировать 10 байт, где области памяти источника и назначения перекрываются тремя байтами. Чтобы визуализировать:

```
               overlapping area
               |
               _ _
              |   |
              v   v
T h i s   i s   a n   e x a m p l e \0
^             ^
|             |
|             destination
|
source
```

Из-за перекрытия результирующее поведение не определено.

Среди стандартных библиотечных функций с ограничением такого рода являются `memcpy()` , `strcpy()` , `strcat()` , `sprintf()` и `sscanf()` . В стандарте говорится об этих и некоторых других функциях:

> Если копирование происходит между перекрывающимися объектами, поведение не определено.

Функция `memmove()` является основным исключением из этого правила. Его определение указывает, что функция ведет себя так, как если исходные данные были сначала

скопированы во временный буфер, а затем записаны на целевой адрес. Не существует исключения для перекрытия областей источника и получателя, а также никакой необходимости в одном, поэтому `memmove()` имеет `memmove()` поведение в таких случаях.

Различие отражает эффективность *против* . общий компромисс. Копирование таких функций, как эти функции, обычно происходит между непересекающимися областями памяти, и часто во время разработки можно узнать, будет ли конкретный экземпляр копирования памяти в эту категорию. Предполагая, что не-перекрытие дает сравнительно более эффективные реализации, которые не дают достоверных результатов, когда предположение не выполняется. Большинство функций библиотеки C допускают более эффективные реализации, а `memmove()` заполняет пробелы, обслуживая случаи, когда источник и получатель могут или перекрываются. Однако для обеспечения правильного эффекта во всех случаях он должен выполнять дополнительные тесты и / или использовать сравнительно менее эффективную реализацию.

Чтение неинициализированного объекта, который не поддерживается памятью

C11

Чтение объекта приведет к неопределенному поведению, если объект равен [1] :

- неинициализированным
- с автоматической продолжительностью хранения
- его адрес никогда не принимается

Переменная a в приведенном ниже примере удовлетворяет всем этим условиям:

```
void Function( void )
{
    int a;
    int b = a;
}
```

---

[1] (Цитируется по: ISO: IEC 9899: 201X 6.3.2.1 Lvalues, массивы и обозначения функций 2) Если lvalue обозначает объект с продолжительностью автоматического хранения, который мог быть объявлен с классом хранения регистров (никогда не был принят его адрес), и этот объект не инициализирован (не объявлен с инициализатором, и его назначение не было выполнено до использования), поведение не определено.

Гонка данных

C11

C11 представила поддержку для нескольких потоков исполнения, что дает возможность

расследовать данные. Программа содержит гонку данных, если к объекту в ней обращаются [1] двумя разными потоками, где по крайней мере один из доступа является неатомным, по крайней мере один изменяет объект, а семантика программы не позволяет гарантировать, что два доступа не могут перекрываться во время. [2] Хорошо помните, что фактическая параллельность задействованных доступов не является условием гонки данных; Гонки данных охватывают более широкий класс проблем, возникающих из (разрешенных) несоответствий в представлениях разных потоков.

Рассмотрим этот пример:

```
#include <threads.h>

int a = 0;

int Function( void* ignore )
{
    a = 1;

    return 0;
}

int main( void )
{
    thrd_t id;
    thrd_create( &id , Function , NULL );

    int b = a;

    thrd_join( id , NULL );
}
```

Основные вызовы нити `thrd_create` , чтобы начать новую нить работает функция `Function` . Второй поток изменяет `a` , а основной поток читает `a` . Ни один из этих доступов не является атомарным, и эти два потока ничего не делают ни индивидуально, ни совместно, чтобы гарантировать, что они не перекрываются, поэтому существует гонка данных.

Среди способов, с помощью которых эта программа могла избежать гонки данных,

- основной поток может выполнять свое чтение из перед началом другого потока; `a`
- основной поток может выполнять свое чтение из после обеспечения с помощью `a` `thrd_join` , что другие прекратившие;
- потоки могли синхронизировать их обращения через мьютекс, каждый из которых блокировал этот мьютекс, прежде чем получить доступ к `a` и разблокировать его позже.

Как показывает опция mutex, избегая гонки данных, не требуется обеспечить определенный порядок операций, например, дочерний поток, модифицирующий `a` до того, как основной поток прочитает его; достаточно (для избежания гонки данных) обеспечить, чтобы для данного исполнения один доступ выполнялся перед другим.

---

[1] Изменение или чтение объекта.

[2] (Цитируется по ISO: IEC 9889: 201x, раздел 5.1.2.4 «Многопоточные исполнения и расписания данных»)
Выполнение программы содержит гонку данных, если она содержит два конфликтующих действия в разных потоках, по крайней мере один из которых не является атомарным, и не происходит до другого. Любая такая гонка данных приводит к неопределенному поведению.

Чтение значения указателя, который был освобожден

Даже просто **считывание** значения освобожденного указателя (т. Е. Без попытки разыменования указателя) является неопределенным поведением (UB), например

```
char *p = malloc(5);
free(p);
if (p == NULL) /* NOTE: even without dereferencing, this may have UB */
{

}
```

Цитирование **ISO / IEC 9899: 2011** , раздел 6.2.4. §2:

> [...] Значение указателя становится неопределенным, когда объект, на который он указывает (или только что прошел), достигает конца своего срока службы.

Использование неопределенной памяти для чего угодно, включая, по-видимому, безвредное сравнение или арифметику, может иметь неопределенное поведение, если это значение может быть ловушечным представлением для типа.

Изменить строковый литерал

В этом примере кода указатель char `p` инициализируется адресом строкового литерала. Попытка изменить строковый литерал имеет неопределенное поведение.

```
char *p = "hello world";
p[0] = 'H'; // Undefined behavior
```

Однако изменение измененного массива `char` напрямую или через указатель, естественно, не является неопределенным поведением, даже если его инициализатор является литеральной строкой. Хорошо:

```
char a[] = "hello, world";
char *p = a;

a[0] = 'H';
p[7] = 'W';
```

Это связано с тем, что строковый литерал эффективно копируется в массив каждый раз, когда массив инициализируется (один раз для переменных со статической продолжительностью, каждый раз, когда массив создается для переменных с автоматической или длительностью потока - переменные с назначенной продолжительностью не инициализируются) и это прекрасно, чтобы изменить содержимое массива.

Освобождение памяти дважды

Освободить память дважды - это неопределенное поведение, например

```
int * x = malloc(sizeof(int));
*x = 9;
free(x);
free(x);
```

Цитата из стандарта (7.20.3.2. Свободная функция C99):

> В противном случае, если аргумент не соответствует указателю, ранее возвращенному функцией calloc, malloc или realloc, или если пространство было освобождено вызовом free или realloc, поведение не определено.

Использование неверного спецификатора формата в printf

Использование неверного спецификатора формата в первом аргументе `printf` вызывает неопределенное поведение. Например, приведенный ниже код вызывает неопределенное поведение:

```
long z = 'B';
printf("%c\n", z);
```

Вот еще один пример

```
printf("%f\n",0);
```

Над строкой кода указано неопределенное поведение. `%f` ожидает двойной. Однако 0 имеет тип `int` .

Обратите внимание, что ваш компилятор обычно может помочь вам избежать таких случаев, если вы включите соответствующие флаги во время компиляции (`-Wformat` в `clang` и `gcc`). Из последнего примера:

```
warning: format specifies type 'double' but the argument has type
      'int' [-Wformat]
    printf("%f\n",0);
            ~~    ^
            %d
```

Преобразование между типами указателей приводит к неверно выровненному результату

Следующие *могут* иметь неопределенное поведение из-за неправильного выравнивания указателя:

```
char *memory_block = calloc(sizeof(uint32_t) + 1, 1);
uint32_t *intptr = (uint32_t*)(memory_block + 1);  /* possible undefined behavior */
uint32_t mvalue = *intptr;
```

Неопределенное поведение происходит по мере преобразования указателя. Согласно C11, если *преобразование между двумя типами указателей приводит к некорректному результату (6.3.2.3), поведение не определено* . Здесь `uint32_t` может потребовать выравнивания 2 или 4, например.

`calloc` с другой стороны, требуется вернуть указатель, подходящий для любого типа объекта; поэтому `memory_block` правильно выровнен, чтобы содержать `uint32_t` в своей начальной части. Затем в системе, где `uint32_t` требует выравнивания 2 или 4, `memory_block + 1` будет *нечетным* адресом и, следовательно, не будет правильно выровнен.

Обратите внимание, что стандарт C требует, чтобы операция литья была неопределенной. Это наложено, потому что на платформах, где адреса сегментированы, байт-адрес `memory_block + 1` может даже не иметь правильного представления в виде целочисленного указателя.

Кастинг `char *` для указателей на другие типы без каких-либо проблем с выравниванием иногда используется неправильно для декодирования упакованных структур, таких как заголовки файлов или сетевые пакеты.

Вы можете избежать неопределенного поведения, связанного с неправильным преобразованием указателя, используя `memcpy` :

```
memcpy(&mvalue, memory_block + 1, sizeof mvalue);
```

Здесь не происходит преобразования указателя в `uint32_t*` и байты копируются один за другим.

Эта операция копирования для нашего примера приводит только к допустимому значению `mvalue` потому что:

- Мы использовали `calloc` , поэтому байты правильно инициализированы. В нашем случае все байты имеют значение `0` , но любая другая правильная инициализация.
- `uint32_t` - это точный тип ширины и не имеет битов заполнения
- Любой произвольный битовый шаблон является допустимым представлением для любого неподписанного типа.

Сложение или вычитание указателя неправильно ограничено

Следующий код имеет неопределенное поведение:

```
char buffer[6] = "hello";
char *ptr1 = buffer - 1;  /* undefined behavior */
char *ptr2 = buffer + 5;  /* OK, pointing to the '\0' inside the array */
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char *ptr4 = buffer + 7;  /* undefined behavior */
```

В соответствии с C11, если сложение или вычитание указателя на объект или объект, находящийся за ним за пределы объекта, приводит к результату, который не указывает на один объект массива или только за ним, поведение не определено (6.5.6).

Кроме того, это естественно неопределенное поведение для *разыменования* указателя, который указывает только на массив:

```
char buffer[6] = "hello";
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char value = *ptr3;       /* undefined behavior */
```

Изменение константной переменной с помощью указателя

```
int main (void)
{
    const int foo_readonly = 10;
    int *foo_ptr;

    foo_ptr = (int *)&foo_readonly; /* (1) This casts away the const qualifier */
    *foo_ptr = 20; /* This is undefined behavior */

    return 0;
}
```

Цитирование *ISO / IEC 9899: 201x* , раздел 6.7.3 §2:

> Если предпринимается попытка изменить объект, определенный с помощью типа, соответствующего const, с использованием значения lvalue с неконстантированным классом, поведение не определено. [...]

---

(1) В GCC это может вызвать следующее предупреждение: `warning: assignment discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]`

Передача нулевого указателя на преобразование printf% s

Преобразование `%s` `printf` утверждает, что соответствующий аргумент *является указателем на исходный элемент массива типа символа* . Нулевой указатель не указывает на начальный элемент любого массива символьного типа, и поэтому поведение следующего

не определено:

```
char *foo = NULL;
printf("%s", foo); /* undefined behavior */
```

Однако неопределенное поведение не всегда означает, что программа вылетает из строя - некоторые системы предпринимают шаги, чтобы избежать сбоя, который обычно происходит, когда нулевой указатель разыменовывается. Например, Glibc, как известно, печатает

```
(null)
```

для кода выше. Однако добавьте (просто) новую строку в строку формата, и вы получите сбой:

```
char *foo = 0;
printf("%s\n", foo); /* undefined behavior */
```

В этом случае это происходит потому, что GCC имеет оптимизацию, которая возвращает `printf("%s\n", argument);` в вызов `puts` с `puts(argument)` , и `puts` в Glibc не обрабатывает нулевые указатели. Все это поведение стандартно.

Обратите внимание: *нулевой указатель* отличается от *пустой строки* . Итак, справедливо следующее и не имеет неопределенного поведения. Он просто напечатает *новую строку* :

```
char *foo = "";
printf("%s\n", foo);
```

Несогласованная привязка идентификаторов

```
extern int var;
static int var; /* Undefined behaviour */
```

C11, §6.2.2, 7 гласит:

> Если внутри единицы перевода один и тот же идентификатор появляется как с внутренней, так и с внешней связью, поведение не определено.

Обратите внимание: если предыдущее объявление идентификатора будет видимым, оно будет иметь связь с предыдущим объявлением. *C11, §6.2.2, 4* позволяет:

> Для идентификатора, объявленного с указанием класса хранилища extern в области, в которой видна предварительная декларация этого идентификатора, 31), если предыдущее объявление указывает внутреннюю или внешнюю связь, связь идентификатора с последующим объявлением такая же, как и связь, указанная в предыдущей декларации. Если никакое предварительное

объявление не видно или если в предыдущем объявлении не указано никакой привязки, идентификатор имеет внешнюю связь.

```
/* 1. This is NOT undefined */
static int var;
extern int var;


/* 2. This is NOT undefined */
static int var;
static int var;

/* 3. This is NOT undefined */
extern int var;
extern int var;
```

Использование fflush на входном потоке

Стандарты POSIX и C явно `fflush` что использование `fflush` во входном потоке является неопределенным поведением. `fflush` определяется только для выходных потоков.

```
#include <stdio.h>

int main()
{
    int i;
    char input[4096];

    scanf("%i", &i);
    fflush(stdin); // <-- undefined behavior
    gets(input);

    return 0;
}
```

Нет стандартного способа удаления непрочитанных символов из входного потока. С другой стороны, некоторые реализации используют `fflush` для очистки буфера `stdin` . Microsoft определяет поведение `fflush` во входном потоке: если поток открыт для ввода, `fflush` очищает содержимое буфера. Согласно POSIX.1-2008, поведение `fflush` не определено, если входной файл не доступен для поиска.

Дополнительную информацию см. В разделе Использование `fflush(stdin)` .

Бит-сдвиг с использованием отрицательных отсчетов или за пределами ширины типа

Если значение *счетчика сдвига* является **отрицательным значением,** то обе операции *сдвига влево* и *вправо* не определены [1] :

```
int x = 5 << -3; /* undefined */
int x = 5 >> -3; /* undefined */
```

Если *сдвиг влево* выполняется по **отрицательному значению** , он не определен:

```
int x = -5 << 3; /* undefined */
```

Если *сдвиг влево* выполняется по **положительному значению,** и результат математического значения **не** представляется в типе, он не определен [1] :

```
/* Assuming an int is 32-bits wide, the value '5 * 2^72' doesn't fit
 * in an int. So, this is undefined. */

int x = 5 << 72;
```

Обратите внимание, что *сдвиг вправо* по **отрицательному значению** (.eg `-5 >> 3`) *не* является неопределенным, а *определяется реализацией* .

---

[1] Цитирование *ISO / IEC 9899: 201x* , раздел 6.5.7:

> Если значение правого операнда отрицательное или больше или равно ширине продвинутого левого операнда, поведение не определено.

Изменение строки, возвращаемой функциями getenv, strerror и setlocale

Изменение строк, возвращаемых стандартными функциями `getenv()` , `strerror()` и `setlocale()` не определено. Таким образом, реализации могут использовать статическое хранилище для этих строк.

Функция getenv (), C11, §7.22.4.7, 4 , говорит:

> Функция getenv возвращает указатель на строку, связанную с элементом согласованного списка. Указанная строка не должна изменяться программой, но может быть перезаписана последующим вызовом функции getenv.

Функция strerror (), C11, §7.23.6.3, 4 говорит:

> Функция strerror возвращает указатель на строку, содержимое которой является локальным. Указанный массив не должен быть изменен программой, но может быть перезаписан последующим вызовом функции strerror.

Функция setlocale (), C11, §7.11.1.1, 8 гласит:

> Указатель на строку, возвращаемую функцией setlocale, таков, что последующий вызов с этим строковым значением и связанной с ним категорией восстановит эту часть локали программы. Указанная строка не должна изменяться программой, но может быть перезаписана последующим вызовом функции setlocale.

Точно так же функция `localeconv()` возвращает указатель на `struct lconv` который не должен быть изменен.

Функция localeconv (), C11, §7.11.2.1, 8 гласит:

> Функция localeconv возвращает указатель на заполненный объект. Структура, на которую указывает возвращаемое значение, не должна изменяться программой, но может быть перезаписана последующим вызовом функции localeconv.

Возврат из функции, объявленной с помощью спецификатора функции `_Noreturn` или` noreturn`

C11

Спецификатор функции `_Noreturn` был введен в C11. Заголовок `<stdnoreturn.h>` предоставляет макрос `noreturn` который расширяется до `_Noreturn` . Поэтому использование `_Noreturn` или `noreturn` из `<stdnoreturn.h>` является прекрасным и эквивалентным.

Функция, объявленная с помощью `_Noreturn` (или `noreturn`), не может вернуться к вызывающей `noreturn` . Если такая функция *не* вернется к абоненту, поведение не определено.

В следующем примере `func()` объявляется с `noreturn` спецификатора `noreturn` но возвращается к вызывающему.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void func(void);

void func(void)
{
    printf("In func()...\n");
} /* Undefined behavior as func() returns */

int main(void)
{
    func();
    return 0;
}
```

`gcc` и `clang` вызывают предупреждения для вышеуказанной программы:

```
$ gcc test.c
test.c: In function ‘func’:
test.c:9:1: warning: ‘noreturn’ function does return
 }
 ^
$ clang test.c
```

```
test.c:9:1: warning: function declared 'noreturn' should not return [-Winvalid-noreturn]
}
^
```

Пример использования `noreturn` который имеет четко определенное поведение:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void my_exit(void);

/* calls exit() and doesn't return to its caller. */
void my_exit(void)
{
    printf("Exiting...\n");
    exit(0);
}

int main(void)
{
    my_exit();
    return 0;
}
```

Прочитайте Неопределенное поведение онлайн: https://riptutorial.com/ru/c/topic/364/неопределенное-поведение

глава 26: Неявные и явные конверсии

Синтаксис

- Явное преобразование (иначе говоря, «Кастинг»): (тип) выражение

замечания

« *Явное преобразование* » также обычно называют «литье».

Examples

Целочисленные преобразования в вызовах функций

Учитывая, что функция имеет правильный прототип, целые числа расширяются для вызовов функций в соответствии с правилами целочисленного преобразования, C11 6.3.1.3.

> **6.3.1.3 Целочисленные и беззнаковые целые числа**
> Когда значение с целым типом преобразуется в другой целочисленный тип, отличный от _Bool, если значение может быть представлено новым типом, оно не изменяется.
>
> В противном случае, если новый тип без знака, значение преобразуется путем многократного добавления или вычитания более чем максимального значения, которое может быть представлено в новом типе, пока значение не будет в диапазоне нового типа.
>
> В противном случае новый тип будет подписан и значение не может быть представлено в нем; либо результат определяется реализацией, либо генерируется сигнал, определяемый реализацией.

Обычно вы не должны усекать широко подписанный тип до более узкого типа, потому что, очевидно, значения не могут соответствовать, и нет четкого значения, которое это должно иметь. Приведенный выше стандарт C определяет эти случаи как «определенные реализацией», то есть они не переносимы.

Следующий пример предполагает, что `int` имеет ширину 32 бит.

```
#include <stdio.h>
#include <stdint.h>

void param_u8(uint8_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
```

```
}

void param_u16(uint16_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_u32(uint32_t val) {
    printf("%s val is %u\n", __func__, val);  /* here val fits into unsigned */
}

void param_u64(uint64_t val) {
    printf("%s val is " PRI64u "\n", __func__, val); /* Fixed with format string */
}

void param_s8(int8_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_s16(int16_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_s32(int32_t val) {
    printf("%s val is %d\n", __func__, val); /* val has same width as int */
}

void param_s64(int64_t val) {
    printf("%s val is " PRI64d "\n", __func__, val); /* Fixed with format string */
}

int main(void) {

    /* Declare integers of various widths */
    uint8_t  u8  = 127;
    uint8_t  s64  = INT64_MAX;

    /* Integer argument is widened when function parameter is wider */
    param_u8(u8);   /* param_u8 val is 127 */
    param_u16(u8);  /* param_u16 val is 127 */
    param_u32(u8);  /* param_u32 val is 127 */
    param_u64(u8);  /* param_u64 val is 127 */
    param_s8(u8);   /* param_s8 val is 127 */
    param_s16(u8);  /* param_s16 val is 127 */
    param_s32(u8);  /* param_s32 val is 127 */
    param_s64(u8);  /* param_s64 val is 127 */

    /* Integer argument is truncated when function parameter is narrower */
    param_u8(s64);  /* param_u8 val is 255 */
    param_u16(s64); /* param_u16 val is 65535 */
    param_u32(s64); /* param_u32 val is 4294967295 */
    param_u64(s64); /* param_u64 val is 9223372036854775807 */
    param_s8(s64);  /* param_s8 val is implementation defined */
    param_s16(s64); /* param_s16 val is implementation defined */
    param_s32(s64); /* param_s32 val is implementation defined */
    param_s64(s64); /* param_s64 val is 9223372036854775807 */

    return 0;
}
```

Конверсии указателей в вызовах функций

Преобразование указателей в `void*` неявно, но любое другое преобразование указателя должно быть явным. Хотя компилятор допускает явное преобразование любого типа указателя на данные в любой другой тип указателя на данные, доступ к объекту с помощью неправильно введенного указателя является ошибочным и приводит к неопределенному поведению. Единственный случай, когда они разрешены, - если типы совместимы или если указатель, с которым вы смотрите объект, является типом символа.

```
#include <stdio.h>

void func_voidp(void* voidp) {
    printf("%s Address of ptr is %p\n", __func__, voidp);
}

/* Structures have same shape, but not same type */
struct struct_a {
    int a;
    int b;
} data_a;

struct struct_b {
    int a;
    int b;
} data_b;

void func_struct_b(struct struct_b* bp) {
    printf("%s Address of ptr is %p\n", __func__, (void*) bp);
}

int main(void) {


    /* Implicit ptr conversion allowed for void* */
    func_voidp(&data_a);

    /*
     * Explicit ptr conversion for other types
     *
     * Note that here although the have identical definitions,
     * the types are not compatible, and that the this call is
     * erroneous and leads to undefined behavior on execution.
     */
    func_struct_b((struct struct_b*)&data_a);

    /* My output shows: */
    /* func_charp Address of ptr is 0x601030 */
    /* func_voidp Address of ptr is 0x601030 */
    /* func_struct_b Address of ptr is 0x601030 */

    return 0;
}
```

Прочитайте Неявные и явные конверсии онлайн: https://riptutorial.com/ru/c/topic/2529/неявные-и-явные-конверсии

глава 27: Область идентификатора

Examples

Область применения

Идентификатор имеет область блока, если соответствующее объявление появляется внутри блока (применяется объявление параметра в определении функции). Область заканчивается в конце соответствующего блока.

Никакие другие объекты с одним и тем же идентификатором не могут иметь одинаковую область видимости, но области могут перекрываться. В случае пересекающихся областей видимость видима только одна, заявленная в самой внутренней области.

```
#include <stdio.h>

void test(int bar)                    // bar has scope test function block
{
    int foo = 5;                      // foo has scope test function block
    {
        int bar = 10;                 // bar has scope inner block, this overlaps with previous
test:bar declaration, and it hides test:bar
        printf("%d %d\n", foo, bar); // 5 10
    }                                 // end of scope for inner bar
    printf("%d %d\n", foo, bar);      // 5 5, here bar is test:bar
}                                     // end of scope for test:foo and test:bar

int main(void)
{
    int foo = 3;          // foo has scope main function block

    printf("%d\n", foo); // 3
    test(5);
    printf("%d\n", foo); // 3
    return 0;
}                         // end of scope for main:foo
```

Функция Prototype Scope

```
#include <stdio.h>

/* The parameter name, apple, has function prototype scope.  These names
   are not significant outside the prototype itself.  This is demonstrated
   below. */

int test_function(int apple);

int main(void)
{
    int orange = 5;

    orange = test_function(orange);
```

```
    printf("%d\r\n", orange); //orange = 6

    return 0;
}

int test_function(int fruit)
{
    fruit += 1;
    return fruit;
}
```

Обратите внимание, что вы получаете непонятные сообщения об ошибках, если ввести прототип названия типа:

```
int function(struct whatever *arg);

struct whatever
{
    int a;
    // ...
};

int function(struct whatever *arg)
{
    return arg->a;
}
```

С GCC 6.3.0 этот код (исходный файл `dc11.c`) создает:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -c dc11.c
dc11.c:1:25: error: ‘struct whatever’ declared inside parameter list will not be visible
outside of this definition or declaration [-Werror]
     int function(struct whatever *arg);
                         ^~~~~~~~
dc11.c:9:9: error: conflicting types for ‘function’
     int function(struct whatever *arg)
         ^~~~~~~~
dc11.c:1:9: note: previous declaration of ‘function’ was here
     int function(struct whatever *arg);
         ^~~~~~~~
cc1: all warnings being treated as errors
$
```

Поместите определение структуры перед объявлением функции или добавьте `struct whatever;` как строка перед объявлением функции, и нет проблем. Вы не должны вводить новые имена типов в прототипе функции, потому что нет способа использовать этот тип, и, следовательно, нет способа определить или использовать эту функцию.

Область файла

```
#include <stdio.h>

/* The identifier, foo, is declared outside all blocks.
   It can be used anywhere after the declaration until the end of
```

```
   the translation unit. */
static int foo;

void test_function(void)
{
    foo += 2;
}

int main(void)
{
    foo = 1;

    test_function();
    printf("%d\r\n", foo); //foo = 3;

    return 0;
}
```

Область действия

Область функций - это специальная область для **ярлыков** . Это связано с их необычным свойством. **Метка** отображается через всю функцию, которую она определена, и ее можно перескакивать (используя инструкцию `goto label`) из любой точки в той же функции. Хотя это не полезно, следующий пример иллюстрирует точку:

```
#include <stdio.h>

int main(int argc,char *argv[]) {
    int a = 0;
    goto INSIDE;
  OUTSIDE:
    if (a!=0) {
        int i=0;
      INSIDE:
        printf("a=%d\n",a);
        goto OUTSIDE;
    }
}
```

`INSIDE` может показаться определенным *внутри* блока `if` , так как это имеет место для `i` который является блоком, но это не так. Он виден во всей функции, поскольку инструкция `goto INSIDE;` иллюстрирует. Таким образом, не может быть двух меток с одним и тем же идентификатором в одной функции.

Возможным использованием является следующий шаблон для реализации правильной комплексной очистки выделенных ресурсов:

```
#include <stdlib.h>
#include <stdio.h>

void a_function(void) {
   double* a = malloc(sizeof(double[34]));
   if (!a) {
      fprintf(stderr,"can't allocate\n");
```

```
      return;                 /* No point in freeing a if it is null */
   }
   FILE* b = fopen("some_file","r");
   if (!b) {
      fprintf(stderr,"can't open\n");
      goto CLEANUP1;          /* Free a; no point in closing b */
   }
   /* do something reasonable */
   if (error) {
      fprintf(stderr,"something's wrong\n");
      goto CLEANUP2;       /* Free a and close b to prevent leaks */
   }
   /* do yet something else */
CLEANUP2:
   close(b);
CLEANUP1:
   free(a);
}
```

Ярлыки, такие как `CLEANUP1` и `CLEANUP2` являются специальными идентификаторами, которые ведут себя по-другому от всех других идентификаторов. Они видны повсюду внутри функции, даже в местах, которые выполняются перед помеченным заявлением, или даже в местах, которые никогда не могут быть достигнуты, если ни одно из `goto` не выполняется. Ярлыки часто пишутся в нижнем регистре, а не в верхнем регистре.

Прочитайте Область идентификатора онлайн: https://riptutorial.com/ru/c/topic/1804/область-идентификатора

глава 28: Обработка ошибок

Синтаксис

- #include <errno.h>
- int errno; / * определенная реализация * /
- #include <string.h>
- char * strerror (int errnum);
- #include <stdio.h>
- void perror (const char * s);

замечания

Имейте в виду, что `errno` не обязательно является переменной, но синтаксис является лишь показателем того, как он *мог* быть объявлен. Во многих современных системах с интерфейсами потоков `errno` представляет собой макрос, который разрешает объект, локальный для текущего потока.

Examples

ERRNO

Когда стандартная функция библиотеки выходит из строя, она часто устанавливает errno в соответствующий код ошибки. Для стандарта C требуется установить не менее 3 значений для errno:

Значение	Имея в виду
EDOM	Ошибка домена
ERANGE	Ошибка диапазона
EILSEQ	Нелегальная многобайтовая последовательность символов

strerror

Если `perror` недостаточно гибкий, вы можете получить пользовательское описание ошибки, вызвав `strerror` из `<string.h>` .

```
int main(int argc, char *argv[])
{
    FILE *fout;
    int last_error = 0;
```

```
    if ((fout = fopen(argv[1], "w")) == NULL) {
        last_error = errno;
         /* reset errno and continue */
         errno = 0;
    }

    /* do some processing and try opening the file differently, then */


    if (last_error) {
        fprintf(stderr, "fopen: Could not open %s for writing: %s",
                argv[1], strerror(last_error));
        fputs("Cross fingers and continue", stderr);
    }

    /* do some other processing */

    return EXIT_SUCCESS;
}
```

PError

Чтобы напечатать прочитанное пользователем сообщение об ошибке в `stderr` , вызовите `perror` из <stdio.h>.

```
int main(int argc, char *argv[])
{
   FILE *fout;

   if ((fout = fopen(argv[1], "w")) == NULL) {
      perror("fopen: Could not open file for writing");
      return EXIT_FAILURE;
   }
return EXIT_SUCCESS;
}
```

Это выведет сообщение об ошибке относительно текущего значения `errno` .

Прочитайте Обработка ошибок онлайн: https://riptutorial.com/ru/c/topic/2486/обработка-ошибок

глава 29: Обработка сигналов

Синтаксис

- void (* signal (int sig, void (* func) (int))) (int);

параметры

параметр	подробности
сиг	Сигнал для установки обработчика сигналов, один из `SIGABRT` , `SIGFPE` , `SIGILL` , `SIGTERM` , `SIGINT` , `SIGSEGV` или некоторое определенное значение реализации
FUNC	Обработчик сигнала, который является одним из следующих: `SIG_DFL` , для обработчика по умолчанию, `SIG_IGN` для игнорирования сигнала или указателя функции с сигнатурой `void foo(int sig);` ,

замечания

Использование обработчиков сигналов только с гарантиями из стандарта C накладывает различные ограничения, которые могут или не могут быть выполнены в пользовательском обработчике сигналов.

- Если функция, определенная пользователем, возвращается при обработке `SIGSEGV` , `SIGFPE` , `SIGILL` или любого другого аппаратного прерывания, определяемого реализацией, поведение не определено стандартом C. Это связано с тем, что интерфейс C не дает средств для изменения неисправного состояния (например, после деления на `0`), и поэтому при возврате программа находится в точно таком же ошибочном состоянии, что и до того, как произошло прерывание аппаратного обеспечения.
- Если пользовательская функция была вызвана в результате вызова `abort` или `raise` , обработчик сигнала не может снова вызвать `raise` .
- Сигналы могут поступать в середине любой операции, и поэтому неделимость операций может вообще не гарантироваться, и обработка сигналов не работает с оптимизацией. Поэтому все изменения данных в обработчике сигналов должны быть в переменных
    - типа `sig_atomic_t` (все версии) или атомарного типа без блокировки (поскольку C11, необязательный)

    - которые являются `volatile` .

- Другие функции из стандартной библиотеки C обычно не будут соблюдать эти ограничения, поскольку они могут изменять переменные в глобальном состоянии программы. Стандарт C только делает гарантии для `abort` , `_Exit` (с C99), `quick_exit` (с C11), `signal` (для того же самого числа сигналов), а также некоторые атомарные операции (начиная с C11).

Поведение не определено стандартом C, если нарушено какое-либо из вышеперечисленных правил. Платформы могут иметь определенные расширения, но они, как правило, не переносятся за пределы этой платформы.

- Обычно системы имеют свой собственный список функций, которые *безопасны для асинхронного сигнала* , то есть из функций библиотеки C, которые могут использоваться из обработчика сигналов. Например, часто функция `printf` входит в число этих функций.

- В частности, стандарт C не определяет многое о взаимодействии с его интерфейсом потоков (начиная с C11) или с любыми конкретными библиотеками потоков, такими как потоки POSIX. Такие платформы должны сами определять взаимодействие таких библиотек потоков с сигналами.

Examples

Обработка сигналов с помощью «signal ()»

Номера сигналов могут быть синхронными (например, `SIGSEGV` - segmentation fault), когда они срабатывают при неправильной работе самой программы или асинхронной (например, `SIGINT` - интерактивное внимание), когда они инициируются извне программы, например, `Cntrl-C` клавиши `Cntrl-C` .

Функция `signal()` является частью стандарта ISO C и может использоваться для назначения функции для обработки определенного сигнала

```
#include <stdio.h>  /* printf() */
#include <stdlib.h> /* abort()  */
#include <signal.h> /* signal() */

void handler_nonportable(int sig)
{
    /* undefined behavior, maybe fine on specific platform */
    printf("Catched: %d\n", sig);

    /* abort is safe to call */
    abort();
}

sig_atomic_t volatile finished = 0;
```

```
void handler(int sig)
{
    switch (sig) {
    /* hardware interrupts should not return */
    case SIGSEGV:
    case SIGFPE:
    case SIGILL:
```

C11

```
      /* quick_exit is safe to call */
      quick_exit(EXIT_FAILURE);
```

C11

```
      /* use _Exit in pre-C11 */
      _Exit(EXIT_FAILURE);
```

```
    default:
       /* Reset the signal to the default handler,
          so we will not be called again if things go
          wrong on return. */
      signal(sig, SIG_DFL);
      /* let everybody know that we are finished */
      finished = sig;
      return;
   }
}

int main(void)
{

    /* Catch the SIGSEGV signal, raised on segmentation faults (i.e NULL ptr access */
    if (signal(SIGSEGV, &handler) == SIG_ERR) {
        perror("could not establish handler for SIGSEGV");
        return EXIT_FAILURE;
    }

    /* Catch the SIGTERM signal, termination request */
    if (signal(SIGTERM, &handler) == SIG_ERR) {
        perror("could not establish handler for SIGTERM");
        return EXIT_FAILURE;
    }

    /* Ignore the SIGINT signal, by setting the handler to `SIG_IGN`. */
    signal(SIGINT, SIG_IGN);


    /* Do something that takes some time here, and leaves
       the time to terminate the program from the keyboard. */

    /* Then: */

    if (finished) {
       fprintf(stderr, "we have been terminated by signal %d\n", (int)finished);
        return EXIT_FAILURE;
    }
```

```
    /* Try to force a segmentation fault, and raise a SIGSEGV */
    {
      char* ptr = 0;
      *ptr = 0;
    }

    /* This should never be executed */
    return EXIT_SUCCESS;
}
```

Использование `signal()` налагает важные ограничения на то, что вам разрешено делать внутри обработчиков сигналов, см. Примечания для получения дополнительной информации.

POSIX рекомендует использовать `sigaction()` вместо `signal()` из-за его неопределенного поведения и значительных вариантов реализации. POSIX также определяет гораздо больше сигналов, чем стандарт ISO C, включая `SIGUSR1` и `SIGUSR2` , которые могут использоваться свободно программистом для любых целей.

Прочитайте Обработка сигналов онлайн: https://riptutorial.com/ru/c/topic/453/обработка-сигналов

глава 30: Общие идиомы программирования C и методы разработчика

Examples

Сравнение литералов и переменных

Предположим, что вы сравниваете значение с некоторой переменной

```
if ( i  == 2) //Bad-way
{
    doSomething;
}
```

Теперь предположим, что вы ошибаетесь `==` с `=` . Тогда вам понадобится ваше сладкое время, чтобы понять это.

```
if( 2 == i) //Good-way
{
    doSomething;
}
```

Затем, если знак равенства случайно исключен, компилятор будет жаловаться на «попытку присвоения литералу». Это не будет защищать вас при сравнении двух переменных, но каждый бит помогает.

См. Здесь для получения дополнительной информации.

Не оставляйте список параметров функции пустым - используйте void

Предположим, вы создаете функцию, которая не требует никаких аргументов при ее вызове, и вы столкнулись с дилеммой о том, как вы должны определить список параметров в прототипе функции и определении функции.

- У вас есть выбор: сохранить список параметров пустым как для прототипа, так и для определения. Таким образом, они выглядят так же, как и запрос вызова функции.
- Вы где-то читали, что одно из применений ключевого слова ***void*** (их всего несколько) - это определение списка параметров функций, которые не принимают никаких аргументов в их вызове. Таким образом, это тоже выбор.

Итак, что является правильным выбором?

ОТВЕТ: использование ключевого слова ***void***

ОБЩИЕ КОНСУЛЬТАЦИИ: Если язык предоставляет определенную функцию для использования в специальных целях, вам лучше использовать это в своем коде. Например, используя `enum` s вместо `#define` макросов (это для другого примера).

Раздел C11 6.7.6.3 В пункте 10 «Деклараторы функций» говорится:

> Частный случай неименованного параметра типа void как единственного элемента в списке указывает, что функция не имеет параметров.

Параграф 14 этого же раздела показывает единственную разницу:

> ... Пустой список в деклараторе функции, который является частью определения этой функции, указывает, что функция не имеет параметров. Пустой список в объявлении функции, который не является частью определения этой функции, указывает, что информация о количестве или типах параметров не указана.

Упрощенное объяснение, представленное K & R (pgs-72-73) для вышеуказанного материала:

> Кроме того, если объявление функции не содержит аргументов, как в `double atof();` , это тоже означает, что ничего не следует предполагать в отношении аргументов `atof` ; вся проверка параметров отключена. Это специальное значение пустого списка аргументов предназначено для того, чтобы более старые программы на C могли компилироваться с новыми компиляторами. Но это плохая идея использовать его с новыми программами. Если функция принимает аргументы, объявляйте их; если он не принимает аргументов, используйте `void` .

Так вот как должен выглядеть ваш прототип функции:

```
int foo(void);
```

И так должно быть определение функции:

```
int foo(void)
{
    ...
    <statements>
    ...
    return 1;
}
```

Одно из преимуществ использования выше, над объявлением типа `int foo()` (т. Е. Без использования ключевого слова ***void***) заключается в том, что компилятор может

обнаружить ошибку, если вы вызываете свою функцию, используя ошибочное утверждение, например `foo(42)` . Такой тип выражения вызова функции не вызывает никаких ошибок, если оставить список параметров пустым. Ошибка будет проходить молча, незаметно, и код все равно будет выполняться.

Это также означает, что вы должны определить функцию `main()` следующим образом:

```
int main(void)
{
    ...
    <statements>
    ...
    return 0;
}
```

Обратите внимание, что хотя функция, определенная с пустым списком параметров, не принимает аргументов, она не предоставляет прототип функции, поэтому компилятор не будет жаловаться, если функция впоследствии вызывается с аргументами. Например:

```
#include <stdio.h>

static void parameterless()
{
    printf("%s called\n", __func__);
}

int main(void)
{
    parameterless(3, "arguments", "provided");
    return 0;
}
```

Если этот код сохраняется в файле `proto79.c` , его можно скомпилировать в Unix с GCC (версия 7.1.0 на macOS Sierra 10.12.5, используемая для демонстрации), например:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -pedantic proto79.c -o
proto79
$
```

Если вы скомпилируете более строгие параметры, вы получите ошибки:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-
style-definition -pedantic proto79.c -o proto79
proto79.c:3:13: error: function declaration isn't a prototype [-Werror=strict-prototypes]
 static void parameterless()
             ^~~~~~~~~~~~~
proto79.c: In function 'parameterless':
proto79.c:3:13: error: old-style function definition [-Werror=old-style-definition]
cc1: all warnings being treated as errors
$
```

Если вы даете функции формальный прототип `static void parameterless(void)` , то

компиляция дает ошибки:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-
style-definition -pedantic proto79.c -o proto79
proto79.c: In function ‘main’:
proto79.c:10:5: error: too many arguments to function ‘parameterless’
     parameterless(3, "arguments", "provided");
     ^~~~~~~~~~~~
proto79.c:3:13: note: declared here
 static void parameterless(void)
             ^~~~~~~~~~~~
$
```

Мораль - всегда убедитесь, что у вас есть прототипы, и убедитесь, что ваш компилятор говорит вам, когда вы не соблюдаете правила.

Прочитайте Общие идиомы программирования C и методы разработчика онлайн: https://riptutorial.com/ru/c/topic/10543/общие-идиомы-программирования-с-и-методы-разработчика

глава 31: Общий выбор

Синтаксис

- _Generic (присваивание-выражение, общий-ассоциированный список)

параметры

параметр	подробности
родовое-ассоциативный список	generic-association **ИЛИ** generic-assoc-list, generic-association
родовая-ассоциация	type-name: присваивание-выражение **ИЛИ** default: присваивание-выражение

замечания

1. Все квалификаторы типов будут `_Generic` во время оценки первичного выражения `_Generic` .
2. `_Generic` первичное выражение оценивается на этапе 7 перевода . Таким образом, фазы, такие как конкатенация строк, были закончены до его оценки.

Examples

Проверьте, имеет ли переменная определенный квалифицированный тип

```
#include <stdio.h>

#define is_const_int(x) _Generic((&x),  \
        const int *: "a const int",     \
        int *:       "a non-const int", \
        default:     "of other type")

int main(void)
{
    const int i = 1;
    int j = 1;
    double k = 1.0;
    printf("i is %s\n", is_const_int(i));
    printf("j is %s\n", is_const_int(j));
    printf("k is %s\n", is_const_int(k));
}
```

Выход:

```
i is a const int
j is a non-const int
k is of other type
```

Однако, если общий макрос типа реализуется следующим образом:

```
#define is_const_int(x) _Generic((x), \
        const int: "a const int",     \
        int:       "a non-const int", \
        default:   "of other type")
```

Выход:

```
i is a non-const int
j is a non-const int
k is of other type
```

Это связано с тем, что все квалификаторы типов отбрасываются для оценки управляющего выражения первичного выражения `_Generic` .

Макрос типовой печати

```
#include <stdio.h>

void print_int(int x) { printf("int: %d\n", x); }
void print_dbl(double x) { printf("double: %g\n", x); }
void print_default() { puts("unknown argument"); }

#define print(X) _Generic((X), \
        int: print_int, \
        double: print_dbl, \
        default: print_default)(X)

int main(void) {
    print(42);
    print(3.14);
    print("hello, world");
}
```

Выход:

```
int: 42
double: 3.14
unknown argument
```

Обратите внимание, что если тип не является ни `int` ни `double` , будет создано предупреждение. Чтобы исключить предупреждение, вы можете добавить этот тип в макрос `print(X)` .

Общий выбор, основанный на нескольких аргументах

Если требуется выбор по нескольким аргументам для типового выражения типа, и все типы, о которых идет речь, являются арифметическими типами, простой способ избежать вложенных выражений `_Generic` заключается в использовании добавления параметров в управляющем выражении:

```
int max_int(int, int);
unsigned max_unsigned(unsigned, unsigned);
double max_double(double, double);

#define MAX(X, Y) _Generic((X)+(Y),                \
                           int:      max_int,      \
                           unsigned: max_unsigned, \
                           default:  max_double)   \
                     ((X), (Y))
```

Здесь управляющее выражение `(X)+(Y)` проверяется только по типу и не оценивается. Для определения выбранного типа выполняются обычные преобразования для арифметических операндов.

Для более сложной ситуации выбор может быть сделан на основе нескольких аргументов оператору, путем их объединения.

В этом примере выбираются четыре функции, выполняемые извне, которые принимают комбинации из двух аргументов int и / или string и возвращают их сумму.

```
int AddIntInt(int a, int b);
int AddIntStr(int a, const char* b);
int AddStrInt(const char*  a, int b );
int AddStrStr(const char*  a, const char*  b);

#define AddStr(y)                              \
   _Generic((y),         int: AddStrInt,     \
                        char*: AddStrStr,    \
                  const char*: AddStrStr )

#define AddInt(y)                              \
   _Generic((y),          int: AddIntInt,     \
                        char*: AddIntStr,    \
                  const char*: AddIntStr )

#define Add(x, y)                              \
   _Generic((x) ,        int: AddInt(y) ,    \
                       char*: AddStr(y) ,    \
                 const char*: AddStr(y))     \
                         ((x), (y))

int main( void )
{
    int result = 0;
    result = Add( 100 , 999 );
    result = Add( 100 , "999" );
    result = Add( "100" , 999 );
    result = Add( "100" , "999" );

    const int a = -123;
```

```
    char b[] = "4321";
    result = Add( a , b );

    int c = 1;
    const char d[] = "0";
    result = Add( d , ++c );
}
```

Несмотря на то, что он выглядит так, как если бы аргумент `y` оценивался более одного раза, это не [1] . Оба аргумента оцениваются только один раз, в конце макроса Add: `( x , y )` , как и в обычном вызове функции.

---

[1] (Цитируется по: ISO: IEC 9899: 201X 6.5.1.1 Общий выбор 3)
Контролирующее выражение общего выбора не оценивается.

Прочитайте Общий выбор онлайн: https://riptutorial.com/ru/c/topic/571/общий-выбор

глава 32: Объявления

замечания

Объявление идентификатора, ссылающегося на объект или функцию, часто упоминается как краткое как просто объявление объекта или функции.

Examples

Вызов функции из другого файла C

foo.h

```
#ifndef FOO_DOT_H    /* This is an "include guard" */
#define FOO_DOT_H    /* prevents the file from being included twice. */
                     /* Including a header file twice causes all kinds */
                     /* of interesting problems.*/

/**
 * This is a function declaration.
 * It tells the compiler that the function exists somewhere.
 */
void foo(int id, char *name);

#endif /* FOO_DOT_H */
```

foo.c

```
#include "foo.h"    /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.  Put this
                     * header first in foo.c to ensure the header is self-contained.
                     */
#include <stdio.h>

/**
 * This is the function definition.
 * It is the actual body of the function which was declared elsewhere.
 */
void foo(int id, char *name)
{
    fprintf(stderr, "foo(%d, \"%s\");\n", id, name);
    /* This will print how foo was called to stderr - standard error.
     * e.g., foo(42, "Hi!") will print `foo(42, "Hi!")`
     */
}
```

main.c

```
#include "foo.h"
```

```
int main(void)
{
    foo(42, "bar");
    return 0;
}
```

Компиляция и ссылка

Сначала мы *скомпилируем файлы* `foo.c` и `main.c` в *объектные файлы* . Здесь мы используем компилятор `gcc` , ваш компилятор может иметь другое имя и нуждаться в других параметрах.

```
$ gcc -Wall -c foo.c
$ gcc -Wall -c main.c
```

Теперь мы связываем их вместе для создания нашего окончательного исполняемого файла:

```
$ gcc -o testprogram foo.o main.o
```

Использование глобальной переменной

Использование глобальных переменных обычно обескураживается. Это делает вашу программу более трудной для понимания и сложнее отлаживать. Но иногда использование глобальной переменной приемлемо.

global.h

```
#ifndef GLOBAL_DOT_H    /* This is an "include guard" */
#define GLOBAL_DOT_H

/**
 * This tells the compiler that g_myglobal exists somewhere.
 * Without "extern", this would create a new variable named
 * g_myglobal in _every file_ that included it. Don't miss this!
 */
extern int g_myglobal; /* _Declare_ g_myglobal, that is promise it will be _defined_ by
                        * some module. */

#endif /* GLOBAL_DOT_H */
```

global.c

```
#include "global.h" /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.
                     */

int g_myglobal;     /* _Define_ my_global. As living in global scope it gets initialised to 0
                     * on program start-up. */
```

main.c

```
#include "global.h"

int main(void)
{
    g_myglobal = 42;
    return 0;
}
```

См. Также Как использовать `extern` для обмена переменными между исходными файлами?

Использование глобальных констант

Заголовки могут использоваться, чтобы объявлять глобально используемые ресурсы только для чтения, например таблицы строк.

Объявите те в отдельном заголовке, который включается в любой файл (« *Блок переводов* »), который хочет их использовать. Удобно использовать один и тот же заголовок, чтобы объявить связанное перечисление для идентификации всех строковых ресурсов:

resources.h:

```
#ifndef RESOURCES_H
#define RESOURCES_H

typedef enum { /* Define a type describing the possible valid resource IDs. */
  RESOURCE_UNDEFINED = -1, /* To be used to initialise any EnumResourceID typed variable to be

                              marked as "not in use", "not in list", "undefined", wtf.
                              Will say un-initialised on application level, not on language
level. Initialised uninitialised, so to say ;-)
                              Its like NULL for pointers ;-)*/
  RESOURCE_UNKNOWN = 0,    /* To be used if the application uses some resource ID,
                              for which we do not have a table entry defined, a fall back in
                              case we _need_ to display something, but do not find anything
                              appropriate. */

  /* The following identify the resources we have defined: */
  RESOURCE_OK,
  RESOURCE_CANCEL,
  RESOURCE_ABORT,
  /* Insert more here. */

  RESOURCE_MAX /* The maximum number of resources defined. */
} EnumResourceID;


extern const char * const resources[RESOURCE_MAX]; /* Declare, promise to anybody who includes

                                      this, that at linkage-time this symbol will be around.
                                      The 1st const guarantees the strings will not change,
                                      the 2nd const guarantees the string-table entries
                                      will never suddenly point somewhere else as set during
                                      initialisation. */
#endif
```

Чтобы фактически определить ресурсы, созданные связанным .c-файлом, это еще одна единица перевода, содержащая фактические экземпляры того, что было объявлено в соответствующем файле заголовка (.h):

resources.c:

```
#include "resources.h" /* To make sure clashes between declaration and definition are
                          recognised by the compiler include the declaring header into
                          the implementing, defining translation unit (.c file).

/* Define the resources. Keep the promise made in resources.h. */
const char * const resources[RESOURCE_MAX] = {
  "<unknown>",
  "OK",
  "Cancel",
  "Abort"
};
```

Программа, использующая это, может выглядеть так:

main.c:

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h>

#include "resources.h"


int main(void)
{
  EnumResourceID resource_id = RESOURCE_UNDEFINED;

  while ((++resource_id) < RESOURCE_MAX)
  {
    printf("resource ID: %d, resource: '%s'\n", resource_id, resources[resource_id]);
  }

  return EXIT_SUCCESS;
}
```

Скомпилируйте три файла выше, используя GCC, и соедините их, чтобы стать `main` файлом программы, например, используя это:

```
gcc -Wall -Wextra -pedantic -Wconversion -g  main.c resources.c -o main
```

(используйте эту `-Wall -Wextra -pedantic -Wconversion` чтобы сделать компилятор действительно придирчивым, поэтому вы не пропустите ничего, прежде чем отправлять код в SO, скажете мир или даже поместите его в производство)

Запустить созданную программу:

```
$ ./main
```

И получить:

```
resource ID: 0, resource: '<unknown>'
resource ID: 1, resource: 'OK'
resource ID: 2, resource: 'Cancel'
resource ID: 3, resource: 'Abort'
```

Вступление

Пример объявлений:

```
int a; /* declaring single identifier of type int */
```

В приведенном выше объявлении объявляется один идентификатор с именем `a` который относится к некоторому объекту с типом `int` .

```
int a1, b1; /* declaring 2 identifiers of type int */
```

Второе объявление объявляет 2 идентификатора с именами `a1` и `b1` которые относятся к некоторым другим объектам, но с тем же типом `int` .

В принципе, способ, которым это работает, выглядит следующим образом: сначала вы помещаете какой-то тип , затем вы пишете одно или несколько выражений, разделенных через запятую (`,`) (**которые не будут оцениваться в этой точке), и которые в противном случае следует называть деклараторами в этот контекст**). При написании таких выражений вам разрешено применять только некоторые индексы (`*`), функции call (`()`) или индексы (или индексирования массива - `[]`) к некоторому идентификатору (вы также не можете использовать каких-либо операторов). Используемый идентификатор не требуется быть видимым в текущей области. Некоторые примеры:

```
/* 1 */ int /* 2 */ (*z) /* 3 */ , /* 4 */ *x , /* 5 */ **c /* 6 */ ;
```

#	Описание
1	Имя целочисленного типа.
2	Невыраженное выражение, применяющее косвенность к некоторому идентификатору `z` .
3	У нас есть запятая, указывающая, что еще одно выражение будет следовать в той же декларации.
4	Невыраженное выражение, применяющее косвенность к другому идентификатору `x` .
5	Невыраженное выражение, применяющее косвенность к значению выражения `(*c)`

#	Описание
	.
6	Конец объявления.

Обратите внимание: ни один из вышеперечисленных идентификаторов не был видимым до этого объявления, и поэтому используемые выражения не будут действительны перед ним.

После каждого такого выражения используемый в нем идентификатор вводится в текущую область. (Если идентификатор назначил ему привязку, он также может быть повторно объявлен с помощью того же типа связи, чтобы оба идентификатора ссылались на один и тот же объект или функцию)

Кроме того, для инициализации может использоваться знак равенства (=). Если за объявлением не оценено выражение (декларатор), то = внутри декларации - мы говорим, что вводимый идентификатор также инициализируется. После знака = мы можем снова добавить некоторое выражение, но на этот раз оно будет оценено, и его значение будет использоваться как начальное для объявленного объекта.

Примеры:

```
int l = 90; /* the same as: */

int l; l = 90; /* if it the declaration of l was in block scope */

int c = 2, b[c]; /* ok, equivalent to: */

int c = 2; int b[c];
```

Позже в вашем коде вам разрешено записывать то же самое выражение из части объявления нового введенного идентификатора, предоставляя вам объект типа, указанного в начале объявления, при условии, что вы присвоили действительные значения всем доступным объектов в пути. Примеры:

```
void f()
{
    int b2; /* you should be able to write later in your code b2
            which will directly refer to the integer object
            that b2 identifies */

    b2 = 2; /* assign a value to b2 */

    printf("%d", b2); /*ok - should print 2*/

    int *b3; /* you should be able to write later in your code *b3 */

    b3 = &b2; /* assign valid pointer value to b3 */

    printf("%d", *b3); /* ok - should print 2 */

    int **b4; /* you should be able to write later in your code **b4 */
```

```
    b4 = &b3;

    printf("%d", **b4); /* ok - should print 2 */

    void (*p)(); /* you should be able to write later in your code (*p)() */

    p = &f; /* assign a valid pointer value */

    (*p)(); /* ok - calls function f by retrieving the
            pointer value inside p -    p
            and dereferencing it -      *p
            resulting in a function
            which is then called -      (*p)() -

            it is not *p() because else first the () operator is
            applied to p and then the resulting void object is
            dereferenced which is not what we want here */
}
```

Объявление `b3` указывает, что вы можете использовать значение `b3` как среднее для доступа к некоторому целочисленному объекту.

Конечно, для применения косвенности (`*`) к `b3` , вы также должны иметь правильное значение, хранящееся в нем (подробнее см. Указатели). Вы также должны сначала сохранить некоторое значение в объект, прежде чем пытаться его извлечь (вы можете увидеть больше об этой проблеме здесь). Мы все это сделали в приведенных выше примерах.

```
int a3(); /* you should be able to call a3 */
```

Это говорит компилятору, что вы попытаетесь вызвать `a3` . В этом случае `a3` ссылается на функцию вместо объекта. Одна разница между объектом и функцией заключается в том, что функции всегда будут иметь какую-то связь. Примеры:

```
void f1()
{
    {
        int f2(); /* 1 refers to some function f2 */
    }

    {
        int f2(); /* refers to the exact same function f2 as (1) */
    }
}
```

В приведенном выше примере 2 объявления относятся к одной и той же функции `f2` , тогда как если бы они объявляли объекты, то в этом контексте (с двумя различными блочными областями) у них было бы два разных разных объекта.

```
int (*a3)(); /* you should be able to apply indirection to `a3` and then call it */
```

Теперь это может показаться сложным, но если вы знаете приоритет операторов, у вас будет 0 проблем с чтением вышеуказанной декларации. Скобки нужны, потому что оператор `*` имеет меньшее приоритет, чем `( )` .

В случае использования оператора подстроки результирующее выражение не будет действительно действительным после объявления, потому что индекс, используемый в нем (значение внутри `[` и `]`), всегда будет на 1 превышать максимально допустимое значение для этого объекта / функции.

```
int a4[5]; /* here a4 shouldn't be accessed using the index 5 later on */
```

Но он должен быть доступен всеми другими индексами ниже 5. Примеры:

```
a4[0], a4[1]; a4[4];
```

`a4[5]` приведет к UB. Более подробную информацию о массивах можно найти здесь .

```
int (*a5)[5](); /* here a4 could be applied indirection
                   indexed up to (but not including) 5
                   and called */
```

К сожалению для нас, хотя и синтаксически возможно, декларация `a5` запрещена действующим стандартом.

Typedef

Typedefs - это объявления с ключевым словом `typedef` впереди и перед типом. Например:

```
typedef int (*(*t0)())[5];
```

(*вы можете технически поставить typedef после типа тоже - как это* `int typedef (*(*t0)())[5];` *но это не рекомендуется*)

Вышеуказанные объявления объявляют идентификатор для имени typedef. Вы можете использовать его следующим образом:

```
t0 pf;
```

Что будет иметь такой же эффект, как и запись:

```
int (*(*pf)())[5];
```

Как вы можете видеть, имя typedef «сохраняет» объявление как тип, который будет использоваться позже для других объявлений. Таким образом вы можете сэкономить несколько нажатий клавиш. Также как объявление с использованием `typedef` все еще является объявлением, вы не ограничены только приведенным выше примером:

```
t0 (*pf1);
```

Такой же как:

```
int (*(**pf1)())[5];
```

Использование правого или левого правила для расшифровки декларации C

Правило «право-левое» является полностью регулярным правилом для расшифровки объявлений C. Он также может быть полезен при их создании.

Прочтите символы, когда вы встретите их в объявлении ...

```
*   as "pointer to"          - always on the left side
[]  as "array of"            - always on the right side
()  as "function returning"  - always on the right side
```

Как применить правило

ШАГ 1

Найдите идентификатор. Это ваша отправная точка. Тогда скажите себе: «Идентификатор есть». Вы начали свою декларацию.

ШАГ 2

Посмотрите на символы справа от идентификатора. Если, скажем, вы найдете `()` там, то вы знаете, что это объявление для функции. Таким образом, у вас тогда будет *« идентификатор возвращается функция»* . Или, если вы нашли там `[]` , вы бы сказали: *« Идентификатор - это массив»* . Продолжайте, пока не закончите символы ИЛИ не попадете в правую скобку `)` . (Если вы нажмете левую круглую скобку `(` это начало символа `()` , даже если между скобками есть вещи. Подробнее об этом ниже.)

ШАГ 3

Посмотрите на символы слева от идентификатора. Если это не один из наших символов выше (скажем, что-то вроде «int»), просто скажите это. В противном случае переведите его на английский, используя приведенную выше таблицу. Продолжайте движение до тех пор, пока не закончите символы ИЛИ не удалите левую скобку `(` .

Теперь повторите шаги 2 и 3, пока вы не сформируете свою декларацию.

Вот некоторые примеры:

```
int *p[];
```

Во-первых, найдите идентификатор:

```
int *p[];
     ^
```

"p"

Теперь двигайтесь прямо, пока из символов или правой скобки не попадет.

```
int *p[];
      ^^
```

"p является массивом"

Больше не может двигаться (из символов), поэтому переместите влево и найдите:

```
int *p[];
    ^
```

"p - массив указателя на"

Продолжайте движение влево и найдите:

```
int *p[];
^^^
```

«p - массив указателя на int».

(или *«p - массив, в котором каждый элемент имеет указатель типа на int»*)

Другой пример:

```
int *(*func())();
```

Найдите идентификатор.

```
int *(*func())();
       ^^^^
```

"func is"

Двигаться вправо.

```
int *(*func())();
           ^^
```

«func возвращает функцию»

Больше не может двигаться дальше из-за правильной круглой скобки, поэтому двигайтесь влево.

```
int *(*func())();
      ^
```

"func - это функция, возвращающая указатель на"

Больше не может двигаться влево из-за левой скобки, так что продолжайте идти вправо.

```
int *(*func())();
              ^^
```

«func - это функция, возвращающая указатель на функцию возврата»

Не могу больше двигаться, потому что у нас нет символов, поэтому идите налево.

```
int *(*func())();
    ^
```

"func - это функция, возвращающая указатель на функцию, возвращающую указатель на"

И, наконец, продолжайте идти, потому что справа ничего не осталось.

```
int *(*func())();
^^^
```

«func - это функция, возвращающая указатель на функцию, возвращающую указатель на int».

Как вы можете видеть, это правило может быть весьма полезным. Вы также можете использовать его для проверки здравомыслия во время создания объявлений и дать вам подсказку о том, где поставить следующий символ, и нужны ли скобки.

Некоторые декларации выглядят намного сложнее, чем из-за размеров массивов и списков аргументов в прототипе. Если вы видите `[3]` , это читается как *«массив (размер 3) ...»* . Если вы видите `(char *,int)` который читается как * "function expecting (char *, int) и возврат ..."* .

Вот весело:

```
int (*(*fun_one)(char *,double))[9][20];
```

Я не буду проходить каждый шаг, чтобы расшифровать этот.

* «fun_one - это указатель на функцию expecting (char *, double) и возвращающий указатель*

на массив (размер 9) массива (размер 20) для int».

Как вы можете видеть, это не так сложно, если вы избавитесь от размеров массивов и списков аргументов:

```
int (*(*fun_one)())[][];
```

Вы можете расшифровать его таким образом, а затем поместить в массивы размеры и списки аргументов позже.

Некоторые заключительные слова:

Вполне возможно сделать незаконные объявления, используя это правило, поэтому необходимо знать некоторые из них, что является законным в C. Например, если выше было:

```
int *((*fun_one)())[][];
```

он бы прочитал *«fun_one - это указатель на функцию, возвращающую массив массива указателя на int»* . Поскольку функция не может вернуть массив, но только указатель на массив, это объявление является незаконным.

Незаконные комбинации включают:

```
[]() - cannot have an array of functions
()() - cannot have a function that returns a function
()[] - cannot have a function that returns an array
```

Во всех вышеперечисленных случаях вам понадобится набор круглых скобок для привязки символа `*` слева от этих `()` и `[]` правых символов, чтобы объявление было законным.

Вот еще несколько примеров:

легальный

```
int i;              an int
int *p;             an int pointer (ptr to an int)
int a[];            an array of ints
int f();            a function returning an int
int **pp;           a pointer to an int pointer (ptr to a ptr to an int)
int (*pa)[];        a pointer to an array of ints
int (*pf)();        a pointer to a function returning an int
int *ap[];          an array of int pointers (array of ptrs to ints)
int aa[][];         an array of arrays of ints
int *fp();          a function returning an int pointer
int ***ppp;         a pointer to a pointer to an int pointer
int (**ppa)[];      a pointer to a pointer to an array of ints
int (**ppf)();      a pointer to a pointer to a function returning an int
```

```
int *(*pap)[];       a pointer to an array of int pointers
int (*paa)[][];      a pointer to an array of arrays of ints
int *(*pfp)();       a pointer to a function returning an int pointer
int **app[];         an array of pointers to int pointers
int (*apa[])[];      an array of pointers to arrays of ints
int (*apf[])();      an array of pointers to functions returning an int
int *aap[][];        an array of arrays of int pointers
int aaa[][][];       an array of arrays of arrays of int
int **fpp();         a function returning a pointer to an int pointer
int (*fpa())[];      a function returning a pointer to an array of ints
int (*fpf())();      a function returning a pointer to a function returning an int
```

нелегальный

```
int af[]();          an array of functions returning an int
int fa()[];          a function returning an array of ints
int ff()();          a function returning a function returning an int
int (*pfa)()[];      a pointer to a function returning an array of ints
int aaf[][]();       an array of arrays of functions returning an int
int (*paf)[]();      a pointer to a an array of functions returning an int
int (*pff)()();      a pointer to a function returning a function returning an int
int *afp[]();        an array of functions returning int pointers
int afa[]()[];       an array of functions returning an array of ints
int aff[]()();       an array of functions returning functions returning an int
int *fap()[];        a function returning an array of int pointers
int faa()[][];       a function returning an array of arrays of ints
int faf()[]();       a function returning an array of functions returning an int
int *ffp()();        a function returning a function returning an int pointer
```

Источник: http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html

Прочитайте Объявления онлайн: https://riptutorial.com/ru/c/topic/3729/объявления

глава 33: Обычные подводные камни

Вступление

В этом разделе обсуждаются некоторые распространенные ошибки, о которых программист C должен знать и чего следует избегать. Подробнее о некоторых неожиданных проблемах и их причинах см. В разделе Неопределенное поведение

Examples

Смешение целых чисел без знака в арифметических операциях

Это, как правило , не очень хорошая идея , чтобы смешивать `signed` и `unsigned` целые числа в арифметических операциях. Например, что будет выводиться из следующего примера?

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1000;
    signed int b = -1;

    if (a > b) puts("a is more than b");
    else puts("a is less or equal than b");

    return 0;
}
```

Поскольку 1000 больше, чем -1, вы ожидаете, что результат будет `a is more than b` , однако этого не произойдет.

Арифметические операции между различными интегральными типами выполняются в общем типе, определяемом так называемыми обычными арифметическими преобразованиями (см. Спецификацию языка, 6.3.1.8).

В этом случае «общий тип» является `unsigned int` , потому что, как указано в обычных арифметических преобразованиях ,

> 714 В противном случае, если операнд с целым типом без знака имеет ранг, больший или равный рану типа другого операнда, тогда операнд со знаком целочисленного типа преобразуется в тип операнда с целым типом без знака.

Это означает, что `int` operand `b` будет преобразован в `unsigned int` перед сравнением.

Когда -1 преобразуется в `unsigned int` результатом является максимально возможное значение `unsigned int` , которое больше 1000, что означает, что `a > b` является ложным.

Ошибочно писать = вместо == при сравнении

Оператор `=` используется для назначения.

Для сравнения используется оператор `==` .

Нужно быть осторожным, чтобы не смешивать их. Иногда ошибочно пишет

```
/* assign y to x */
if (x = y) {
     /* logic */
}
```

когда действительно нужно:

```
/* compare if x is equal to y */
if (x == y) {
    /* logic */
}
```

Первый присваивает значение y x и проверяет, не является ли это значение ненулевым, вместо сравнения, что эквивалентно:

```
if ((x = y) != 0) {
    /* logic */
}
```

Бывают случаи, когда тестирование результата присвоения предназначено и обычно используется, поскольку оно позволяет избежать дублирования кода и его обработки в первый раз. сравнить

```
while ((c = getopt_long(argc, argv, short_options, long_options, &option_index)) != -1) {
        switch (c) {
        ...
        }
}
```

против

```
c = getopt_long(argc, argv, short_options, long_options, &option_index);
while (c != -1) {
        switch (c) {
        ...
        }
        c = getopt_long(argc, argv, short_options, long_options, &option_index);
}
```

Современные компиляторы распознают этот шаблон и не предупреждают, когда назначение находится в круглых скобках, как указано выше, но может предупреждать о

других применениях. Например:

```
if (x = y)          /* warning */

if ((x = y))        /* no warning */
if ((x = y) != 0)   /* no warning; explicit */
```

Некоторые программисты используют стратегию размещения константы слева от оператора (обычно называемые условиями Йоды). Поскольку константы являются значениями rvalues, этот стиль условия заставит компилятор выкинуть ошибку, если использовался неправильный оператор.

```
if (5 = y) /* Error */

if (5 == y) /* No error */
```

Однако это значительно снижает читаемость кода и не считается необходимым, если программист следует хорошей практике кодирования C и не помогает при сравнении двух переменных, поэтому он не является универсальным решением. Кроме того, многие современные компиляторы могут давать предупреждения, когда код написан с условиями Yoda.

Неверное использование точек с запятой

Будьте осторожны с точкой с запятой. Следующий пример

```
if (x > a);
   a = x;
```

фактически означает:

```
if (x > a) {}
a = x;
```

что означает `x` будут назначены в любом случае, который не может быть то , что вы хотели изначально. `a`

Иногда отсутствие точки с запятой также вызывает незаметную проблему:

```
if (i < 0)
    return
day = date[0];
hour = date[1];
minute = date[2];
```

Точка с запятой позади пропущена, так что день = дата [0] будет возвращена.

Один из способов избежать этой и подобных проблем - всегда использовать фигурные скобки для многострочных условных и циклов. Например:

```
if (x > a) {
    a = x;
}
```

Забыв выделить один дополнительный байт для \ 0

Когда вы копируете строку в буфер `malloc` ed, всегда помните, чтобы добавить 1 в `strlen` .

```
char *dest = malloc(strlen(src)); /* WRONG */
char *dest = malloc(strlen(src) + 1); /* RIGHT */

strcpy(dest, src);
```

Это связано с тем, что `strlen` не включает длину `\0` в длину. Если вы примете метод `WRONG` (как показано выше), при вызове `strcpy` ваша программа будет ссылаться на неопределенное поведение.

Это также относится к ситуациям, когда вы читаете строку известной максимальной длины из `stdin` или другого источника. Например

```
#define MAX_INPUT_LEN 42

char buffer[MAX_INPUT_LEN]; /* WRONG */
char buffer[MAX_INPUT_LEN + 1]; /* RIGHT */

scanf("%42s", buffer);  /* Ensure that the buffer is not overflowed */
```

Забывая освободить память (утечки памяти)

Лучшая практика программирования - освободить любую память, которая была выделена непосредственно вашим собственным кодом, или неявно, вызывая внутреннюю или внешнюю функцию, такую как библиотечный API, такой как `strdup()` . Неспособность освободить память может привести к утечке памяти, которая может накапливаться в значительном количестве потерянной памяти, которая недоступна вашей программе (или системе), что может привести к сбоям или неопределенному поведению. Проблемы чаще возникают, если утечка возникает неоднократно в цикле или рекурсивной функции. Риск сбоя программы увеличивает время простоя программы. Иногда проблемы появляются мгновенно; в других случаях проблемы не будут наблюдаться в течение нескольких часов или даже лет постоянной работы. Ошибки исчерпания памяти могут быть катастрофическими, в зависимости от обстоятельств.

Следующий бесконечный цикл является примером утечки, которая в конечном итоге исчерпает доступную утечку памяти, вызвав функцию `getline()` , которая неявно выделяет новую память, не освобождая эту память.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    /* The loop below leaks memory as fast as it can */

    for(;;) {
        getline(&line, &size, stdin); /* New memory implicitly allocated */

        /* <do whatever> */

        line = NULL;
    }

    return 0;
}
```

Напротив, в приведенном ниже коде также используется функция `getline()` , но на этот раз выделенная память правильно освобождается, избегая утечки.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    for(;;) {
        if (getline(&line, &size, stdin) < 0) {
            free(line);
            line = NULL;

            /* Handle failure such as setting flag, breaking out of loop and/or exiting */
        }

        /* <do whatever> */

        free(line);
        line = NULL;

    }

    return 0;
}
```

Утечка памяти не всегда имеет ощутимые последствия и не обязательно является функциональной проблемой. В то время как «лучшая практика» диктует строгому освобождению памяти в стратегических точках и условиях, уменьшает объем памяти и снижает риск исчерпания памяти, могут быть исключения. Например, если программа ограничена по продолжительности и объему, риск сбоя распределения может считаться слишком маленьким, чтобы беспокоиться. В этом случае обход явного освобождения

может считаться приемлемым. Например, большинство современных операционных систем автоматически освобождают всю память, потребляемую программой, когда она завершается, из-за сбоя программы, системного вызова `exit()` , завершения процесса или достижения конца `main()` . Явное освобождение памяти в момент неминуемой остановки программы может фактически быть избыточным или ввести штраф за производительность.

Выделение может завершиться неудачно, если недостаточно памяти, и обработка сбоев должна учитываться на соответствующих уровнях стека вызовов. `getline()` , показанный выше, является интересным прецедентом, потому что он является библиотечной функцией, которая не только выделяет память, которую она оставляет для вызывающего абонента, но и может не работать по ряду причин, и все это необходимо учитывать. Поэтому при использовании C API важно прочитать *документацию (справочную страницу)* и обратить особое внимание на условия ошибки и использование памяти, а также знать, какой программный уровень несет бремя освобождения возвращенной памяти.

Еще одна распространенная практика обработки памяти состоит в том, чтобы последовательно устанавливать указатели на память в NULL сразу после освобождения памяти, на которую ссылаются эти указатели, поэтому эти указатели могут быть проверены на достоверность в любое время (например, для NULL / non-NULL), поскольку доступ к свободной памяти может привести к серьезным проблемам, таким как получение данных мусора (операция чтения) или повреждение данных (операция записи) и / или сбой программы. В большинстве современных операционных систем освобождение ячейки памяти 0 (`NULL`) является NOP (например, безвредным), как того требует стандарт C, поэтому, устанавливая указатель на NULL, нет риска двойной освобождающей памяти, если указатель передается в `free()` . Имейте в виду, что память с двойным освобождением может привести к очень много времени, запутыванию и *затруднению диагностики* сбоев.

Копирование слишком много

```
char buf[8]; /* tiny buffer, easy to overflow */

printf("What is your name?\n");
scanf("%s", buf); /* WRONG */
scanf("%7s", buf); /* RIGHT */
```

Если пользователь вводит строку длиной более 7 символов (- 1 для нулевого терминатора), память за буфером `buf` будет перезаписана. Это приводит к неопределенному поведению. Вредоносные хакеры часто используют это, чтобы перезаписать обратный адрес и изменить его на адрес вредоносного кода хакера.

Забыв скопировать возвращаемое значение realloc во временную

Если `realloc` не работает, он возвращает `NULL` . Если вы присвоите значение исходного буфера возвращаемому значению `realloc` , и если он вернет `NULL` , то исходный буфер

(старый указатель) будет потерян, что приведет к *утечке памяти* . Решение состоит в том, чтобы скопировать во временный указатель, и если это временное значение не является NULL, **то** скопируйте его в реальный буфер.

```
char *buf, *tmp;

buf = malloc(...);
...

/* WRONG */
if ((buf = realloc(buf, 16)) == NULL)
    perror("realloc");

/* RIGHT */
if ((tmp = realloc(buf, 16)) != NULL)
    buf = tmp;
else
    perror("realloc");
```

Сравнение чисел с плавающей запятой

Типы плавающей точки (`float` , `double` и `long double`) не могут точно представлять некоторые числа, потому что они имеют конечную точность и представляют значения в двоичном формате. Подобно тому, как мы повторяем десятичные числа в базе 10 для фракций, таких как 1/3, существуют фракции, которые также не могут быть представлены конечно в двоичном виде (например, 1/3, но также, что более важно, 1/10). Не сравнивать результаты с плавающей запятой напрямую; вместо этого используйте дельта.

```
#include <float.h> // for DBL_EPSILON and FLT_EPSILON
#include <math.h>  // for fabs()

int main(void)
{
    double a = 0.1; // imprecise: (binary) 0.000110...

    // may be false or true
    if (a + a + a + a + a + a + a + a + a + a == 1.0) {
        printf("10 * 0.1 is indeed 1.0. This is not guaranteed in the general case.\n");
    }

    // Using a small delta value.
    if (fabs(a + a + a + a + a + a + a + a + a + a - 1.0) < 0.000001) {
        // C99 5.2.4.2.2p8 guarantees at least 10 decimal digits
        // of precision for the double type.
        printf("10 * 0.1 is almost 1.0.\n");
    }

    return 0;
}
```

Другой пример:

```
gcc -O3   -g   -I./inc   -std=c11   -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-
prototypes  -Wold-style-definition       rd11.c -o rd11 -L./lib -lsoq
```

```
#include <stdio.h>
#include <math.h>

static inline double rel_diff(double a, double b)
{
    return fabs(a - b) / fmax(fabs(a), fabs(b));
}

int main(void)
{
    double d1 = 3.14159265358979;
    double d2 = 355.0 / 113.0;

    double epsilon = 1.0;
    for (int i = 0; i < 10; i++)
    {
        if (rel_diff(d1, d2) < epsilon)
            printf("%d:%.10f <=> %.10f within tolerance %.10f (rel diff %.4E)\n",
                   i, d1, d2, epsilon, rel_diff(d1, d2));
        else
            printf("%d:%.10f <=> %.10f out of tolerance %.10f (rel diff %.4E)\n",
                   i, d1, d2, epsilon, rel_diff(d1, d2));
        epsilon /= 10.0;
    }
    return 0;
}
```

Выход:

```
0:3.1415926536 <=> 3.1415929204 within tolerance 1.0000000000 (rel diff 8.4914E-08)
1:3.1415926536 <=> 3.1415929204 within tolerance 0.1000000000 (rel diff 8.4914E-08)
2:3.1415926536 <=> 3.1415929204 within tolerance 0.0100000000 (rel diff 8.4914E-08)
3:3.1415926536 <=> 3.1415929204 within tolerance 0.0010000000 (rel diff 8.4914E-08)
4:3.1415926536 <=> 3.1415929204 within tolerance 0.0001000000 (rel diff 8.4914E-08)
5:3.1415926536 <=> 3.1415929204 within tolerance 0.0000100000 (rel diff 8.4914E-08)
6:3.1415926536 <=> 3.1415929204 within tolerance 0.0000010000 (rel diff 8.4914E-08)
7:3.1415926536 <=> 3.1415929204 within tolerance 0.0000001000 (rel diff 8.4914E-08)
8:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000100 (rel diff 8.4914E-08)
9:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000010 (rel diff 8.4914E-08)
```

Выполнение дополнительного масштабирования в арифметике указателя

В арифметике указателя целое число, добавляемое или вычитаемое в указатель, интерпретируется не как изменение *адреса,* а как количество перемещаемых *элементов* .

```
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + sizeof(int) * 2; /* wrong */
    printf("%d %d\n", *ptr, *ptr2);
    return 0;
}
```

Этот код делает дополнительное масштабирование при вычислении указателя,

назначенного `ptr2` . Если `sizeof(int)` равно 4, что характерно для современных 32-битных сред, выражение означает «8 элементов после `array[0]` », которое выходит за пределы диапазона, и вызывает *неопределенное поведение* .

Чтобы `ptr2` указывал на то, что является 2 элементами после `array[0]` , вы должны просто добавить 2.

```
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + 2;
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}
```

Явная арифметика указателя с использованием аддитивных операторов может вводить в заблуждение, поэтому использование подтипов массива может быть лучше.

```
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = &ptr[2];
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}
```

`E1[E2]` идентичен `(*((E1)+(E2)))` (N1570 6.5.2.1, параграф 2) и `&(E1[E2])` эквивалентен `((E1)+(E2))` (N 1570 6.5.3.2, сноска 102).

В качестве альтернативы, если предпочтительна арифметика указателей, то приведение указателя к другому типу данных позволяет разрешить байтовую адресацию. Будьте осторожны: endianness может стать проблемой, а отбрасывание на типы, отличные от «указателя на символ», приводит к строгим проблемам псевдонимов .

```
#include <stdio.h>

int main(void) {
    int array[3] = {1,2,3};  // 4 bytes * 3 allocated
    unsigned char *ptr = (unsigned char *) array;  // unsigned chars only take 1 byte
    /*
     * Now any pointer arithmetic on ptr will match
     * bytes in memory.  ptr can be treated like it
     * was declared as: unsigned char ptr[12];
     */

    return 0;
}
```

Макросы - это простые замены строк

Макросы - это простые замены строк. (Строго говоря, они работают с токенами предварительной обработки, а не с произвольными строками).

```
#include <stdio.h>

#define SQUARE(x) x*x

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

Вы можете ожидать, что этот код напечатает `9` (`3*3`), но на самом деле будет напечатано `5` , потому что макрос будет расширен до `1+2*1+2` .

Вы должны обернуть аргументы и все макрокоманды в круглые скобки, чтобы избежать этой проблемы.

```
#include <stdio.h>

#define SQUARE(x) ((x)*(x))

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

Другая проблема заключается в том, что аргументы макроса не гарантируются для оценки один раз; они не могут быть оценены вообще или могут быть оценены несколько раз.

```
#include <stdio.h>

#define MIN(x, y) ((x) <= (y) ? (x) : (y))

int main(void) {
    int a = 0;
    printf("%d\n", MIN(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

В этом коде макрос будет расширен до `((a++) <= (10) ? (a++) : (10))` . Так `a++` (`0`) меньше `10` , `a++` будет оцениваться дважды, и он заставит значение `a` и то, что возвращается из `MIN` отличается от того, что вы можете ожидать.

Этого можно избежать, используя функции, но обратите внимание, что типы будут фиксироваться определением функции, тогда как макросы могут быть (тоже) гибкими с типами.

```
#include <stdio.h>

int min(int x, int y) {
    return x <= y ? x : y;
}

int main(void) {
    int a = 0;
    printf("%d\n", min(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

Теперь проблема двойной оценки фиксирована, но эта функция `min` не может обрабатывать `double` данные без усечения, например.

Директивы макросов могут быть двух типов:

```
#define OBJECT_LIKE_MACRO     followed by a "replacement list" of preprocessor tokens
#define FUNCTION_LIKE_MACRO(with, arguments) followed by a replacement list
```

То, что отличает эти два типа макросов, - это символ, который следует за идентификатором после `#define` : если это *lparen* , это функционально-подобный макрос; в противном случае это объект-подобный макрос. Если намерение состоит в том, чтобы написать функцию, как макрос, там не должно быть белое пространство между концом имени макроса и `(` . Проверьте это для подробного объяснения.

C99

В C99 или более поздней версии вы можете использовать `static inline int min(int x, int y) { … }` .

C11

В C11 вы можете написать выражение типа «type-generic» для `min` .

```
#include <stdio.h>

#define min(x, y) _Generic((x), \
                        long double: min_ld, \
                        unsigned long long: min_ull, \
                        default: min_i \
                        )(x, y)

#define gen_min(suffix, type) \
    static inline type min_##suffix(type x, type y) { return (x < y) ? x : y; }

gen_min(ld, long double)
gen_min(ull, unsigned long long)
gen_min(i, int)

int main(void)
{
    unsigned long long ull1 = 50ULL;
```

```
    unsigned long long ull2 = 37ULL;
    printf("min(%llu, %llu) = %llu\n", ull1, ull2, min(ull1, ull2));
    long double ld1 = 3.141592653L;
    long double ld2 = 3.141592652L;
    printf("min(%.10Lf, %.10Lf) = %.10Lf\n", ld1, ld2, min(ld1, ld2));
    int i1 = 3141653;
    int i2 = 3141652;
    printf("min(%d, %d) = %d\n", i1, i2, min(i1, i2));
    return 0;
}
```

Общее выражение может быть расширено с помощью большего количества типов, таких как `double` , `float` , `long long` , `unsigned long` , `long` , `unsigned` - и соответствующие `gen_min` макросов `gen_min` .

Неопределенные опорные ошибки при связывании

Одна из наиболее распространенных ошибок в компиляции происходит на этапе связывания. Ошибка выглядит примерно так:

```
$ gcc undefined_reference.c
/tmp/ccoXhwF0.o: In function `main':
undefined_reference.c:(.text+0x15): undefined reference to `foo'
collect2: error: ld returned 1 exit status
$
```

Итак, давайте посмотрим на код, который породил эту ошибку:

```
int foo(void);

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

Мы видим здесь *объявление* foo (`int foo();`), но не *определение* его (фактическая функция). Таким образом , мы обеспечили компилятор с заголовком функции, но не было никакой такой функции , определенная в любом месте, так что проходит этап компиляции , но линкер выходит с `Undefined reference` ошибкой.
Чтобы исправить эту ошибку в нашей маленькой программе, нам нужно было бы добавить *определение* для foo:

```
/* Declaration of foo */
int foo(void);

/* Definition of foo */
int foo(void)
{
    return 5;
}
```

```
int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

Теперь этот код будет скомпилирован. Возникает альтернативная ситуация, когда источник для `foo()` находится в отдельном исходном файле `foo.c` (и есть заголовок `foo.h` чтобы объявить `foo()` который включен как в `foo.c` и `undefined_reference.c`). Затем исправление должно связывать как объектный файл с файлами `foo.c` и `undefined_reference.c` или компилировать оба исходных файла:

```
$ gcc -c undefined_reference.c
$ gcc -c foo.c
$ gcc -o working_program undefined_reference.o foo.o
$
```

Или же:

```
$ gcc -o working_program undefined_reference.c foo.c
$
```

Более сложным является случай, когда библиотеки участвуют, как в коде:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv)
{
    double first;
    double second;
    double power;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <denom> <nom>\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* Translate user input to numbers, extra error checking
     * should be done here. */
    first = strtod(argv[1], NULL);
    second = strtod(argv[2], NULL);

    /* Use function pow() from libm - this will cause a linkage
     * error unless this code is compiled against libm! */
    power = pow(first, second);

    printf("%f to the power of %f = %f\n", first, second, power);

    return EXIT_SUCCESS;
}
```

Код синтаксически правильный, объявление для `pow()` существует из `#include <math.h>` , поэтому мы пытаемся скомпилировать и связать, но получим такую ошибку:

```
$ gcc no_library_in_link.c -o no_library_in_link
/tmp/ccduQQqA.o: In function `main':
no_library_in_link.c:(.text+0x8b): undefined reference to `pow'
collect2: error: ld returned 1 exit status
$
```

Это происходит потому, что *определение* для `pow()` не было найдено на этапе связывания. Чтобы исправить это, мы должны указать, что мы хотим связать с математической библиотекой `libm` , указав флаг `-lm` . (Обратите внимание, что существуют такие платформы, как macOS, где `-lm` не требуется, но когда вы получаете неопределенную ссылку, необходима библиотека.)

Итак, мы снова запускаем этап компиляции, на этот раз определяя библиотеку (после исходных или объектных файлов):

```
$ gcc no_library_in_link.c -lm -o library_in_link_cmd
$ ./library_in_link_cmd 2 4
2.000000 to the power of 4.000000 = 16.000000
$
```

И это работает!

Неверное разложение массива

Общей проблемой в коде, которая использует многомерные массивы, массивы указателей и т. Д., Является то, что `Type**` и `Type[M][N]` являются принципиально разными типами:

```
#include <stdio.h>

void print_strings(char **strings, size_t n)
{
    size_t i;
    for (i = 0; i < n; i++)
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(s, 4);
    return 0;
}
```

Пример вывода компилятора:

```
file1.c: In function 'main':
file1.c:13:23: error: passing argument 1 of 'print_strings' from incompatible pointer type [-
Wincompatible-pointer-types]
```

```
        print_strings(strings, 4);
                      ^
file1.c:3:10: note: expected 'char **' but argument is of type 'char (*)[20]'
     void print_strings(char **strings, size_t n)
```

Ошибка указывает, что массив `s` в `main` функции передается функции `print_strings` , которая ожидает другого типа указателя, чем полученная. Он также содержит примечание, выражающее тип, ожидаемый `print_strings` и тип, который был передан ему из `main` .

Проблема связана с чем-то, называемым *распадом массива* . Что происходит, когда `s` с типом `char[4][20]` (массив из 4 массивов из 20 символов) передается функции, он превращается в указатель на свой первый элемент, как если бы вы написали `&s[0]` , который имеет тип `char (*)[20]` (указатель на 1 массив из 20 символов). Это происходит для любого массива, включая массив указателей, массив массивов массивов (3-D массивов) и массив указателей на массив. Ниже приведена таблица, иллюстрирующая, что происходит, когда массив распадается. Изменения в описании типа выделены, чтобы проиллюстрировать, что происходит:

Перед распадом		**После распада**	
`char [20]`	**массив (20 символов)**	`char *`	**указатель на (1 символ)**
`char [4][20]`	**массив (4 массива** из 20 символов)	`char (*)[20]`	**указатель на (1 массив** из 20 символов)
`char *[4]`	**массив (4 указателя** на 1 символ)	`char **`	**указатель на (1 указатель** на 1 символ)
`char [3][4][20]`	**массив (3 массива** из 4 массивов из 20 символов)	`char (*)[4][20]`	**указатель на (1 массив** из 4 массивов из 20 символов)
`char (*[4])[20]`	**массив (4 указателя** на 1 массив из 20 символов)	`char (**)[20]`	**указатель на (1 указатель** на 1 массив из 20 символов)

Если массив может распадаться на указатель, то можно сказать, что указатель может считаться массивом по меньшей мере из 1 элемента. Исключением является нулевой указатель, который ничего не указывает и, следовательно, не является массивом.

Распад массива происходит только один раз. Если массив разложился на указатель, он теперь является указателем, а не массивом. Даже если у вас есть указатель на массив, помните, что указатель может считаться массивом хотя бы одного элемента, поэтому разложение массива уже произошло.

Другими словами, указатель на массив (`char (*)[20]`) никогда не станет указателем на указатель (`char **`). Чтобы исправить функцию `print_strings` , просто `print_strings` ее

получить правильный тип:

```
void print_strings(char (*strings)[20], size_t n)
/* OR */
void print_strings(char strings[][20], size_t n)
```

Проблема возникает, когда вы хотите, `print_strings` функция `print_strings` была общей для любого массива символов: что, если есть 30 символов вместо 20? Или 50? Ответ заключается в том, чтобы добавить еще один параметр перед параметром массива:

```
#include <stdio.h>

/*
 * Note the rearranged parameters and the change in the parameter name
 * from the previous definitions:
 *      n (number of strings)
 *   => scount (string count)
 *
 * Of course, you could also use one of the following highly recommended forms
 * for the `strings` parameter instead:
 *
 *    char strings[scount][ccount]
 *    char strings[][ccount]
 */
void print_strings(size_t scount, size_t ccount, char (*strings)[ccount])
{
    size_t i;
    for (i = 0; i < scount; i++)
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(4, 20, s);
    return 0;
}
```

Компиляция не приводит к ошибкам и приводит к ожидаемому результату:

```
Example 1
Example 2
Example 3
Example 4
```

Передача несвязанных массивов в функции, ожидающие «реальных» многомерных массивов

При распределении многомерных массивов с помощью `malloc` , `calloc` и `realloc` общий шаблон заключается в распределении внутренних массивов с несколькими вызовами (даже если вызов появляется только один раз, он может находиться в цикле):

```
/* Could also be `int **` with malloc used to allocate outer array. */
```

```
int *array[4];
int i;

/* Allocate 4 arrays of 16 ints. */
for (i = 0; i < 4; i++)
    array[i] = malloc(16 * sizeof(*array[i]));
```

Разница в байтах между последним элементом одного из внутренних массивов и первым элементом следующего внутреннего массива может быть не 0, поскольку они были бы с «реальным» многомерным массивом (например, `int array[4][16];`) :

```
/* 0x40003c, 0x402000 */
printf("%p, %p\n", (void *)(array[0] + 15), (void *)array[1]);
```

Принимая во внимание размер `int` , вы получаете разницу в 8128 байт (8132-4), что составляет 2032 элемента массива `int` -sized, и это проблема: «реальный» многомерный массив не имеет пробелов между элементами.

Если вам нужно использовать динамически выделенный массив с функцией, ожидающей «реального» многомерного массива, вы должны выделить объект типа `int *` и использовать арифметику для выполнения вычислений:

```
void func(int M, int N, int *array);
...

/* Equivalent to declaring `int array[M][N] = {{0}};` and assigning to array4_16[i][j]. */
int *array;
int M = 4, N = 16;
array = calloc(M, N * sizeof(*array));
array[i * N + j] = 1;
func(M, N, array);
```

Если `N` является макросом или целым литералом, а не переменной, код может просто использовать более естественную двухмерную запись массива после выделения указателя на массив:

```
void func(int M, int N, int *array);
#define N 16
void func_N(int M, int (*array)[N]);
...

int M = 4;
int (*array)[N];
array = calloc(M, sizeof(*array));
array[i][j] = 1;

/* Cast to `int *` works here because `array` is a single block of M*N ints with no gaps,
   just like `int array2[M * N];` and `int array3[M][N];` would be. */
func(M, N, (int *)array);
func_N(M, array);
```

C99

Если `N` не является макросом или целым литералом, то `array` укажет на `array` переменной длины (VLA). Это все еще можно использовать с `func` путем литья в `int *` а новая функция `func_vla` заменит `func_N` :

```
void func(int M, int N, int *array);
void func_vla(int M, int N, int array[M][N]);
...

int M = 4, N = 16;
int (*array)[N];
array = calloc(M, sizeof(*array));
array[i][j] = 1;
func(M, N, (int *)array);
func_vla(M, N, array);
```

C11

Примечание : VLA являются необязательными с C11. Если ваша реализация поддерживает C11 и определяет макрос `__STDC_NO_VLA__` до 1, вы застряли в методах pre-C99.

Использование символьных констант вместо строковых литералов и наоборот

В C символьные константы и строковые литералы - это разные вещи.

Персонаж, окруженный одинарными кавычками типа `'a'` является *символьной константой* . Символьная константа представляет собой целое число, значение которого является символьным кодом, обозначающим символ. Как интерпретировать символьные константы с несколькими символами, такими как `'abc'` определяется реализацией.

Ноль или более символов, окруженных двойными кавычками типа `"abc"` является *строковым литералом* . Строковый литерал - это немодифицируемый массив, чьи элементы являются `char` типа. Строка в двойных кавычках плюс завершающий нуль-символ - это содержимое, поэтому `"abc"` имеет 4 элемента (`{'a', 'b', 'c', '\0'}`)

В этом примере используется символьная константа, в которой должен использоваться строковый литерал. Эта константа символа будет преобразована в указатель в определенном реализацией образом, и мало шансов на то, что преобразованный указатель будет действительным, поэтому этот пример вызовет *неопределенное поведение* .

```
#include <stdio.h>

int main(void) {
    const char *hello = 'hello, world'; /* bad */
    puts(hello);
    return 0;
}
```

В этом примере используется строковый литерал, в котором должна использоваться символьная константа. Указатель, преобразованный из строкового литерала, будет преобразован в целое число в соответствии с реализацией, и он будет преобразован в `char` в соответствии с реализацией. (Как преобразовать целое число в подписанный тип, который не может представлять значение для преобразования, определено в соответствии с реализацией, а также подписан ли `char` , также определяется реализацией.) Выход будет какой-то бессмысленной вещью.

```
#include <stdio.h>

int main(void) {
    char c = "a"; /* bad */
    printf("%c\n", c);
    return 0;
}
```

Почти во всех случаях компилятор будет жаловаться на эти путаницы. Если это не так, вам нужно использовать дополнительные параметры предупреждения компилятора или рекомендуется использовать лучший компилятор.

Игнорирование возвращаемых значений функций библиотеки

Почти каждая функция в стандартной библиотеке C возвращает что-то об успехе, а что-то еще об ошибке. Например, `malloc` вернет указатель на блок памяти, выделенный функцией при успешном завершении, и, если функции не удалось выделить запрошенный блок памяти, нулевой указатель. Поэтому вы всегда должны проверить возвращаемое значение для более легкой отладки.

Это плохо:

```
char* x = malloc(100000000000UL * sizeof *x);
/* more code */
scanf("%s", x); /* This might invoke undefined behaviour and if lucky causes a segmentation
violation, unless your system has a lot of memory */
```

Это хорошо:

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char* x = malloc(100000000000UL * sizeof *x);
    if (x == NULL) {
        perror("malloc() failed");
        exit(EXIT_FAILURE);
    }

    if (scanf("%s", x) != 1) {
        fprintf(stderr, "could not read string\n");
        free(x);
```

```
        exit(EXIT_FAILURE);
    }

    /* Do stuff with x. */

    /* Clean up. */
    free(x);

    return EXIT_SUCCESS;
}
```

Таким образом, вы сразу знаете причину ошибки, иначе вы можете потратить часы на поиск ошибки в совершенно неправильном месте.

Символ новой строки не потребляется при типичном вызове scanf ()

Когда эта программа

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;

    scanf("%d", &num);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

выполняется с этим вводом

```
42
life
```

выход будет `42 ""` вместо ожидаемого `42 "life"` .

Это связано с тем, что символ newline после `42` не потребляется при вызове `scanf()` и он потребляется `fgets()` прежде чем он считывает информацию о `life` . Затем `fgets()` перестает читать перед чтением `life` .

Чтобы избежать этой проблемы, один из способов, который полезен, когда максимальная длина строки известна, - например, при решении проблем в онлайн-системе судьи - избегает напрямую использовать `scanf()` и считывает все строки через `fgets()` . Вы можете использовать `sscanf()` для анализа прочитанных строк.

```
#include <stdio.h>
#include <string.h>
```

```
int main(void) {
    int num = 0;
    char line_buffer[128] = "", str[128], *lf;

    fgets(line_buffer, sizeof(line_buffer), stdin);
    sscanf(line_buffer, "%d", &num);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Другой способ - прочитать, пока вы не нажмете символ новой строки после использования `scanf()` и до использования `fgets()` .

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;
    int c;

    scanf("%d", &num);
    while ((c = getchar()) != '\n' && c != EOF);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Добавление точки с запятой в #define

Легко запутаться в препроцессоре C и рассматривать его как часть самого C, но это ошибка, потому что препроцессор - это просто механизм замены текста. Например, если вы пишете

```
/* WRONG */
#define MAX 100;
int arr[MAX];
```

код расширяется до

```
int arr[100;];
```

который является синтаксической ошибкой. Средством устранения является точка с запятой из строки `#define` . Почти всегда ошибка заканчивается `#define` точкой с запятой.

Многострочные комментарии не могут быть вложенными

В C многострочные комментарии / * и * / не вложены.

Если вы комментируете блок кода или функцию, используя этот стиль комментария:

```
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

Вы не сможете легко прокомментировать это:

```
//Trying to comment out the block...
/*

/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

//Causes an error on the line below...
*/
```

Одним из решений является использование комментариев стиля C99:

```
// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

Теперь весь блок можно легко прокомментировать:

```
/*

// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

*/
```

Другое решение заключается в том, чтобы избежать отключения кода с использованием синтаксиса комментариев, вместо этого вместо него следует использовать директивы препроцессора `#ifdef` или `#ifndef` . Эти директивы *делают* гнездо, оставив вас свободно комментировать свой код в стиле вы предпочитаете.

```
#define DISABLE_MAX /* Remove or comment this line to enable max() code block */

#ifdef DISABLE_MAX
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
#endif
```

Некоторые руководства заходят так далеко, что рекомендуют, чтобы разделы кода *никогда* не комментировались и что, если код должен быть временно отключен, можно прибегнуть к использованию `#if 0` .

См. #If 0 для блокировки разделов кода .

Пересечение границ массива

Массивы основаны на нуле, то есть индекс всегда начинается с 0 и заканчивается длиной массива индекса минус 1. Таким образом, следующий код не будет выводить первый

элемент массива и выводит мусор для окончательного значения, которое он печатает.

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements

    for(x = 1; x <= 5; x++) //Looping from 1 till 5.
       printf("%d\t", myArray[x]);

    printf("\n");
    return 0;
}
```

Выход: `2 3 4 5 GarbageValue`

Ниже демонстрируется правильный способ достижения желаемого результата:

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements

    for(x = 0; x < 5; x++) //Looping from 0 till 4.
       printf("%d\t", myArray[x]);

    printf("\n");
    return 0;
}
```

Выход: `1 2 3 4 5`

Важно знать длину массива, прежде чем работать с ним, так как иначе вы можете повредить буфер или вызвать ошибку сегментации, обратившись к ячейкам памяти, которые находятся за пределами границ.

Рекурсивная функция - отсутствует базовое условие

Вычисление факториала числа является классическим примером рекурсивной функции.

Отсутствует базовое состояние:

```
#include <stdio.h>

int factorial(int n)
{
       return n * factorial(n - 1);
}

int main()
```

```
{
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Типичный выход: `Segmentation fault: 11`

Проблема с этой функцией заключается в том, что она будет чередоваться бесконечно, вызывая сбои сегментации - для этого требуется базовое условие остановки рекурсии.

Базовое условие:

```
#include <stdio.h>

int factorial(int n)
{
    if (n == 1) // Base Condition, very crucial in designing the recursive functions.
    {
       return 1;
    }
    else
    {
       return n * factorial(n - 1);
    }
}

int main()
{
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Образец вывода

```
Factorial 3 = 6
```

Эта функция прекращается, как только она достигает условия `n` равна 1 (при условии, что начальное значение `n` достаточно мало - верхняя граница равна `12` когда `int` является 32-разрядной величиной).

Правила, которым необходимо следовать:

1. Инициализировать алгоритм. Рекурсивным программам часто требуется начальное значение. Это достигается либо с использованием параметра, переданного функции, либо путем предоставления нерекурсивной функции шлюза, но которая устанавливает начальные значения для рекурсивного вычисления.
2. Проверьте, соответствует ли текущее значение (значения) совпадающим с базовым регистром. Если да, обработайте и верните значение.
3. Переопределите ответ в терминах меньшей или более простой подзадачи или под-проблем.
4. Запустите алгоритм в подзапросе.

5. Объедините результаты в формулировке ответа.
6. Верните результаты.

Источник: рекурсивная функция

Проверка логического выражения на 'true'

Первоначальный стандарт C не имел встроенного булева типа, поэтому `bool` , `true` и `false` не имели неотъемлемого значения и часто определялись программистами. Обычно `true` будет определяться как 1, а `false` будет определяться как 0.

C99

C99 добавляет встроенный тип `_Bool` и заголовок `<stdbool.h>` который определяет `bool` (расширяется до `_Bool`), `false` и `true` . Это также позволяет вам переопределить `bool` , `true` и `false` , но отмечает, что это устаревшая функция.

Что еще более важно, логические выражения обрабатывают все, что оценивается как ноль как ложное, а любая ненулевая оценка - как истина. Например:

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    if ((bitField & 0x80) == true)  /* Comparison only succeeds if true is 0x80 and bitField
has that bit set */
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

В приведенном выше примере функция пытается проверить, установлен ли верхний бит и вернуть значение `true` если оно есть. Однако, явно проверяя `true` , оператор `if` будет успешным только в том случае, если `(bitfield & 0x80)` оценивается как любое `true` , которое обычно равно `1` и очень редко `0x80` . Либо явным образом проверяю случай, который вы ожидаете:

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    if ((bitField & 0x80) == 0x80) /* Explicitly test for the case we expect */
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Или оцените любое ненулевое значение как истинное.

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    /* If upper bit is set, result is 0x80 which the if will evaluate as true */
    if (bitField & 0x80)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Литералы с плавающей точкой имеют тип double по умолчанию

Следует проявлять осторожность при инициализации переменных типа `float` до литеральных значений или их сравнении с литеральными значениями, поскольку обычные литералы с плавающей запятой, такие как `0.1` имеют тип `double` . Это может привести к неожиданностям:

```
#include <stdio.h>
int main() {
    float  n;
    n = 0.1;
    if (n > 0.1) printf("Wierd\n");
    return 0;
}
// Prints "Wierd" when n is float
```

Здесь `n` получает инициализацию и округление до одной точности, что приводит к значению 0.10000000149011612. Затем `n` преобразуется обратно в двойную точность для сравнения с `0.1` литеральным (что равно 0,10000000000000001), что приводит к несоответствию.

Помимо ошибок округления, смешивание переменных `float` с `double` литералами приведет к низкой производительности на платформах, которые не имеют аппаратной поддержки для двойной точности.

Прочитайте Обычные подводные камни онлайн: https://riptutorial.com/ru/c/topic/2006/обычные-подводные-камни

глава 34: Ограничения

замечания

Ограничения - это термин, используемый во всех существующих спецификациях C (недавно ISO-IEC 9899-2011). Они являются одной из трех частей языка, описанных в разделе 6 стандарта (наряду с синтаксисом и семантикой).

ISO-IEC 9899-2011 определяет ограничение как:

> ограничение, либо синтаксическое, либо семантическое, с помощью которого следует интерпретировать изложение языковых элементов

(Также обратите внимание, что в терминах стандарта C «ограничение времени выполнения» не является своего рода ограничением и имеет множество разных правил).

Другими словами, ограничение описывает правило языка, которое делает иначе синтаксически действующую программу незаконной. В этом отношении ограничения несколько напоминают неопределенное поведение, любая программа, которая не следует за ними, не определяется в терминах языка C.

С другой стороны, ограничения имеют очень важное отличие от Undefined Behaviors. А именно, для обеспечения диагностического сообщения во время фазы перевода (части компиляции) требуется выполнение, если нарушение нарушено, это сообщение может быть предупреждением или может остановить компиляцию.

Examples

Дублирующие имена переменных в той же области

Пример ограничения, выраженного в стандарте C, имеет две переменные с одним и тем же именем, объявленные в области [1] , например:

```
void foo(int bar)
{
    int var;
    double var;
}
```

Этот код нарушает ограничение и должен давать диагностическое сообщение во время компиляции. Это очень полезно по сравнению с неопределенным поведением, поскольку разработчик будет проинформирован о проблеме до запуска программы, потенциально делая что-либо.

Таким образом, ограничения, как правило, являются ошибками, которые легко обнаруживаются во время компиляции, такие как проблемы, которые приводят к неопределенному поведению, но которые были бы трудными или невозможными для обнаружения во время компиляции, поэтому не являются ограничениями.

---

1) точная формулировка:

C99

Если идентификатор не имеет привязки, должно быть не более одного объявления идентификатора (в указателе или спецификаторе типа) с той же областью и в том же пространстве имен, за исключением тегов, указанных в 6.7.2.3.

Унарные арифметические операторы

Унарные `+` и `-` операторы применимы только к арифметическим типам, поэтому, если, например, один пытается использовать их в структуре, программа создаст диагностику, например:

```
struct foo
{
    bool bar;
};

void baz(void)
{
   struct foo testStruct;
   -testStruct; /* This breaks the constraint so must produce a diagnostic */
}
```

Прочитайте Ограничения онлайн: https://riptutorial.com/ru/c/topic/7397/ограничения

глава 35: операторы

Вступление

Оператор на языке программирования является символом, который сообщает компилятору или интерпретатору выполнять определенную математическую, реляционную или логическую операцию и дает конечный результат.

C имеет много мощных операторов. Многие операторы C являются двоичными операторами, что означает, что они имеют два операнда. Например, в `a / b` , `/` является двоичным оператором, который принимает два операнда (`a` , `b`). Есть некоторые унарные операторы, которые берут один операнд (например: `~` , `++`) и только один тернарный оператор `? :` .

Синтаксис

- Оператор expr1
- оператор expr2
- Оператор expr1 expr2
- expr1? expr2: expr3

замечания

Операторы имеют *арность, предшествование* и *ассоциативность.*

- *Arity* указывает количество операндов. В C существуют три различные категории операторов:
    - Унарный (1 операнд)
    - Двоичные (2 операнда)
    - Тройной (3 операнда)
- *Приоритет* указывает, какие операторы «связывают» сначала с их операндами. То есть оператор имеет приоритет для работы с операндами. Например, язык C подчиняется соглашению о том, что умножение и деление имеют приоритет над сложениями и вычитанием:

    ```
    a * b + c
    ```

    Дает тот же результат, что и

    ```
    (a * b) + c
    ```

Если это не то, что нужно, приоритет можно принудительно использовать с помощью круглых скобок, поскольку они имеют *наивысший* приоритет для всех операторов.

```
a * (b + c)
```

Это новое выражение приведет к результату, который отличается от предыдущих двух выражений.

Язык C имеет много уровней приоритета; Ниже приведена таблица всех операторов в порядке убывания приоритета.

Таблица приоритетов

операторы	Ассоциативность
`() [] -> .`	слева направо
`! ~ ++ -- + - *` (разыменование) `(type) sizeof`	справа налево
`*` (умножение) `/ %`	слева направо
`+ -`	слева направо
`<< >>`	слева направо
`< <= > >=`	слева направо
`== !=`	слева направо
`&`	слева направо
`^`	слева направо
`|`	слева направо
`&&`	слева направо
`||`	слева направо
`?:`	справа налево
`= += -= *= /= %= &= ^= |= <<= >>=`	справа налево
`,`	слева направо

- *Ассоциативность* показывает, как по умолчанию назначаются операторы с приоритетом приоритета, и существует два вида: *слева направо* и *справа налево* .

Примером привязки *Left-to-Right* является оператор вычитания (`-`). Выражение

```
a - b - c - d
```

имеет три вычитания с одинаковым приоритетом, но дает тот же результат, что и

```
((a - b) - c) - d
```

потому что самое левое `-` связывается сначала с двумя его операндами.

Примером *правоотношенной* ассоциативности являются операторы разыменования `*` и post-increment `++` . Оба имеют одинаковый приоритет, поэтому, если они используются в выражении, таком как

```
* ptr ++
```

, это эквивалентно

```
* (ptr ++)
```

потому что самый правый, унарный оператор (`++`) связывает сначала свой единственный операнд.

Examples

Реляционные операторы

Операторы отношения проверяют, является ли конкретное отношение между двумя операндами истинным. Результат оценивается в 1 (что означает *true*) или 0 (что означает *false*). Этот результат часто используется для влияния на поток управления (через `if` , `while` , `for`), но также может храниться в переменных.

Равно "=="

Проверяет, равны ли поставленные операнды.

```
1 == 0;         /* evaluates to 0. */
1 == 1;         /* evaluates to 1. */

int x = 5;
int y = 5;
int *xptr = &x, *yptr = &y;
xptr == yptr;   /* evaluates to 0, the operands hold different location addresses. */
*xptr == *yptr; /* evaluates to 1, the operands point at locations that hold the same value.
*/
```

Внимание: этот оператор не следует путать с оператором присваивания (`=`)!

Не равно "! ="

Проверяет, не равны ли поставленные операнды.

```
1 != 0;         /* evaluates to 1. */
1 != 1;         /* evaluates to 0. */

int x = 5;
int y = 5;
int *xptr = &x, *yptr = &y;
xptr != yptr;   /* evaluates to 1, the operands hold different location addresses. */
*xptr != *yptr; /* evaluates to 0, the operands point at locations that hold the same value.
*/
```

Этот оператор фактически возвращает противоположный результат к `==` оператора equals (`==`).

Не "!"

Проверьте, равен ли объект `0` .

`!` также можно использовать непосредственно с переменной следующим образом:

```
!someVal
```

Это имеет тот же эффект, что и:

```
someVal == 0
```

Больше, чем ">"

Проверяет, имеет ли левый операнд большее значение, чем правый операнд

```
5 > 4      /* evaluates to 1. */
4 > 5      /* evaluates to 0. */
4 > 4      /* evaluates to 0. */
```

Меньше, чем "<"

Проверяет, имеет ли левый операнд меньшее значение, чем правый операнд

```
5 < 4      /* evaluates to 0. */
4 < 5      /* evaluates to 1. */
4 < 4      /* evaluates to 0. */
```

Больше или равно "> ="

Проверяет, имеет ли левый операнд большее или равное значение правильному операнду.

```
5 >= 4      /* evaluates to 1. */
4 >= 5      /* evaluates to 0. */
4 >= 4      /* evaluates to 1. */
```

Меньше или равно "<="

Проверяет, имеет ли левый операнд меньшее или равное значение правильному операнду.

```
5 <= 4      /* evaluates to 0. */
4 <= 5      /* evaluates to 1. */
4 <= 4      /* evaluates to 1. */
```

Операторы присваивания

Присваивает значение правого операнда ячейке хранения, названной левым операндом, и возвращает значение.

```
int x = 5;      /* Variable x holds the value 5. Returns 5. */
char y = 'c';   /* Variable y holds the value 99. Returns 99
                 * (as the character 'c' is represented in the ASCII table with 99).
                 */
float z = 1.5;  /* variable z holds the value 1.5. Returns 1.5. */
char const* s = "foo"; /* Variable s holds the address of the first character of the string
'foo'. */
```

Несколько арифметических операций имеют *составной* оператор *присваивания* .

```
a += b  /* equal to: a = a + b */
a -= b  /* equal to: a = a - b */
a *= b  /* equal to: a = a * b */
a /= b  /* equal to: a = a / b */
a %= b  /* equal to: a = a % b */
a &= b  /* equal to: a = a & b */
a |= b  /* equal to: a = a | b */
a ^= b  /* equal to: a = a ^ b */
a <<= b /* equal to: a = a << b */
a >>= b /* equal to: a = a >> b */
```

Важной особенностью этих составных назначений является то, что выражение в левой части (`a`) оценивается только один раз. Например, если `p` является указателем

```
*p += 27;
```

разыгрывания `p` только один раз, тогда как следующее делает это дважды.

```
*p = *p + 27;
```

Следует также отметить, что результатом присвоения, такого как `a = b` является то, что известно как *rvalue* . Таким образом, присваивание фактически имеет значение, которое затем может быть присвоено другой переменной. Это позволяет цепочки присвоений задавать несколько переменных в одном выражении.

Это значение *rvalue* может использоваться в управляющих выражениях операторов `if` (или циклов или операторов `switch`), которые защищают некоторый код от результата другого выражения или вызова функции. Например:

```
char *buffer;
if ((buffer = malloc(1024)) != NULL)
{
    /* do something with buffer */
    free(buffer);
}
else
{
    /* report allocation failure */
}
```

Из-за этого следует проявлять осторожность, чтобы избежать общей опечатки, которая может привести к таинственным ошибкам.

```
int a = 2;
/* ... */
if (a = 1)
    /* Delete all files on my hard drive */
```

Это будет иметь катастрофические результаты, так как `a = 1` всегда будет оцениваться до `1` и, таким образом, контролирующее выражение оператора `if` всегда будет истинным (подробнее об этом распространенном явлении см. Здесь). Автор почти наверняка имел в виду использовать оператор равенства (`==`), как показано ниже:

```
int a = 2;
/* ... */
if (a == 1)
    /* Delete all files on my hard drive */
```

Ассоциативность операторов

```
int a, b = 1, c = 2;
a = b = c;
```

Это присваивает `c` в `b` , который возвращает `b` , который присваивается `a` . Это происходит потому, что все операторы присваивания имеют правую ассоциативность, что означает, что самая правая операция в выражении сначала оценивается и продолжается справа налево.

Основная арифметика

Возвращает значение, которое является результатом применения левого операнда в правый операнд, используя соответствующую математическую операцию. Применяются нормальные математические правила коммутации (т. Е. Сложение и умножение являются коммутативными, вычитание, деление и модуль не являются).

Оператор добавления

Оператор сложения (`+`) используется для добавления двух операндов вместе. Пример:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;

    int c = a + b; /* c now holds the value 12 */

    printf("%d + %d = %d",a,b,c); /* will output "5 + 7 = 12" */

    return 0;
}
```

Оператор вычитания

Оператор вычитания (`-`) используется для вычитания второго операнда из первого. Пример:

```
#include <stdio.h>

int main(void)
{
    int a = 10;
    int b = 7;

    int c = a - b; /* c now holds the value 3 */

    printf("%d - %d = %d",a,b,c); /* will output "10 - 7 = 3" */

    return 0;
}
```

Оператор умножения

Оператор умножения (`*`) используется для умножения обоих операндов. Пример:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;

    int c = a * b; /* c now holds the value 35 */

    printf("%d * %d = %d",a,b,c); /* will output "5 * 7 = 35" */

    return 0;
}
```

Не следует путать с `*` оператора разыменования.

Оператор отдела

Оператор деления (`/`) делит первый операнд на второй. Если оба операнда деления являются целыми числами, он вернет целочисленное значение и отбросит остаток (используйте для вычисления и получения остатка оператор modulo `%`).

Если один из операндов является значением с плавающей запятой, результатом является аппроксимация фракции.

Пример:

```
#include <stdio.h>

int main (void)
{
    int a = 19 / 2 ; /* a holds value 9   */
    int b = 18 / 2 ; /* b holds value 9   */
    int c = 255 / 2; /* c holds value 127 */
    int d = 44 / 4 ; /* d holds value 11  */
    double e = 19 / 2.0 ; /* e holds value 9.5   */
    double f = 18.0 / 2 ; /* f holds value 9.0   */
    double g = 255 / 2.0; /* g holds value 127.5 */
    double h = 45.0 / 4 ; /* h holds value 11.25 */

    printf("19 / 2 = %d\n", a);    /* Will output "19 / 2 = 9"    */
    printf("18 / 2 = %d\n", b);    /* Will output "18 / 2 = 9"    */
    printf("255 / 2 = %d\n", c);   /* Will output "255 / 2 = 127" */
    printf("44 / 4 = %d\n", d);    /* Will output "44 / 4 = 11"   */
    printf("19 / 2.0 = %g\n", e);  /* Will output "19 / 2.0 = 9.5"    */
    printf("18.0 / 2 = %g\n", f);  /* Will output "18.0 / 2 = 9"      */
    printf("255 / 2.0 = %g\n", g); /* Will output "255 / 2.0 = 127.5" */
    printf("45.0 / 4 = %g\n", h);  /* Will output "45.0 / 4 = 11.25"  */

    return 0;
}
```

Оператор Modulo

Оператор modulo (`%`) принимает только целые операнды и используется для вычисления остатка после того, как первый операнд делится на второй. Пример:

```
#include <stdio.h>

int main (void) {
    int a = 25 % 2;    /* a holds value 1  */
    int b = 24 % 2;    /* b holds value 0  */
    int c = 155 % 5;   /* c holds value 0  */
    int d = 49 % 25;   /* d holds value 24 */

    printf("25 % 2 = %d\n", a);     /* Will output "25 % 2 = 1"    */
    printf("24 % 2 = %d\n", b);     /* Will output "24 % 2 = 0"    */
    printf("155 % 5 = %d\n", c);    /* Will output "155 % 5 = 0"   */
    printf("49 % 25 = %d\n", d);    /* Will output "49 % 25 = 24"  */

    return 0;
}
```

Операторы приращения / уменьшения

Операторы increment (`a++`) и `a--` (`a--`) отличаются друг от друга тем, что они изменяют значение переменной, к которой вы применяете их, без оператора присваивания. Вы можете использовать операторы increment и decment либо до, либо после переменной. Размещение оператора изменяет время нарастания / уменьшения значения до или после его назначения переменной. Пример:

```
#include <stdio.h>

int main(void)
{
    int a = 1;
    int b = 4;
    int c = 1;
    int d = 4;

    a++;
    printf("a = %d\n",a);    /* Will output "a = 2" */
    b--;
    printf("b = %d\n",b);    /* Will output "b = 3" */

    if (++c > 1) { /* c is incremented by 1 before being compared in the condition */
        printf("This will print\n");    /* This is printed */
    } else {
        printf("This will never print\n");    /* This is not printed */
    }

    if (d-- < 4) {  /* d is decremented after being compared */
        printf("This will never print\n");    /* This is not printed */
    } else {
        printf("This will print\n");    /* This is printed */
    }
}
```

Как показывает пример для `c` и `d` , оба оператора имеют две формы: префиксную нотацию

и постфиксную нотацию. Оба имеют одинаковый эффект при добавлении (`++`) или уменьшении (`--`) переменной, но отличаются значением, которое они возвращают: операции префикса выполняют операцию сначала, а затем возвращают значение, тогда как операции postfix сначала определяют значение, которое возвращаться, а затем выполнять операцию.

Из-за этого потенциально контр-интуитивного поведения использование операторов increment / decment внутри выражений противоречиво.

Логические операторы

Логические И

Выполняет логическое логическое AND-IN двух операндов, возвращающих 1, если оба операнда отличны от нуля. Логический оператор И имеет тип `int` .

```
0 && 0  /* Returns 0. */
0 && 1  /* Returns 0. */
2 && 0  /* Returns 0. */
2 && 3  /* Returns 1. */
```

Логический ИЛИ

Выполняет логическое логическое OR-IN двух операндов, возвращающих 1, если какой-либо из операндов отличен от нуля. Логический оператор OR имеет тип `int` .

```
0 || 0  /* Returns 0. */
0 || 1  /* Returns 1.  */
2 || 0  /* Returns 1.  */
2 || 3  /* Returns 1.  */
```

Логическое НЕ

Выполняет логическое отрицание. Логический оператор NOT имеет тип `int` . Оператор NOT проверяет, равен ли хотя бы один бит 1, если он возвращает 0. Иначе он возвращает 1;

```
!1 /* Returns 0. */
!5 /* Returns 0. */
!0 /* Returns 1. */
```

Оценка короткого замыкания

Существуют некоторые важные свойства, общие как для `&&` и для `||` :

- левый операнд (LHS) полностью оценивается до того, как будет оценен правый

операнд (RHS)
- существует точка последовательности между оценкой левого операнда и правого операнда,
- и, что более важно, правый операнд вообще не оценивается, если результат левого операнда определяет общий результат.

Это означает, что:

- если LHS оценивает значение «true» (отличное от нуля), RHS `||` не будет оценен (потому что результат «true OR any» равен «true»)
- если LHS оценивает значение «false» (ноль), RHS `&&` не будет оцениваться (потому что результат «false AND anything» равен «false»).

Это важно, поскольку позволяет писать код, например:

```
const char *name_for_value(int value)
{
    static const char *names[] = { "zero", "one", "two", "three", };
    enum { NUM_NAMES = sizeof(names) / sizeof(names[0]) };
    return (value >= 0 && value < NUM_NAMES) ? names[value] : "infinity";
}
```

Если отрицательное значение передается функции, `value >= 0` определяет `value < NUM_NAMES` false, а `value < NUM_NAMES` термин не оценивается.

Приращение / уменьшение

Операторы increment and decment существуют в *префиксной* и *постфиксной* форме.

```
int a = 1;
int b = 1;
int tmp = 0;

tmp = ++a;         /* increments a by one, and returns new value; a == 2, tmp == 2  */
tmp = a++;         /* increments a by one, but returns old value; a == 3, tmp == 2 */
tmp = --b;         /* decrements b by one, and returns new value; b == 0, tmp == 0 */
tmp = b--;         /* decrements b by one, but returns old value; b == -1, tmp == 0 */
```

Обратите внимание, что арифметические операции не вводят точки последовательности , поэтому некоторые выражения с `++` или `--` операторами могут вводить неопределенное поведение .

Условный оператор / Тернарный оператор

Вычисляет свой первый операнд и, если результирующее значение не равно нулю, оценивает его второй операнд. В противном случае он оценивает свой третий операнд, как показано в следующем примере:

```
a = b ? c : d;
```

эквивалентно:

```
if (b)
    a = c;
else
    a = d;
```

Этот псевдокод представляет это: `condition ? value_if_true : value_if_false` . Каждое значение может быть результатом оцененного выражения.

```
int x = 5;
int y = 42;
printf("%i, %i\n", 1 ? x : y, 0 ? x : y); /* Outputs "5, 42" */
```

Условный оператор может быть вложенным. Например, следующий код определяет большее из трех чисел:

```
big= a > b ? (a > c ? a : c)
           : (b > c ? b : c);
```

Следующий пример записывает даже целые числа в один файл и нечетные целые числа в другой файл:

```
#include<stdio.h>

int main()
{
    FILE *even, *odds;
    int n = 10;
    size_t k = 0;

    even = fopen("even.txt", "w");
    odds = fopen("odds.txt", "w");

    for(k = 1; k < n + 1; k++)
    {
        k%2==0 ? fprintf(even, "\t%5d\n", k)
               : fprintf(odds, "\t%5d\n", k);
    }
    fclose(even);
    fclose(odds);

    return 0;
}
```

Условный оператор ассоциируется справа налево. Рассмотрим следующее:

```
exp1 ? exp2 : exp3 ? exp4 : exp5
```

Поскольку ассоциация справа налево, указанное выше выражение оценивается как

```
exp1 ? exp2 : ( exp3 ? exp4 : exp5 )
```

Comma Operator

Вычисляет его левый операнд, отбрасывает полученное значение и затем оценивает его операнд прав, и результат дает значение его самого правого операнда.

```
int x = 42, y = 42;
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
```

Оператор запятой вводит точку последовательности между ее операндами.

Обратите внимание, что *запятая,* используемая в функциях, вызывает, что отдельные аргументы НЕ являются *оператором запятой* , скорее это называется *разделителем,* который отличается от *оператора запятой* . Следовательно, он не обладает свойствами *оператора запятой* .

Вышеупомянутый вызов `printf()` содержит как *оператор запятой, так* и *разделитель* .

```
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
/*           ^        ^ this is a comma operator */
/*           this is a separator */
```

Оператор запятой часто используется в секции инициализации, а также в секции обновления цикла `for` . Например:

```
for(k = 1; k < 10; printf("\%d\\n", k), k += 2);   /*outputs the odd numbers below 9/*

/* outputs sum to first 9 natural numbers */
for(sumk = 1, k = 1; k < 10; k++, sumk += k)
    printf("\%5d\%5d\\n", k, sumk);
```

Оператор ротации

Выполняет *явное* преобразование в заданный тип из значения, полученного в результате вычисления данного выражения.

```
int x = 3;
int y = 4;
printf("%f\n", (double)x / y); /* Outputs "0.750000". */
```

При этом величина `x` преобразуется в `double` деление способствует значение `y` к `double` , тоже, и результат отделения, `double` передается `printf` для печати.

Оператор размера

С типом в качестве операнда

Вычисляет размер в байтах типа `size_t` объектов данного типа. Требуется круглые скобки вокруг типа.

```
printf("%zu\n", sizeof(int)); /* Valid, outputs the size of an int object, which is platform-
dependent. */
printf("%zu\n", sizeof int); /* Invalid, types as arguments need to be surrounded by
parentheses! */
```

С выражением в качестве операнда

Оценивает размер в байтах типа `size_t` объектов типа данного выражения. Само выражение не оценивается. Скобки не требуются; однако, поскольку данное выражение должно быть унарным, считается, что наилучшей практикой всегда пользоваться ими.

```
char ch = 'a';
printf("%zu\n", sizeof(ch)); /* Valid, will output the size of a char object, which is always
1 for all platforms. */
printf("%zu\n", sizeof ch);  /* Valid, will output the size of a char object, which is always
1 for all platforms. */
```

Арифметика указателей

Добавление указателя

С учетом указателя и скалярного типа `N` вычисляется в указатель на `N` й элемент указанного типа, который непосредственно преследует объект с указателем в памяти.

```
int arr[] = {1, 2, 3, 4, 5};
printf("*(arr + 3) = %i\n", *(arr + 3)); /* Outputs "4", arr's fourth element. */
```

Не имеет значения, используется ли указатель в качестве значения операнда или скалярного значения. Это означает, что действуют такие вещи, как `3 + arr` . Если `arr[k]` является `k+1` элементом массива, то `arr+k` является указателем на `arr[k]` . Другими словами, `arr` или `arr+0` является указателем на `arr[0]` , `arr+1` является указателем на `arr[2]` и т. Д. В общем случае `*(arr+k)` совпадает с `arr[k]` .

В отличие от обычной арифметики добавление `1` к указателю на `int` добавит `4` байта к текущему значению адреса. Поскольку имена массива являются постоянными указателями, `+` является единственным оператором, который мы можем использовать для доступа к элементам массива посредством нотации указателя с использованием имени массива. Однако, определяя указатель на массив, мы можем получить большую гибкость для обработки данных в массиве. Например, мы можем напечатать элементы массива следующим образом:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", *(arr + k));
    }
    return 0;
}
```

Определив указатель на массив, вышеуказанная программа эквивалентна следующему:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr; /* or int *ptr = &arr[0]; */
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", ptr[k]);
        /* or   printf("\n\t%d", *(ptr + k)); */
        /* or   printf("\n\t%d", *ptr++); */
    }
    return 0;
}
```

Посмотрите, что члены массива `arr` получают доступ с помощью операторов `+` и `++` . Другие операторы, которые могут использоваться с указателем `ptr` , `-` и `--` .

Вычитание указателя

Учитывая два указателя на один и тот же тип, оценивается объект типа `ptrdiff_t` который содержит скалярное значение, которое должно быть добавлено ко второму указателю, чтобы получить значение первого указателя.

```
int arr[] = {1, 2, 3, 4, 5};
int *p = &arr[2];
int *q = &arr[3];
ptrdiff_t diff = q - p;

printf("q - p = %ti\n", diff); /* Outputs "1". */
printf("*(p + (q - p)) = %d\n", *(p + diff)); /* Outputs "4". */
```

Операторы доступа

Операторы доступа члена (точка `.` И стрелка `->`) используются для доступа к члену `struct` .

Участник объекта

Вычисляет значение lvalue, обозначающее объект, являющийся членом объекта доступа.

```
struct MyStruct
{
    int x;
    int y;
};

struct MyStruct myObject;
myObject.x = 42;
myObject.y = 123;

printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Outputs ".x = 42, .y = 123". */
```

Участник объекта с указателем

Синтаксический сахар для разыменования, за которым следует доступ членов. Эффективно выражение вида `x->y` является сокращением для `(*x).y` - но оператор стрелки намного ясен, особенно если указатели структуры вложены.

```
struct MyStruct
{
    int x;
    int y;
};

struct MyStruct myObject;
struct MyStruct *p = &myObject;

p->x = 42;
p->y = 123;

printf(".x = %i, .y = %i\n", p->x, p->y); /* Outputs ".x = 42, .y = 123". */
printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Also outputs ".x = 42, .y = 123". */
```

Адресов

Унарный оператор `&` является адресом оператора. Он оценивает данное выражение, где результирующий объект должен быть lvalue. Затем он вычисляет объект, тип которого является указателем на тип результирующего объекта и содержит адрес результирующего объекта.

```
int x = 3;
int *p = &x;
printf("%p = %p\n", (void *)&x, (void *)p); /* Outputs "A = A", for some implementation-
defined A. */
```

разыменовывают

Оператор унарного `*` разыскивает указатель. Он оценивает значение lvalue, возникающее в результате разыменования указателя, который возникает в результате оценки данного выражения.

```
int x = 42;
int *p = &x;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 42, *p = 42". */

*p = 123;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 123, *p = 123". */
```

индексирование

Индексирование - это синтаксический сахар для добавления указателя, за которым следует разыменование. Эффективно выражение вида `a[i]` эквивалентно `*(a + i)` но явное обозначение индекса является предпочтительным.

```
int arr[] = { 1, 2, 3, 4, 5 };
printf("arr[2] = %i\n", arr[2]); /* Outputs "arr[2] = 3". */
```

Взаимозаменяемость индексации

Добавление указателя в целое число является коммутативной операцией (т. Е. Порядок операндов не изменяет результат), так что `pointer + integer == integer + pointer` .

Следствием этого является то, что `arr[3]` и `3[arr]` эквивалентны.

```
printf("3[arr] = %i\n", 3[arr]); /* Outputs "3[arr] = 4". */
```

Использование выражения `3[arr]` вместо `arr[3]` обычно не рекомендуется, так как оно влияет на читаемость кода. Он имеет тенденцию быть популярным в запутанных конкурсах программирования.

Оператор вызова функции

Первый операнд должен быть указателем на функцию (признак функции также приемлем, поскольку он будет преобразован в указатель на функцию), идентифицируя функцию для вызова, а все другие операнды, если они есть, все вместе известны как аргументы вызова функции , Вычисляет возвращаемое значение, вызванное вызовом соответствующей функции с соответствующими аргументами.

```
int myFunction(int x, int y)
{
```

```
    return x * 2 + y;
}

int (*fn)(int, int) = &myFunction;
int x = 42;
int y = 123;

printf("(*fn)(%i, %i) = %i\n", x, y, (*fn)(x, y)); /* Outputs "fn(42, 123) = 207". */
printf("fn(%i, %i) = %i\n", x, y, fn(x, y)); /* Another form: you don't need to dereference
explicitly */
```

Побитовые операторы

Побитовые операторы могут использоваться для выполнения операций с битовым уровнем по переменным.
Ниже приведен список всех шести побитовых операторов, поддерживаемых в C:

Условное обозначение	оператор
&	побитовое И
|	побитовое включение ИЛИ
^	побитовое исключающее ИЛИ (XOR)
~	побитовое (дополнение)
<<	логический сдвиг влево
>>	логическая сдвиг вправо

Следующая программа иллюстрирует использование всех побитовых операторов:

```
#include <stdio.h>

int main(void)
{
   unsigned int a = 29;    /* 29 = 0001 1101 */
   unsigned int b = 48;    /* 48 = 0011 0000 */
   int c = 0;

   c = a & b;              /* 32 = 0001 0000 */
   printf("%d & %d = %d\n", a, b, c );

   c = a | b;              /* 61 = 0011 1101 */
   printf("%d | %d = %d\n", a, b, c );

   c = a ^ b;              /* 45 = 0010 1101 */
   printf("%d ^ %d = %d\n", a, b, c );

   c = ~a;                 /* -30 = 1110 0010 */
   printf("~%d = %d\n", a, c );
```

```
   c = a << 2;             /* 116 = 0111 0100 */
   printf("%d << 2 = %d\n", a, c );

   c = a >> 2;             /* 7 = 0000 0111 */
   printf("%d >> 2 = %d\n", a, c );

   return 0;
}
```

Следует избегать побитовых операций с подписанными типами, поскольку знаковый бит такого битового представления имеет особое значение. Конкретные ограничения применяются к операторам сдвига:

- Перемещение влево 1 бит в бит бит является ошибочным и приводит к неопределенному поведению.
- Правильное смещение отрицательного значения (со знаком бит 1) является реализацией, и поэтому не переносится.
- Если значение правого операнда оператора сдвига отрицательное или больше или равно ширине продвинутого левого операнда, поведение не определено.

Маскировка:

Маскировка относится к процессу извлечения желаемых битов из (или преобразования желаемых бит) переменной с использованием логических побитовых операций. Операнд (константа или переменная), который используется для выполнения маскировки, называется *маской* .

Маскировка используется по-разному:

- Чтобы решить битовый шаблон целочисленной переменной.
- Чтобы скопировать часть данного битового шаблона в новую переменную, в то время как остальная часть новой переменной заполняется 0 (с помощью побитового И)
- Чтобы скопировать часть данного битового шаблона в новую переменную, в то время как остальная часть новой переменной заполняется 1 с (с использованием побитового ИЛИ).
- Чтобы скопировать часть заданного битового шаблона в новую переменную, в то время как оставшаяся часть исходного битового шаблона инвертируется в новой переменной (с использованием побитового исключения OR).

Следующая функция использует маску для отображения битовой диаграммы переменной:

```
#include <limits.h>
void bit_pattern(int u)
{
    int i, x, word;
    unsigned mask = 1;
    word = CHAR_BIT * sizeof(int);
```

```
    mask = mask << (word - 1);    /* shift 1 to the leftmost position */
    for(i = 1; i <= word; i++)
    {
        x = (u & mask) ? 1 : 0;  /* identify the bit */
        printf("%d", x);         /* print bit value */
        mask >>= 1;              /* shift mask to the right by 1 bit */
    }
}
```

_Alignof

C11

Запросит требование выравнивания для указанного типа. Требование выравнивания представляет собой положительную интегральную мощность 2, представляющую количество байтов, между которыми могут быть выделены два объекта типа. В C требование выравнивания измеряется в `size_t` .

Имя типа не может быть неполным, а не типом функции. Если в качестве типа используется массив, используется тип элемента массива.

К этому оператору часто обращаются через `alignof` макросов `alignof` из `<stdalign.h>` .

```
int main(void)
{
    printf("Alignment of char = %zu\n", alignof(char));
    printf("Alignment of max_align_t = %zu\n", alignof(max_align_t));
    printf("alignof(float[10]) = %zu\n", alignof(float[10]));
    printf("alignof(struct{char c; int n;}) = %zu\n",
            alignof(struct {char c; int n;}));
}
```

Возможный выход:

```
Alignment of char = 1
Alignment of max_align_t = 16
alignof(float[10]) = 4
alignof(struct{char c; int n;}) = 4
```

http://en.cppreference.com/w/c/language/_Alignof

Короткое замыкание логических операторов

Короткое замыкание - это функциональность, которая пропускает оценку состояния условия (if / while / ...), когда это возможно. В случае логической операции над двумя операндами первый операнд оценивается (с истинным или ложным), и если есть вердикт (т.е. первый операнд является ложным при использовании &&, первый операнд имеет значение true при использовании ||), второй операнд Не Оценено.

Пример:

```
#include <stdio.h>

int main(void) {
  int a = 20;
  int b = -5;

  /* here 'b == -5' is not evaluated,
     since a 'a != 20' is false. */
  if (a != 20 && b == -5) {
    printf("I won't be printed!\n");
  }

  return 0;
}
```

Проверьте это самостоятельно:

```
#include <stdio.h>

int print(int i) {
  printf("print function %d\n", i);
  return i;
}

int main(void) {
  int a = 20;

  /* here 'print(a)' is not called,
     since a 'a != 20' is false. */
  if (a != 20 && print(a)) {
    printf("I won't be printed!\n");
  }

  /* here 'print(a)' is called,
     since a 'a == 20' is true. */
  if (a == 20 && print(a)) {
    printf("I will be printed!\n");
  }

  return 0;
}
```

Выход:

```
$ ./a.out
print function 20
I will be printed!
```

Короткое замыкание важно, когда вы хотите избежать оценки терминов, которые (вычислительно) являются дорогостоящими. Более того, это может сильно повлиять на поток вашей программы, как в этом случае: почему эта программа печатает «разветвленный!»? 4 раза?

Прочитайте операторы онлайн: https://riptutorial.com/ru/c/topic/256/операторы

глава 36: Параметры функции

замечания

В C обычно используется возвращаемое значение для обозначения ошибок, которые происходят; и возвращать данные через использование переданных в указателях. Это можно сделать по нескольким причинам; в том числе не нужно выделять память в куче или использовать статическое распределение в точке, где вызывается функция.

Examples

Использование параметров указателя для возврата нескольких значений

Общий шаблон на C, чтобы легко имитировать возврат нескольких значений из функции, заключается в использовании указателей.

```
#include <stdio.h>

void Get( int* c , double* d )
{
    *c = 72;
    *d = 175.0;
}

int main(void)
{
    int a = 0;
    double b = 0.0;

    Get( &a , &b );

    printf("a: %d, b: %f\n", a , b );

    return 0;
}
```

Передача массивов в функции

```
int getListOfFriends(size_t size, int friend_indexes[]) {
  size_t i = 0;
  for (; i < size; i++) {
    friend_indexes[i] = i;
  }
}
```

C99 C11

```
/* Type "void" and VLAs ("int friend_indexes[static size]") require C99 at least.
   In C11 VLAs are optional. */
```

```
void getListOfFriends(size_t size, int friend_indexes[static size]) {
  size_t i = 0;
  for (; i < size; i++) {
    friend_indexes[i] = 1;
  }
}
```

Здесь `static` внутри `[]` параметра функции, запросите, чтобы массив аргументов должен иметь как минимум столько элементов, сколько указано (т. `size` Элементы `size`). Чтобы иметь возможность использовать эту функцию, мы должны убедиться, что параметр `size` находится перед параметром массива в списке.

Используйте `getListOfFriends()` следующим образом:

```
#define LIST_SIZE (50)

int main(void) {
  size_t size_of_list = LIST_SIZE;
  int friends_indexes[size_of_list];

  getListOfFriends(size_of_list, friend_indexes); /* friend_indexes decays to a pointer to the
                                                     address of its 1st element:
                                                       &friend_indexes[0] */

  /* Here friend_indexes carries: {0, 1, 2, ..., 49}; */

  return 0;
}
```

Смотрите также

Передача многомерных массивов в функцию

Параметры передаются по значению

В C все параметры функции передаются по значению, поэтому изменение того, что передается в вызываемых функциях, не влияет на локальные переменные функций вызывающего абонента.

```
#include <stdio.h>

void modify(int v) {
    printf("modify 1: %d\n", v); /* 0 is printed */
    v = 42;
    printf("modify 2: %d\n", v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(v);
    printf("main 2: %d\n", v); /* 0 is printed, not 42 */
```

```
    return 0;
}
```

Вы можете использовать указатели, чтобы функции вызываемого абонента изменяли локальные переменные функций вызывающего абонента. Обратите внимание, что это не *пропуск по ссылке,* но *значения* указателя, указывающие на локальные переменные, передаются.

```
#include <stdio.h>

void modify(int* v) {
    printf("modify 1: %d\n", *v); /* 0 is printed */
    *v = 42;
    printf("modify 2: %d\n", *v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(&v);
    printf("main 2: %d\n", v); /* 42 is printed */
    return 0;
}
```

Однако возврат адреса локальной переменной к вызываемому приводит к неопределенному поведению. См. Разделение указателя на переменную за пределами ее срока службы .

Порядок выполнения функции

Порядок выполнения параметров не определен в программировании на C. Здесь он может выполняться слева направо или справа налево. Порядок зависит от реализации.

```
#include <stdio.h>

void function(int a, int b)
{
    printf("%d %d\n", a, b);
}

int main(void)
{
    int a = 1;
    function(a++, ++a);
    return 0;
}
```

Пример функции, возвращающей структуру, содержащую значения с кодами ошибок

Большинство примеров функции, возвращающей значение, включают предоставление

указателя в качестве одного из аргументов, позволяющих функции изменять указанное значение, похожее на следующее. Фактическое возвращаемое значение функции обычно представляет собой некоторый тип, такой как `int` чтобы указывать статус результата, независимо от того, работает он или нет.

```
int func (int *pIvalue)
{
    int iRetStatus = 0;             /* Default status is no change */

    if (*pIvalue > 10) {
        *pIvalue = *pIvalue * 45;   /* Modify the value pointed to */
        iRetStatus = 1;             /* indicate value was changed */
    }

    return iRetStatus;              /* Return an error code */
}
```

Однако вы также можете использовать `struct` как возвращаемое значение, которое позволяет вам возвращать как статус ошибки вместе с другими значениями. Например.

```
typedef struct {
    int    iStat;      /* Return status */
    int    iValue;     /* Return value */
}  RetValue;

RetValue func (int iValue)
{
    RetValue iRetStatus = {0, iValue};

    if (iValue > 10) {
        iRetStatus.iValue = iValue * 45;
        iRetStatus.iStat = 1;
    }

    return iRetStatus;
}
```

Затем эту функцию можно использовать, как в следующем примере.

```
int usingFunc (int iValue)
{
    RetValue iRet = func (iValue);

    if (iRet.iStat == 1) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

Или он может быть использован следующим образом.

```
int usingFunc (int iValue)
{
    RetValue iRet;
```

```
    if ( (iRet = func (iValue)).iStat == 1 ) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

Прочитайте Параметры функции онлайн: https://riptutorial.com/ru/c/topic/1006/параметры-функции

глава 37: Передача 2D-массивов в функции

Examples

Передайте 2D-массив функции

Передача 2d-массива в функции кажется простой и очевидной, и мы счастливо пишем:

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int **, int, int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int n = ROWS;
  int m = COLS;

  fun1(array_2D, n, m);

  return EXIT_SUCCESS;
}

void fun1(int **a, int n, int m)
{
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Но компилятор, здесь GCC в версии 4.9.4, не ценит это хорошо.

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c11 passarr.c -o passarr
passarr.c: In function ‘main’:
passarr.c:16:8: warning: passing argument 1 of ‘fun1’ from incompatible pointer type
   fun1(array_2D, n, m);
        ^
passarr.c:8:6: note: expected ‘int **’ but argument is of type ‘int (*)[2]’
 void fun1(int **, int, int);
```

Причины этого двоякие: основная проблема заключается в том, что массивы не являются указателями, а второе неудобство - так называемым *разложением указателя* . Передача массива в функцию приведет к распаду массива указателю на первый элемент массива - в

случае 2d-массива он распадается на указатель на первую строку, потому что в массивах C сортируются по первой строке.

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int, int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int n = ROWS;
  int m = COLS;

  fun1(array_2D, n, m);

  return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int n, int m)
{
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Необходимо передать количество строк, они не могут быть вычислены.

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int rows = ROWS;

  /* works here because array_2d is still in scope and still an array */
  printf("MAIN: %zu\n",sizeof(array_2D)/sizeof(array_2D[0]));

  fun1(array_2D, rows);

  return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int rows)
{
  int i, j;
  int n, m;
```

```
  n = rows;
  /* Works, because that information is passed (as "COLS").
     It is also redundant because that value is known at compile time (in "COLS"). */
  m = (int) (sizeof(a[0])/sizeof(a[0][0]));

  /* Does not work here because the "decay" in "pointer decay" is meant
     literally--information is lost. */
  printf("FUN1: %zu\n",sizeof(a)/sizeof(a[0]));

  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

C99

Количество столбцов предопределено и, следовательно, фиксировано во время компиляции, но предшественник текущего C-стандарта (который был ISO / IEC 9899: 1999, текущий ISO / IEC 9899: 2011) реализовал VLA (TODO: link it) и хотя текущий стандарт сделал его дополнительным, почти все современные C-компиляторы поддерживают его (TODO: проверьте, поддерживает ли MS Visual Studio сейчас).

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];

  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = (i + 1) * (j + 1);
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(array_2D, rows, cols);

  exit(EXIT_SUCCESS);
}
```

```
void fun1(int (*a)[], int rows, int cols)
{
  int i, j;
  int n, m;

  n = rows;
  /* Does not work anymore, no sizes are specified anymore
  m = (int) (sizeof(a[0])/sizeof(a[0][0])); */
  m = cols;


  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Это не работает, компилятор жалуется:

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c99 passarr.c -o passarr
passarr.c: In function ‘fun1’:
passarr.c:168:7: error: invalid use of array with unspecified bounds
       printf("array[%d][%d]=%d\n", i, j, a[i][j]);
```

Это становится немного яснее, если мы намеренно сделаем ошибку в вызове функции, изменив объявление на `void fun1(int **a, int rows, int cols)` . Это заставляет компилятор жаловаться другим, но одинаково туманным способом

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c99 passarr.c -o passarr
passarr.c: In function ‘main’:
passarr.c:208:8: warning: passing argument 1 of ‘fun1’ from incompatible pointer type
   fun1(array_2D, rows, cols);
        ^
passarr.c:185:6: note: expected ‘int **’ but argument is of type ‘int (*)[(sizetype)(cols)]’
 void fun1(int **, int rows, int cols);
```

Мы можем реагировать несколькими способами, одним из которых является игнорировать все это и делать нечеткие манипуляции с указателем:

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }
```

```
  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(array_2D, rows, cols);

  exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, *( (*a) + (i * cols + j)));
    }
  }
}
```

Или мы делаем это правильно и передаем необходимую информацию в `fun1` . Для этого нам нужно переставить аргументы в `fun1` : размер столбца должен быть до объявления массива. Чтобы сделать его более читаемым, переменная, содержащая количество строк, тоже изменила свое место, и теперь она является первой.

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int rows, int cols, int (*)[]);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
```

```
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(rows, cols, array_2D);

  exit(EXIT_SUCCESS);
}

void fun1(int rows, int cols, int (*a)[cols])
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Это выглядит неловко для некоторых людей, которые придерживаются мнения о том, что порядок переменных не имеет значения. Это не большая проблема, просто объявите указатель и дайте ему указать на массив.

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int rows, int cols, int **);

int main(int argc, char **argv)
{
  int rows, cols, i, j;


  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
```

```
  }
  // a "rows" number of pointers to "int". Again a VLA
  int *a[rows];
  // initialize them to point to the individual rows
  for (i = 0; i < rows; i++) {
      a[i] = array_2D[i];
  }

  fun1(rows, cols, a);

  exit(EXIT_SUCCESS);
}

void fun1(int rows, int cols, int **a)
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Использование плоских массивов в виде 2D-массивов

Часто самым простым решением является простое преобразование 2D и более высоких массивов в виде плоской памяти.

```
/* create 2D array with dimensions determined at runtime */
double *matrix = malloc(width * height * sizeof(double));

/* initialise it (for the sake of illustration we want 1.0 on the diagonal) */
int x, y;
for (y = 0; y < height; y++)
{
    for (x = 0; x < width; x++)
    {
        if (x == y)
            matrix[y * width + x] = 1.0;
        else
            matrix[y * width + x] = 0.0;
    }
}

/* pass it to a subroutine */
 manipulate_matrix(matrix, width, height);


/* do something with the matrix, e.g. scale by 2 */
void manipulate_matrix(double *matrix, int width, int height)
{
    int x, y;
```

```
    for (y = 0; y < height; y++)
    {
        for (x = 0; x < width; x++)
        {
            matrix[y * width + x] *= 2.0;
        }
    }
}
```

Прочитайте Передача 2D-массивов в функции онлайн: https://riptutorial.com/ru/c/topic/6862/передача-2d-массивов-в-функции

глава 38: Перейти к началу страницы

Синтаксис

- return val; / * Возвращает из текущей функции. val может быть значением любого типа, которое преобразуется в возвращаемый тип функции. * /
- вернуть; / * Возвращает из текущей функции void. * /
- перерыв; / * Безусловно перескакивает за пределы («разрывается») итерационного заявления (цикл) или из самого внутреннего оператора switch. * /
- Продолжить; / * Безусловно переходит в начало итерационного заявления (цикл). * /
- goto LBL; / * Переход к метке LBL. * /
- LBL: *statement* / * любое утверждение в той же функции. * /

замечания

Это прыжки, которые интегрируются в C с помощью ключевых слов.

C также имеет другую конструкцию перехода, *длинный прыжок* , который указан с типом данных, `jmp_buf` и вызовами библиотеки C, `setjmp` и `longjmp` .

Смотрите также

Итерационные выражения / Циклы: для, пока, делать-пока

Examples

Использование goto для перехода из вложенных циклов

Выпрыгивание из вложенных циклов обычно требует использования логической переменной с проверкой этой переменной в циклах. Предположим, что мы итерируем по `i` и `j` , это может выглядеть так

```
size_t i,j;
for (i = 0; i < myValue && !breakout_condition; ++i) {
    for (j = 0; j < mySecondValue && !breakout_condition; ++j) {
        ... /* Do something, maybe modifying breakout_condition */
            /* When breakout_condition == true the loops end */
     }
}
```

Но язык C предлагает предложение `goto` , которое может быть полезно в этом случае. Используя его с меткой, объявленной после циклов, мы можем легко вырваться из циклов.

```
size_t i,j;
for (i = 0; i < myValue; ++i) {
    for (j = 0; j < mySecondValue; ++j) {
        ...
        if(breakout_condition)
          goto final;
    }
}
final:
```

Однако часто, когда эта потребность приходит, `return` может быть лучше использовано вместо этого. Эта конструкция также считается «неструктурированной» в теории структурного программирования.

Другая ситуация, когда `goto` может быть полезной, - это прыжок в обработчик ошибок:

```
ptr = malloc(N *  x);
if(!ptr)
  goto out_of_memory;

/* normal processing */
free(ptr);
return SUCCESS;

out_of_memory:
 free(ptr); /* harmless, and necessary if we have further errors */
 return FAILURE;
```

Использование `goto` сохраняет поток ошибок отдельно от обычного потока управления программой. Однако он также считается «неструктурированным» в техническом смысле.

Использование возврата

Возврат значения

Один часто используемый случай: возвращение из `main()`

```
#include <stdlib.h> /* for EXIT_xxx macros */

int main(int argc, char ** argv)
{
  if (2 < argc)
  {
    return EXIT_FAILURE; /* The code expects one argument:
                            leave immediately skipping the rest of the function's code */
  }

  /* Do stuff. */

  return EXIT_SUCCESS;
}
```

Дополнительные примечания:

1. Для функции, имеющей тип возврата как `void` (не включая `void *` или связанные типы), оператор `return` не должен иметь никакого связанного выражения; т.е. единственным допустимым оператором return будет `return;` ,

2. Для функции, имеющей `void` тип `return` оператор `return` не должен появляться без выражения.

3. Для `main()` (и только для `main()`) *явный* оператор `return` не требуется (на C99 или новее). Если выполнение достигает завершения `}` , возвращается неявное значение `0` . Некоторые люди думают, что отказ от этого `return` - плохая практика; другие активно предлагают отказаться от этого.

Возвращение ничего

Возвращение из функции `void`

```
void log(const char * message_to_log)
{
  if (NULL == message_to_log)
  {
    return; /* Nothing to log, go home NOW, skip the logging. */
  }

  fprintf(stderr, "%s:%d %s\n", __FILE__, _LINE__, message_to_log);

  return; /* Optional, as this function does not return a value. */
}
```

Использование break и continue

Сразу же `continue` чтение недопустимого ввода или `break` на запросе пользователя или конце файла:

```
#include <stdlib.h> /* for EXIT_xxx macros */
#include <stdio.h>  /* for printf() and getchar() */
#include <ctype.h> /* for isdigit() */

void flush_input_stream(FILE * fp);


int main(void)
{
  int sum = 0;
  printf("Enter digits to be summed up or 0 to exit:\n");

  do
  {
    int c = getchar();
    if (EOF == c)
    {
      printf("Read 'end-of-file', exiting!\n");
```

```
      break;
    }

    if ('\n' != c)
    {
      flush_input_stream(stdin);
    }

    if (!isdigit(c))
    {
      printf("%c is not a digit! Start over!\n", c);

      continue;
    }

    if ('0' == c)
    {
      printf("Exit requested.\n");

      break;
    }

    sum += c - '0';

    printf("The current sum is %d.\n", sum);
  } while (1);

  return EXIT_SUCCESS;
}

void flush_input_stream(FILE * fp)
{
  size_t i = 0;
  int c;
  while ((c = fgetc(fp)) != '\n' && c != EOF) /* Pull all until and including the next new-
line. */
  {
    ++i;
  }

  if (0 != i)
  {
    fprintf(stderr, "Flushed %zu characters from input.\n", i);
  }
}
```

Прочитайте Перейти к началу страницы онлайн: https://riptutorial.com/ru/c/topic/5568/перейти-к-началу-страницы

глава 39: Переменные аргументы

Вступление

Аргументы переменной используются функциями семейства printf (`printf` , `fprintf` и т. Д.) И другими, чтобы позволить функции вызываться с различным количеством аргументов каждый раз, поэтому имя *varargs* .

Чтобы реализовать функции с использованием функции аргументов переменных, используйте `#include <stdarg.h>` .

Чтобы вызывать функции, которые принимают переменное количество аргументов, убедитесь, что в области видимости имеется полный прототип с `void err_exit(const char *format, ...);` эллипсисом: `void err_exit(const char *format, ...);` например.

Синтаксис

- void va_start (va_list ap, *последний*); / * Запустить обработку вариационных аргументов; *last* - последний параметр функции перед многоточием («...») * /
- *тип* va_arg (va_list ap, *type*); / * Получить следующий переменный аргумент в списке; обязательно передайте правильный *продвинутый* тип * /
- void va_end (va_list ap); / * Обработка конечных аргументов * /
- void va_copy (va_list dst, va_list src); / * C99 или более поздняя версия: список аргументов копирования, то есть текущая позиция в обработке аргументов, в другой список (например, для передачи нескольких аргументов несколько раз) * /

параметры

параметр	подробности
`va_list` ap	указатель аргументов, текущая позиция в списке вариационных аргументов
прошлой	имя последнего аргумента невариантной функции, поэтому компилятор находит правильное место для начала обработки вариативных аргументов; не может быть объявлена как переменная `register` , функция или тип массива
тип	**расширенный** тип VARIADIC аргумента для чтения (например , `int` для `short int` аргумента)
`va_list`	указатель текущего аргумента для копирования

параметр	подробности
src	
`va_list` dst	новый список аргументов, который должен быть заполнен

замечания

Функции `va_start` , `va_arg` , `va_end` и `va_copy` самом деле являются макросами.

Обязательно *всегда* вызывайте `va_start` сначала, и только один раз, и вызывать `va_end` последним, и только один раз, и на каждой точке выхода функции. Не делать этого *может* работать на *вашей* системе, но, безусловно, **не** переносится и, следовательно, вызывает ошибки.

Позаботьтесь о том, чтобы правильно заявить свою функцию, то есть с прототипом, и учитывать ограничения на *последний* невариантный аргумент (не `register` , а не функцию или тип массива). Невозможно объявить функцию, которая принимает только переменные аргументы, так как необходим хотя бы один невариантный аргумент, чтобы можно было начать обработку аргументов.

При вызове `va_arg` вы должны запросить тип **продвигаемого** аргумента, а именно:

- `short` повышается до `int` (и `unsigned short` также увеличивается до `int` если `sizeof(unsigned short) == sizeof(int)` , и в этом случае ему присваивается `unsigned int`).
- `float` повышается `double` .
- `signed char` продвигается до `int` ; `unsigned char` также продвигается до `int` если `sizeof(unsigned char) == sizeof(int)` , что редко бывает.
- `char` обычно повышается до `int` .
- Аналогичным образом продвигаются такие типы C99, как `uint8_t` или `int16_t` .

Историческая (например, K & R) обработка вариационных аргументов объявляется в `<varargs.h>` но не должна использоваться, поскольку она устарела. Стандартная обработка вариационных аргументов (описанная здесь и объявленная в `<stdarg.h>`) была введена в C89; макрос `va_copy` был введен в C99, но предоставлен многими компиляторами до этого.

Examples

Использование явного аргумента count для определения длины va_list

При любой вариационной функции функция должна знать, как интерпретировать список аргументов переменных. С функциями `printf()` или `scanf()` строка формата сообщает функции, что ожидать.

Самый простой способ - передать явный счет других аргументов (которые обычно имеют один и тот же тип). Это продемонстрировано в вариационной функции в приведенном ниже коде, который вычисляет сумму целых чисел, где может быть любое число целых чисел, но этот счет указывается в качестве аргумента перед списком переменных аргументов.

```
#include <stdio.h>
#include <stdarg.h>

/* first arg is the number of following int args to sum. */
int sum(int n, ...) {
    int sum = 0;
    va_list it; /* hold information about the variadic argument list. */

    va_start(it, n); /* start variadic argument processing */
    while (n--)
      sum += va_arg(it, int); /* get and sum the next variadic argument */
    va_end(it); /* end variadic argument processing */

    return sum;
}

int main(void)
{
    printf("%d\n", sum(5, 1, 2, 3, 4, 5)); /* prints 15 */
    printf("%d\n", sum(10, 5, 9, 2, 5, 111, 6666, 42, 1, 43, -6218)); /* prints 666 */
    return 0;
}
```

Использование значений терминатора для определения конца va_list

При любой вариационной функции функция должна знать, как интерпретировать список аргументов переменных. «Традиционный» подход (например, `printf`) заключается в том, чтобы указать количество аргументов впереди. Однако это не всегда хорошая идея:

```
/* First argument specifies the number of parameters; the remainder are also int */
extern int sum(int n, ...);

/* But it's far from obvious from the code. */
sum(5, 2, 1, 4, 3, 6)

/* What happens if i.e. one argument is removed later on? */
sum(5, 2, 1, 3, 6)  /* Disaster */
```

Иногда более удобно добавлять явный терминатор, примером которого является функция POSIX `execlp()` . Вот еще одна функция для вычисления суммы ряда `double` чисел:

```
#include <stdarg.h>
#include <stdio.h>
#include <math.h>

/* Sums args up until the terminator NAN */
double sum (double x, ...) {
  double sum = 0;
```

```
  va_list va;

  va_start(va, x);
  for (; !isnan(x); x = va_arg(va, double)) {
    sum += x;
  }
  va_end(va);

  return sum;
}

int main (void) {
  printf("%g\n", sum(5., 2., 1., 4., 3., 6., NAN));
  printf("%g\n", sum(1, 0.5, 0.25, 0.125, 0.0625, 0.03125, NAN));
}
```

Хорошие значения терминатора:

- целое (должно быть все положительное или неотрицательное) - `0` или `-1`
- типы с плавающей точкой - `NAN`
- типы указателей - `NULL`
- типы перечислителей - какое-то особое значение

Реализация функций с помощью интерфейса `printf ()` -like

Одним из распространенных вариантов списков аргументов переменной длины является реализация функций, которые представляют собой тонкую оболочку вокруг семейства функций `printf()` . Одним из таких примеров является набор функций отчетности об ошибках.

errmsg.h

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h>    // C11

void verrmsg(int errnum, const char *fmt, va_list ap);
noreturn void errmsg(int exitcode, int errnum, const char *fmt, ...);
void warnmsg(int errnum, const char *fmt, ...);

#endif
```

Это пример с голубыми костями; такие пакеты могут быть очень сложными. Обычно программисты будут использовать либо `errmsg()` либо `warnmsg()` , которые сами используют `verrmsg()` внутри. Однако, если кто-то `verrmsg()` необходимость сделать больше, то `verrmsg()` функция `verrmsg()` будет полезна. Вы могли бы избежать воздействия на него до тех пор , пока есть потребность в нем (YAGNI - ты не собираешься это нужно), но потребность возникнет в конце концов (вам понадобится это - YAGNI).

errmsg.c

Этот код должен только пересылать переменные аргументы функции `vfprintf()` для вывода на стандартную ошибку. Он также сообщает системное сообщение об ошибке, соответствующее номеру системной ошибки (`errno`), переданному функциям.

```
#include "errmsg.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void
verrmsg(int errnum, const char *fmt, va_list ap)
{
    if (fmt)
        vfprintf(stderr, fmt, ap);
    if (errnum != 0)
        fprintf(stderr, ": %s", strerror(errnum));
    putc('\n', stderr);
}

void
errmsg(int exitcode, int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
    exit(exitcode);
}

void
warnmsg(int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
}
```

Использование `errmsg.h`

Теперь вы можете использовать эти функции следующим образом:

```
#include "errmsg.h"
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    char buffer[BUFSIZ];
    int fd;
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
    const char *filename = argv[1];

    if ((fd = open(filename, O_RDONLY)) == -1)
        errmsg(EXIT_FAILURE, errno, "cannot open %s", filename);
    if (read(fd, buffer, sizeof(buffer)) != sizeof(buffer))
        errmsg(EXIT_FAILURE, errno, "cannot read %zu bytes from %s", sizeof(buffer),
filename);
    if (close(fd) == -1)
        warnmsg(errno, "cannot close %s", filename);
    /* continue the program */
    return 0;
}
```

Если системные вызовы `open()` или `read()` не выполняются, ошибка записывается в стандартную ошибку, и программа выходит с кодом выхода 1. Если системный вызов `close()` не работает, ошибка просто печатается как предупреждающее сообщение и программа продолжается.

Проверка правильности использования форматов `printf()`

Если вы используете GCC (компилятор GNU C, который является частью сборника компиляторов GNU) или используя Clang, вы можете проверить, что аргументы, переданные в функции сообщений об ошибках, соответствуют тому, что ожидает `printf()` . Поскольку не все компиляторы поддерживают расширение, его необходимо скомпилировать условно, что немного затруднительно. Тем не менее, защита, которую она дает, стоит усилий.

Во-первых, нам нужно знать, как определить, что компилятор - это GCC или Clang, имитирующий GCC. Ответ заключается в том, что GCC определяет `__GNUC__` чтобы указать это.

См. Общие атрибуты функций для получения информации об атрибутах - в частности `format` атрибута `format` .

Переписано `errmsg.h`

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h>    // C11

#if !defined(PRINTFLIKE)
#if defined(__GNUC__)
#define PRINTFLIKE(n,m) __attribute__((format(printf,n,m)))
#else
#define PRINTFLIKE(n,m) /* If only */
#endif /* __GNUC__ */
#endif /* PRINTFLIKE */

void verrmsg(int errnum, const char *fmt, va_list ap);
void noreturn errmsg(int exitcode, int errnum, const char *fmt, ...)
        PRINTFLIKE(3, 4);
```

```
void warnmsg(int errnum, const char *fmt, ...)
        PRINTFLIKE(2, 3);

#endif
```

Теперь, если вы допустили ошибку, например:

```
errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
```

(где `%d` должно быть `%s`), тогда компилятор будет жаловаться:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes \
>     -Wold-style-definition -c erruse.c
erruse.c: In function ‘main’:
erruse.c:20:64: error: format ‘%d’ expects argument of type ‘int’, but argument 4 has type
‘const char *’ [-Werror=format=]
         errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
                                                               ~^
                                                               %s
cc1: all warnings being treated as errors
$
```

Использование строки формата

Использование строки формата предоставляет информацию о ожидаемом числе и типе последующих вариационных аргументов таким образом, чтобы избежать необходимости явного аргумента счетчика или значения терминатора.

В приведенном ниже примере показана функция aa, которая обертывает стандартную функцию `printf()` , только позволяя использовать вариационные аргументы типа `char` , `int` и `double` (в формате десятичной с плавающей запятой). Здесь, как и в случае с `printf()` , первым аргументом функции wrapping является строка формата. По мере разбора строки форматирования функция может определить, есть ли ожидаемый другой вариационный аргумент и каков его тип.

```
#include <stdio.h>
#include <stdarg.h>

int simple_printf(const char *format, ...)
{
    va_list ap; /* hold information about the variadic argument list. */
    int printed = 0; /* count of printed characters */

    va_start(ap, format); /* start variadic argument processing */

    while (*format != '\0') /* read format string until string terminator */
    {
        int f = 0;

        if (*format == '%')
        {
            ++format;
```

```
            switch(*format)
            {
                case 'c' :
                    f = printf("%d", va_arg(ap, int)); /* print next variadic argument, note
type promotion from char to int */
                    break;
                case 'd' :
                    f = printf("%d", va_arg(ap, int)); /* print next variadic argument */
                    break;

                case 'f' :
                    f = printf("%f", va_arg(ap, double)); /* print next variadic argument */
                    break;
                default :
                    f = -1; /* invalid format specifier */
                    break;
            }
        }
        else
        {
            f = printf("%c", *format); /* print any other characters */
        }

        if (f < 0) /* check for errors */
        {
            printed = f;
            break;
        }
        else
        {
            printed += f;
        }
        ++format; /* move on to next character in string */
    }

    va_end(ap); /* end variadic argument processing */

    return printed;
}

int main (int argc, char *argv[])
{
    int x = 40;
    int y = 0;

    y = simple_printf("There are %d characters in this sentence", x);
    simple_printf("\n%d were printed\n", y);
}
```

Прочитайте Переменные аргументы онлайн: https://riptutorial.com/ru/c/topic/455/переменные-аргументы

глава 40: Перечисления

замечания

Перечисления состоят из ключевого слова `enum` и необязательного *идентификатора,* за которым следует *список перечислителей,* заключенный в фигурные скобки.

Идентификатор имеет тип `int` .

Список *перечислителей* имеет как минимум один элемент *перечисления* .

Перечислитель может необязательно «назначать» постоянное выражение типа `int` .

Перечислитель является постоянным и совместим либо с `char` , либо с целым `char` , либо с целым числом без знака. Используемый когда-либо используется для реализации . В любом случае используемый тип должен иметь возможность представлять все значения, определенные для рассматриваемой нумерации.

Если не константа не «назначен» на *переписчик* , и это 1 - й вход в *перечислителях-листе* он принимает значение `0` , иначе получите принимает значение предыдущей записи в *перечислителях-листе* плюс 1.

Использование нескольких «назначений» может привести к тому, что разные *счетчики* того же перечисления несут одни и те же значения.

Examples

Простое перечисление

Перечисление представляет собой пользовательский тип данных, состоящий из интегральных констант, и каждой интегральной константе присваивается имя. Перечисление `enum` слова используется для определения перечислимого типа данных.

Если вы используете `enum` вместо `int` или `string/ char*` , вы увеличиваете проверку времени компиляции и избегаете ошибок при передаче недопустимых констант, и вы документируете, какие значения являются законными для использования.

Пример 1

```
enum color{ RED, GREEN, BLUE };

void printColor(enum color chosenColor)
{
    const char *color_name = "Invalid color";
```

```
    switch (chosenColor)
    {
       case RED:
         color_name = "RED";
         break;

       case GREEN:
        color_name = "GREEN";
        break;

       case BLUE:
        color_name = "BLUE";
        break;
    }
    printf("%s\n", color_name);
}
```

С основной функцией, определенной следующим образом (например):

```
int main(){
    enum color chosenColor;
    printf("Enter a number between 0 and 2");
    scanf("%d", (int*)&chosenColor);
    printColor(chosenColor);
    return 0;
}
```

C99

Пример 2.

(В этом примере используются назначенные инициализаторы, которые стандартизованы с C99.)

```
enum week{ MON, TUE, WED, THU, FRI, SAT, SUN };

static const char* const dow[] = {
  [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
  [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };

void printDayOfWeek(enum week day)
{
   printf("%s\n", dow[day]);
}
```

В этом же примере используется проверка диапазона:

```
enum week{ DOW_INVALID = -1,
  MON, TUE, WED, THU, FRI, SAT, SUN,
  DOW_MAX };

static const char* const dow[] = {
  [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
  [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };

void printDayOfWeek(enum week day)
```

```
{
   assert(day > DOW_INVALID && day < DOW_MAX);
   printf("%s\n", dow[day]);
}
```

Typedef перечисление

Существует несколько возможностей и условных обозначений для перечисления. Первое - использовать *имя тега* сразу после ключевого слова `enum` .

```
enum color
{
    RED,
    GREEN,
    BLUE
};
```

Это перечисление должно всегда использоваться с ключевым словом *и* тегом, как это:

```
enum color chosenColor = RED;
```

Если мы используем `typedef` непосредственно при объявлении `enum` , мы можем опустить имя тега, а затем использовать тип без ключевого слова `enum` :

```
typedef enum
{
    RED,
    GREEN,
    BLUE
} color;

color chosenColor = RED;
```

Но в этом последнем случае мы не можем использовать его как `enum color` , потому что мы не использовали имя тега в определении. Одна общая конвенция использовать оба, так что же имя может использоваться с или без `enum` ключевых слов. Это имеет особое преимущество совместимости с C ++

```
enum color                /* as in the first example */
{
    RED,
    GREEN,
    BLUE
};
typedef enum color color; /* also a typedef of same identifier */

color chosenColor  = RED;
enum color defaultColor = BLUE;
```

Функция:

```
void printColor()
{
    if (chosenColor == RED)
    {
        printf("RED\n");
    }
    else if (chosenColor == GREEN)
    {
        printf("GREEN\n");
    }
    else if (chosenColor == BLUE)
    {
        printf("BLUE\n");
    }
}
```

Подробнее о `typedef` см. Typedef

Перечисление с дублирующимся значением

Значение перечислений никоим образом не должно быть уникальным:

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */


enum Dupes
{
   Base, /* Takes 0 */
   One, /* Takes Base + 1 */
   Two, /* Takes One + 1 */
   Negative = -1,
   AnotherZero /* Takes Negative + 1 == 0, sigh */
};

int main(void)
{
  printf("Base = %d\n", Base);
  printf("One = %d\n", One);
  printf("Two = %d\n", Two);
  printf("Negative = %d\n", Negative);
  printf("AnotherZero = %d\n", AnotherZero);

  return EXIT_SUCCESS;
}
```

Образец печатает:

```
Base = 0
One = 1
Two = 2
Negative = -1
AnotherZero = 0
```

константа перечисления без typename

Типы перечислений также могут быть объявлены без указания им имени:

```
enum { buffersize = 256, };
static unsigned char buffer [buffersize] = { 0 };
```

Это позволяет нам определять константы времени компиляции типа `int` которые могут, как в этом примере, использоваться как длина массива.

Прочитайте Перечисления онлайн: https://riptutorial.com/ru/c/topic/5460/перечисления

глава 41: Побочные эффекты

Examples

Операторы Pre / Post Increment / Decrement

В C есть два унарных оператора - «++» и «-», которые являются очень распространенным источником путаницы. Оператор `++` называется *оператором приращения,* а оператор `--` называется *декрементом* . Оба они могут использоваться как в *префиксной* форме, так и в форме *постфикса* . Синтаксис префикса формы `++` оператора `++operand` и синтаксис формы постфикса является `operand++` . При использовании в форме префикса операнд увеличивается сначала на `1` а результирующее значение операнда используется при оценке выражения. Рассмотрим следующий пример:

```
int n, x = 5;
n = ++x; /* x is incremented  by 1(x=6), and result is assigned to n(6) */
         /* this is a short form for two statements: */
         /* x = x + 1; */
         /* n = x ; */
```

При использовании в постфиксной форме в выражении используется текущее значение операнда, а затем значение операнда увеличивается на `1` . Рассмотрим следующий пример:

```
int n, x = 5;
n = x++; /* value of x(5) is assigned first to n(5), and then x is incremented by 1; x(6) */
         /* this is a short form for two statements: */
         /* n = x; */
         /* x = x + 1; */
```

Рабочий оператор декремента `--` можно понимать аналогичным образом .

Следующий код демонстрирует, что каждый делает

```
    int main()
    {
        int a, b, x = 42;
        a = ++x; /* a and x are 43 */
        b = x++; /* b is 43, x is 44 */
        a = x--; /* a is is 44, x is 43 */
        b = --x; /* b and x are 42 */

        return 0;
    }
```

Из вышесказанного ясно, что пост-операторы возвращают текущее значение переменной, а *затем* изменяют его, но операторы pre модифицируют переменную, а *затем* возвращают измененное значение.

Во всех версиях C порядок оценки операторов pre и post не определен, поэтому следующий код может возвращать неожиданные выходы:

```
int main()
{
    int a, x = 42;
    a = x++ + x; /* wrong */
    a = x + x; /* right */
    ++x;

    int ar[10];
    x = 0;
    ar[x] = x++; /* wrong */
    ar[x++] = x; /* wrong */
    ar[x] = x; /* right */
    ++x;
    return 0;
}
```

Обратите внимание, что также рекомендуется использовать pre over post-операторы, когда они используются отдельно в инструкции. Посмотрите на приведенный выше код для этого.

Также обратите внимание, что при вызове функции все побочные эффекты для аргументов должны выполняться до запуска функции.

```
int foo(int x)
{
    return x;
}

int main()
{
    int a = 42;
    int b = foo(a++);  /* This returns 43, even if it seems like it should return 42 */
    return 0;
}
```

Прочитайте Побочные эффекты онлайн: https://riptutorial.com/ru/c/topic/7094/побочные-эффекты

глава 42: Поведение, определяемое реализацией

замечания

обзор

Стандарт C описывает синтаксис языка, функции, предоставляемые стандартной библиотекой, и поведение совместимых процессоров C (грубо говоря, компиляторов) и соответствующих программ C. Что касается поведения, стандарт в большинстве случаев указывает на специфическое поведение программ и процессоров. С другой стороны, некоторые операции имеют явное или неявное *неопределенное поведение* - таких операций всегда следует избегать, так как вы не можете полагаться на что-либо о них. Между ними существует множество вариантов поведения, *определенных* для *реализации* . Такое поведение может варьироваться между процессорами C, временем выполнения и стандартными библиотеками (в совокупности, *реализациями*), но они являются последовательными и надежными для любой конкретной реализации, а соответствующие реализации документируют их поведение в каждой из этих областей.

Иногда разумно, чтобы программа полагалась на поведение, определяемое реализацией. Например, если программа в любом случае специфична для конкретной операционной среды, то полагаться на определенные по реализации поведения, общие для общих процессоров для этой среды, вряд ли будет проблемой. В качестве альтернативы можно использовать условные компиляционные директивы для выбора определенных поведением поведения, подходящих для используемой реализации. В любом случае, важно знать, какие операции имеют поведение, определяемое реализацией, чтобы либо избежать их, либо принять обоснованное решение о том, использовать ли и как их использовать.

Баланс этих замечаний представляет собой список всех определений поведения и характеристик, определенных в стандарте C2011, со ссылками на стандарт. Многие из них используют терминологию стандарта . Некоторые другие в большей степени полагаются на контекст стандарта, например восемь этапов перевода исходного кода в программу или разницу между размещенными и автономными реализациями. Некоторые, которые могут быть особенно удивительными или заметными, представлены жирным шрифтом. Не все описанные поведения поддерживаются более ранними стандартами C, но, как правило, они имеют поведение, определенное реализацией, во всех версиях стандарта, которые их поддерживают.

Программы и процессоры

генеральный

- **Количество бит в одном байте** (3.6 / 3). Не менее `8` , фактическое значение может быть запрошено с помощью макроса `CHAR_BIT` .
- Какие выходные сообщения считаются «диагностическими сообщениями» (3.10 / 1)

Перевод исходного текста

- Способ отображения многобайтовых символов физического исходного файла в исходный набор символов (5.1.1.2/1).
- Независимо от того, заменяются ли непустые последовательности пробелов без новой строки единичными пробелами во время фазы перевода 3 (5.1.1.2/1)
- Символ (ы) выполнения, к которым преобразуются символьные литералы и символы в строковых константах (во время фазы 5 перевода), когда в противном случае нет соответствующего символа (5.1.1.2/1).

Рабочая среда

- Порядок идентификации диагностических сообщений (5.1.1.3/1).
- Имя и тип функции, вызванной при запуске в автономной реализации (5.1.2.1/1).
- Какие библиотечные средства доступны в автономной реализации за пределами заданного минимального набора (5.1.2.1/1).
- Эффект завершения программы в автономной среде (5.1.2.1/2).
- В размещенной среде любые допустимые сигнатуры для функции `main()` отличные от `int main(int argc, char *arg[])` и `int main(void)` (5.1.2.2.1 / 1).
- Способ, которым размещенная реализация определяет строки, на которые указывает второй аргумент `main()` (5.1.2.2.1 / 2).
- Что представляет собой «интерактивное устройство» для целей разделов 5.1.2.3 (Исполнение программы) и 7.21.3 (Файлы) (5.1.2.3/7).
- Любые ограничения на объекты, на которые ссылаются процедуры прерывания-обработчика в оптимизирующей реализации (5.1.2.3/10).
- В автономной реализации поддерживается ли несколько потоков выполнения (5.1.2.4/1).
- Значения элементов набора символов выполнения (5.2.1 / 1).

- Значения `char` соответствующие определенным алфавитным escape-последовательностям (5.2.2 / 3).
- **Целочисленные и числовые ограничения и характеристики с плавающей запятой** (5.2.4.2/1).
- Точность арифметических операций с плавающей запятой и преобразований стандартной библиотеки из внутренних представлений с плавающей запятой в строковые представления (5.2.4.2.2 / 6).
- Значение макроса `FLT_ROUNDS` , которое кодирует режим округления с плавающей запятой по умолчанию (5.2.4.2.2 / 8).
- `FLT_ROUNDS` округления, характеризуемый поддерживаемыми значениями `FLT_ROUNDS` более 3 или менее -1 (5.2.4.2.2 / 8).
- Значение макроса `FLT_EVAL_METHOD` , которое характеризует поведение оценки с плавающей запятой (5.2.4.2.2 / 9).
- Поведение, характеризуемое любыми поддерживаемыми значениями `FLT_EVAL_METHOD` меньше -1 (5.2.4.2.2 / 9).
- Значения макросов `FLT_HAS_SUBNORM` , `DBL_HAS_SUBNORM` и `LDBL_HAS_SUBNORM` , характеризующие, `LDBL_HAS_SUBNORM` ли стандартные форматы с плавающей запятой субнормальные числа (5.2.4.2.2 / 10)

Типы

- Результат попытки (косвенно) доступа к объекту с длительностью хранения потоков из потока, отличного от того, с которым связан объект (6.2.4 / 4)
- Значение `char` которому присвоен символ вне базового исполнения (6.2.5 / 3).
- Поддерживаемые расширенные подписанные целочисленные типы, если они есть, (6.2.5 / 4) и любые слова расширения, используемые для их идентификации.
- **Является ли `char` тем же представлением и поведением, что и `signed char` или как `unsigned char`** (6.2.5 / 15). Может запрашиваться с `CHAR_MIN` , который равен `0` или `SCHAR_MIN` если `char` без знака или подписан, соответственно.
- **Число, порядок и кодирование байтов в представлениях объектов** , за исключением тех случаев, когда это явно указано стандартом (6.2.6.1/2).
- **Какая из трех признанных форм целочисленного представления применяется в любой заданной ситуации и являются ли определенные битовые шаблоны целых объектов ловушками** (6.2.6.2/2).

- Требования к выравниванию каждого типа (6.2.8 / 1).
- Будут ли и в каких контекстах поддерживаться любые расширенные выравнивания (6.2.8 / 3).
- Набор поддерживаемых расширенных выравниваний (6.2.8 / 4).
- Целочисленные числа преобразования любых расширенных целочисленных типов со знаком относительно друг друга (6.3.1.1/1).
- **Эффект назначения значения вне диапазона значению целого числа** (6.3.1.3/3).
- Когда для объекта с плавающей запятой назначается неопределенное значение, но нерепрезентативное значение, как представляемое значение, хранящееся в объекте, выбирается из двух ближайших представимых значений (6.3.1.4/2 ; 6.3.1.5/1 ; 6.4.4.2 / 3).
- **Результат преобразования целого числа в тип указателя** , за исключением целочисленных константных выражений со значением `0` (6.3.2.3/5).

Форма источника

- Расположение в директивах `#pragma` которых распознаются заголовки заголовков (6.4 / 4).
- Символы, включая многобайтовые символы, кроме подчеркивания, латинские буквы без пробелов , универсальные имена символов и десятичные цифры, которые могут отображаться в идентификаторах (6.4.2.1/1).
- **Число значимых символов в идентификаторе** (6.4.2.1/5).
- За некоторыми исключениями, порядок, в котором исходные символы в целочисленной символьной константе сопоставляются с установленными символами (6.4.4.4/2 , 6.4.4.4/10).
- Текущая локализация, используемая для вычисления значения широкой символьной константы и большинства других аспектов преобразования для многих таких констант (6.4.4.4/11).
- Будут ли конкатенированы различные префиксы с широким строковым литеральным литералом, и если да, то обработка полученной многобайтовой последовательности символов (6.4.5 / 5)
- Локаль, используемая во время фазы перевода 7, для преобразования широких строковых литералов в многобайтовые последовательности символов и их значения, когда результат не представлен в наборе символов выполнения (6.4.5 / 6).

- Способ, которым имена заголовков сопоставляются с именами файлов (6.4.7 / 2).

оценка

- Независимо от того, `FP_CONTRACT` ли выражения с плавающей точкой, когда `FP_CONTRACT` не используется (6.5 / 8).
- **Значения результатов операторов `sizeof` и `_Alignof`** (6.5.3.4/5).
- Размер результата результата вычитания указателя (6.5.6 / 9).
- **Результат правого смещения знакового целого с отрицательным значением** (6.5.7 / 5).

Поведение во время выполнения

- Степень, в которой ключевое слово `register` действует (6.7.1 / 6).
- Является ли тип битового поля, объявленного как `int` , тем же типом, что и `unsigned int` или как `signed int` (6.7.2 / 5).
- Какие типы бит могут принимать, кроме необязательно квалифицированных `_Bool` , `signed int` и `unsigned int` ; могут ли битовые поля иметь атомные типы (6.7.2.1/5).
- Аспекты того, как реализации выкладывают хранилище для бит-полей (6.7.2.1/11).
- Выравнивание небитных элементов структур и объединений (6.7.2.1/14).
- Основной тип для каждого перечисленного типа (6.7.2.2/4).
- Что представляет собой «доступ» к объекту `volatile` -qualifed типа (6.7.3 / 7).
- Эффективность объявлений `inline` функций (6.7.4 / 6).

препроцессор

- Точно ли символьные константы преобразуются в целые значения таким же образом в условных выражениях препроцессора, как в обычных выражениях, и может ли односимвольная константа иметь отрицательное значение (6.10.1 / 4).
- Места искали файлы, указанные в директиве `#include` (6.10.2 / 2-3).
- Способ, с помощью которого имя заголовка формируется из токенов многоточечной директивы `#include` (6.10.2 / 4).
- Предел для `#include` nesting (6.10.2 / 6).
- Вставляет ли символ `\` перед `\` введением универсального символьного имени в

результате выполнения оператора `#` препроцессора (6.10.3.2/2).

- Поведение директивы `STDC #pragma` для прагм, отличных от `STDC` (6.10.6 / 1).
- Значение макросов `__DATE__` и `__TIME__` если нет даты или времени перевода, соответственно, доступно (6.10.8.1/1).
- Внутренняя кодировка символов, используемая для `wchar_t` если макрос `__STDC_ISO_10646__` не определен (6.10.8.2/1).
- Внутренняя кодировка символов, используемая для `char32_t` если макрос `__STDC_UTF_32__` не определен (6.10.8.2/1).

Стандартная библиотека

генеральный

- Формат сообщений, выдаваемых при неудачах утверждений (7.2.1.1/2).

Функции среды с плавающей запятой

- Любые дополнительные исключения с плавающей запятой, за исключением тех, которые определены стандартом (7.6 / 6).
- Любые дополнительные режимы округления с плавающей запятой, отличные от тех, которые определены стандартом (7.6 / 8).
- Любые дополнительные среды с плавающей точкой, отличные от тех, которые определены стандартом (7.6 / 10).
- Значение по умолчанию для переключателя доступа к среде с плавающей запятой (7.6.1 / 2).
- Представление флагов состояния с плавающей запятой, записанных `fegetexceptflag()` (7.6.2.2/1).
- Является ли `feraiseexcept()` дополнительно поднимает «неточное» исключение с плавающей запятой всякий раз, когда оно вызывает исключение с плавающей запятой «overflow» или « underflow» (7.6.2.3/2).

Локальные функции

- Строки locale, отличные от `"C"` поддерживаемые `setlocale()` (7.11.1.1/3).

Математические функции

- Типы, представленные `float_t` и `double_t` когда макрос `FLT_EVAL_METHOD` имеет значение, отличное от `0` , `1` и `2` (7.12 / 2).
- Любые поддерживаемые классы с плавающей запятой, отличные от тех, которые определены стандартом (7.12 / 6).
- Значение, возвращаемое функциями `math.h` в случае ошибки домена (7.12.1 / 2).
- Значение, возвращаемое функциями `math.h` в случае ошибки полюса (7.12.1 / 3).
- Значение, возвращаемое функциями `math.h` когда результат заканчивается, и аспекты того, установлено ли `errno` в `ERANGE` и возникает ли исключение с плавающей точкой в этих обстоятельствах (7.12.1 / 6).
- Значение по умолчанию для FP-сокращения (7.12.2 / 2).
- Независимо от того, возвращают ли функции `fmod()` 0 или повышают ошибку домена, когда их второй аргумент равен 0 (7.12.10.1/3).
- Независимо от того, возвращают ли функции `remainder()` 0 или повышают ошибку домена, когда их второй аргумент равен 0 (7.12.10.2/3).
- Число значимых битов в факториальных модулях, вычисляемых `remquo()` (7.12.10.3/2).
- `remquo()` ли функции `remquo()` 0 или повышают ошибку домена, когда их второй аргумент равен 0 (7.12.10.3/3).

сигналы

- Полный набор поддерживаемых сигналов, их семантика и обработка по умолчанию (7.14 / 4).
- Когда сигнал поднимается и имеется специальный обработчик, связанный с этим сигналом, какие сигналы, если они есть, блокируются на время выполнения обработчика (7.14.1.1/3).
- Какие сигналы, отличные от `SIGFPE` , `SIGILL` и `SIGSEGV` приводят к тому, что поведение при возврате из пользовательского обработчика сигнала не определено (7.14.1.1/3).
- Какие сигналы изначально настроены для игнорирования (независимо от их обработки по умолчанию, 7.14.1.1/6).

Разнообразный

- Конкретная константа нулевого указателя, к которой расширяется макрос `NULL` (7.19 / 3).

Функции обработки файлов

- Требуется ли в последней строке текстового потока завершающая строка новой строки (7.21.2 / 2).
- Число нулевых символов, автоматически добавленных к двоичному потоку (7.21.2 / 3).
- Начальная позиция файла, открытого в режиме добавления (7.21.3 / 1).
- Является ли запись в текстовом потоке причиной усечения потока (7.21.3 / 2).
- Поддержка буферизации потоков (7.21.3 / 3).
- Существуют ли файлы нулевой длины (7.21.3 / 4).
- Правила составления допустимых имен файлов (7.21.3 / 8).
- Можно ли одновременно открыть один и тот же файл несколько раз (7.21.3 / 8).
- Характер и выбор кодирования для многобайтовых символов (7.21.3 / 10).
- Поведение функции `remove()` при открытии целевого файла (7.21.4.1/2).
- Поведение функции `rename()` когда целевой файл уже существует (7.21.4.2/2).
- `tmpfile()` ли файлы, созданные с помощью функции `tmpfile()` , в случае, если программа завершится ненормально (7.21.4.3/2).
- Какой режим изменяется, при каких обстоятельствах разрешается использование `freopen()` (7.21.5.4/3).

Функции ввода / вывода

- Какое из разрешенных представлений значений FP бесконечного и не-номера производится функциями printf () - семейства (7.21.6.1/8).
- Способ, с помощью которого указатели форматируются функциями `printf()` - семейства (7.21.6.1/8).
- Поведение функции `scanf()` -family, когда символ `-` появляется во внутреннем положении списка сканирования поля `[` 7.21.6.2/12].
- Большинство аспектов функции `scanf()` -семейных функций для `p` полей (7.21.6.2/12).
- Значение `errno` заданное функцией `fgetpos()` при сбое (7.21.9.1/2).
- Значение `errno` заданное `fsetpos()` при ошибке (7.21.9.3/2).

- Значение `errno` установленное `ftell()` при сбое (7.21.9.4/3).
- Значение для `strtod()` -семейных функций некоторых поддерживаемых аспектов форматирования NaN (7.22.1.3p4).
- Независимо от того, функции `strtod()` -family устанавливают `errno` в `ERANGE` когда результат заканчивается (7.22.1.3/10).

Функции распределения памяти

- Поведение функций распределения памяти, когда количество запрошенных байтов равно 0 (7.22.3 / 1).

Функции системной среды

- Какие очистки, если они есть, и какой статус возвращается ОС хоста при вызове функции `abort()` (7.22.4.1/2).
- Какой статус возвращается в среду хоста при вызове `exit()` (7.22.4.4/5).
- Обработка открытых потоков и какой статус возвращается в среду хоста при `_Exit()` (7.22.4.5/2).
- Набор названий среды, доступных через `getenv()` и метод изменения среды (7.22.4.6/2).
- Возвращаемое значение функции `system()` (7.22.4.8/3).

Функции даты и времени

- Местный часовой пояс и летнее время (7.27.1 / 1).
- Диапазон и точность времени, представляемого через типы `clock_t` и `time_t` (7.27.1 / 4).
- Начало эры, которая служит ссылкой на время, возвращаемое функцией `clock()` (7.27.2.1/3).
- Начало эпохи, которая служит ссылкой на время, возвращаемое `timespec_get()` (когда временной базой является `TIME_UTC` ; 7.27.2.5/3).
- Замена `strftime()` для спецификатора преобразования `%Z` в локали «C» (7.27.3.5/7).

Широкосимвольные функции ввода / вывода

- Какое из разрешенных представлений бесконечных и не-числовых значений FP создаются функциями `wprintf()` -семейства (7.29.2.1/8).

- Способ, с помощью которого указатели форматируются функциями `wprintf()` -семейства (7.29.2.1/8).
- Поведение `wscanf()` -семейства действует, когда символ `-` появляется во внутреннем положении списка сканирования поля `[` 7.29.2.2/12].
- Большинство аспектов работы `wscanf()` -семейных функций `p` полей (7.29.2.2/12).
- Значение для `wstrtod()` -семейных функций некоторых поддерживаемых аспектов форматирования NaN (7.29.4.1.1 / 4).
- Независимо от того, функции `wstrtod()` -family устанавливают `errno` в `ERANGE` когда результат заканчивается (7.29.4.1.1 / 10).

Examples

Правый сдвиг отрицательного целого числа

```
int signed_integer = -1;

// The right shift operation exhibits implementation-defined behavior:
int result = signed_integer >> 1;
```

Назначение значения вне диапазона для целого числа

```
// Supposing SCHAR_MAX, the maximum value that can be represented by a signed char, is
// 127, the behavior of this assignment is implementation-defined:
signed char integer;
integer = 128;
```

Выделение нулевых байтов

```
// The allocation functions have implementation-defined behavior when the requested size
// of the allocation is zero.
void *p = malloc(0);
```

Представление знаковых целых чисел

Каждый знаковый целочисленный тип может быть представлен в любом из трех форматов; это определяется реализацией, какая из них используется. Реализация в использовании для любого данного подписанного целого типа , по крайней мере так велик , как `int` может быть определено во время выполнения из двух битов низшего порядка представления значения `-1` в том виде, например , так:

```
enum { sign_magnitude = 1, ones_compl = 2, twos_compl = 3, };
#define SIGN_REP(T) ((T)-1 & (T)3)
```

```
switch (SIGN_REP(long)) {
   case sign_magnitude: { /* do something */ break; }
   case ones_compl:     { /* do otherwise */ break; }
   case twos_compl:     { /* do yet else  */ break; }
   case 0:  { _Static_assert(SIGN_REP(long), "bogus sign representation"); }
}
```

Тот же шаблон применяется к представлению более узких типов, но они не могут быть проверены этим методом, потому что операнды `&` подлежат «обычным арифметическим преобразованиям» перед вычислением результата.

Прочитайте Поведение, определяемое реализацией онлайн:
https://riptutorial.com/ru/c/topic/4832/поведение--определяемое-реализацией

глава 43: Препроцессор и макросы

Вступление

Все команды препроцессора начинаются с символа хеша (фунта) `#` . Макрос AC - это только команда препроцессора, которая определяется с помощью `#define` препроцессора `#define` . На этапе предварительной обработки препроцессор C (часть компилятора C) просто заменяет тело макроса везде, где появляется его имя.

замечания

Когда компилятор встречает макрос в коде, он выполняет простую замену строк, никаких дополнительных операций не выполняется. Из-за этого изменения препроцессора не учитывают область программ C - например, определение макроса не ограничено тем, что оно находится внутри блока, поэтому на него не влияет `'}'` который завершает оператор блока.

Препроцессор, по замыслу, не завершается полным - существует несколько типов вычислений, которые не могут быть выполнены только препроцессором.

Обычно у компиляторов есть флаг командной строки (или параметр конфигурации), который позволяет нам прекратить компиляцию после фазы предварительной обработки и проверить результат. На платформах POSIX этот флаг равен `-E` . Таким образом, запуск gcc с этим флагом печатает расширенный источник в stdout:

```
$ gcc -E cprog.c
```

Часто препроцессор реализуется как отдельная программа, которая вызывается компилятором, общее имя для этой программы - `cpp` . Ряд препроцессоров выделяет вспомогательную информацию, такую как информация о номерах строк, которая используется для последующих этапов компиляции для генерации отладочной информации. В случае, когда препроцессор основан на gcc, опция -P подавляет такую информацию.

```
$ cpp -P cprog.c
```

Examples

Условное включение и модификация подписи условных функций

Чтобы условно включить блок кода, препроцессор имеет несколько директив (например, `#if` , `#ifdef` , `#else` , `#endif` и т. Д.).

```
/* Defines a conditional `printf` macro, which only prints if `DEBUG`
 * has been defined
 */
#ifdef DEBUG
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Нормальные операторы отношения C могут использоваться для условия `#if`

```
#if __STDC_VERSION__ >= 201112L
/* Do stuff for C11 or higher */
#elif __STDC_VERSION__ >= 199901L
/* Do stuff for C99 */
#else
/* Do stuff for pre C99 */
#endif
```

Директивы `#if` ведут себя аналогично выражению C `if` , он должен содержать только интегральные константные выражения и не выполнять никаких бросков. Он поддерживает один дополнительный унарный оператор, `defined( identifier )` , который возвращает `1` если идентификатор определен, и `0` противном случае.

```
#if defined(DEBUG) && !defined(QUIET)
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Модификация подписи условной функции

В большинстве случаев ожидается, что выпускная версия приложения будет иметь как можно меньше накладных расходов. Однако при тестировании промежуточной сборки могут оказаться полезными дополнительные журналы и информация о найденных проблемах.

Например, предположим, что существует некоторая функция `SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)` которая при выполнении тестовой сборки хочет создать журнал об использовании. Однако эта функция используется во многих местах, и желательно, чтобы при генерации журнала часть информации заключалась в том, чтобы знать, откуда вызывается функция.

Поэтому, используя условную компиляцию, вы можете иметь что-то вроде следующего в include-файле, объявляющем функцию. Это заменяет стандартную версию функции отладочной версией функции. Препроцессор используется для замены вызовов функции `SerOpPluAllRead()` с вызовами функции `SerOpPluAllRead_Debug()` с двумя дополнительными аргументами, именем файла и номером строки, где используется функция.

Условная компиляция используется для выбора того, следует ли переопределять

стандартную функцию с помощью отладочной версии или нет.

```
#if 0
// function declaration and prototype for our debug version of the function.
SHORT   SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo);

// macro definition to replace function call using old name with debug function with
additional arguments.
#define SerOpPluAllRead(pPif,usLock) SerOpPluAllRead_Debug(pPif,usLock,__FILE__,__LINE__)
#else
// standard function declaration that is normally used with builds.
SHORT   SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd);
#endif
```

Это позволяет переопределить стандартную версию функции `SerOpPluAllRead()` с версией, которая будет содержать имя файла и номер строки в файле, где вызывается функция.

Существует одно важное соображение: *любой файл, использующий эту функцию, должен включать заголовочный файл, в котором используется этот подход, чтобы препроцессор мог изменить функцию. В противном случае вы увидите ошибку компоновщика.*

Определение функции будет выглядеть примерно так: То, что этот источник делает, - это запросить, чтобы препроцессор переименовал функцию `SerOpPluAllRead()` в `SerOpPluAllRead_Debug()` и `SerOpPluAllRead_Debug()` список аргументов, чтобы включить два дополнительных аргумента, указатель на имя файла, в котором была вызвана функция, и номер строки в файле, в котором используется эта функция.

```
#if defined(SerOpPluAllRead)
// forward declare the replacement function which we will call once we create our log.
SHORT    SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd);

SHORT    SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo)
{
    int iLen = 0;
    char  xBuffer[256];

    // only print the last 30 characters of the file name to shorten the logs.
    iLen = strlen (aszFilePath);
    if (iLen > 30) {
        iLen = iLen - 30;
    }
    else {
        iLen = 0;
    }

    sprintf (xBuffer, "SerOpPluAllRead_Debug(): husHandle = %d, File %s, lineno = %d", pPif-
>husHandle, aszFilePath + iLen, nLineNo);
    IssueDebugLog(xBuffer);

    // now that we have issued the log, continue with standard processing.
    return SerOpPluAllRead_Special(pPif, usLockHnd);
}

// our special replacement function name for when we are generating logs.
```

```
SHORT    SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd)
#else
// standard, normal function name (signature) that is replaced with our debug version.
SHORT   SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)
#endif
{
    if (STUB_SELF == SstReadAsMaster()) {
        return OpPluAllRead(pPif, usLockHnd);
    }
    return OP_NOT_MASTER;
}
```

Включение исходного файла

Наиболее распространенными применениями директив `#include` preprocessing являются следующие:

```
#include <stdio.h>
#include "myheader.h"
```

`#include` заменяет инструкцию содержимым указанного файла. Угловые скобки (<>) относятся к файлам заголовков, установленным в системе, а кавычки ("") предназначены для файлов, предоставленных пользователем.

Макросы сами могут развернуть другие макросы один раз, так как этот пример иллюстрирует:

```
#if VERSION == 1
    #define INCFILE  "vers1.h"
#elif VERSION == 2
    #define INCFILE  "vers2.h"
    /*  and so on */
#else
    #define INCFILE  "versN.h"
#endif
/* ... */
#include INCFILE
```

Замена макросов

Простейшей формой замены макросов является определение `manifest constant` , как в

```
#define ARRSIZE 100
int array[ARRSIZE];
```

Это определяет *функционально-подобный* макрос, который умножает переменную на `10` и сохраняет новое значение:

```
#define TIMES10(A) ((A) *= 10)

double b = 34;
```

```
int c = 23;

TIMES10(b);   // good: ((b) *= 10);
TIMES10(c);   // good: ((c) *= 10);
TIMES10(5);   // bad:  ((5) *= 10);
```

Замена выполняется до любой другой интерпретации текста программы. При первом вызове `TIMES10` имя `A` из определения заменяется на `b` а затем расширенный текст помещается вместо вызова. Заметим, что это определение `TIMES10` не эквивалентно

```
#define TIMES10(A) ((A) = (A) * 10)
```

потому что это могло бы оценить замену `A` , дважды, что может иметь нежелательные побочные эффекты.

Далее определяется функционально-подобный макрос, значение которого является максимумом его аргументов. Он имеет преимущества работы для любых совместимых типов аргументов и генерации встроенного кода без накладных расходов на вызов функции. У него есть недостатки, связанные с оценкой одного или другого из его аргументов во второй раз (включая побочные эффекты) и генерирования большего количества кода, чем функция, если она вызывается несколько раз.

```
#define max(a, b) ((a) > (b) ? (a) : (b))

int maxVal = max(11, 43);              /* 43 */
int maxValExpr = max(11 + 36, 51 - 7); /* 47 */

/* Should not be done, due to expression being evaluated twice */
int j = 0, i = 0;
int sideEffect = max(++i, ++j);       /* i == 4 */
```

Из-за этого такие макросы, которые оценивают свои аргументы несколько раз, обычно избегают в производственном коде. Начиная с C11 существует функция `_Generic` которая позволяет избежать таких множественных вызовов.

Обильные круглые скобки в макрорасширениях (правая часть определения) гарантируют, что аргументы и результирующее выражение связаны должным образом и хорошо вписываются в контекст, в котором вызывается макрос.

Директива об ошибках

Если препроцессор встречает директиву `#error` , компиляция прекращается и включается диагностическое сообщение.

```
#define DEBUG

#ifdef DEBUG
#error "Debug Builds Not Supported"
#endif
```

```
int main(void) {
    return 0;
}
```

Возможный выход:

```
$ gcc error.c
error.c: error: #error "Debug Builds Not Supported"
```

#if 0 для блокировки разделов кода

Если есть разделы кода, который вы планируете удалить или хотите временно отключить, вы можете прокомментировать его с комментарием блока.

```
/* Block comment around whole function to keep it from getting used.
 * What's even the purpose of this function?
int myUnusedFunction(void)
{
    int i = 5;
    return i;
}
*/
```

Однако, если исходный код, который вы окружили блочным комментарием, имеет комментарии к блочному стилю в источнике, окончание * / существующих комментариев блока может привести к тому, что ваш новый комментарий блока будет недействительным и вызовет проблемы с компиляцией.

```
/* Block comment around whole function to keep it from getting used.
 * What's even the purpose of this function?
int myUnusedFunction(void)
{
    int i = 5;

    /* Return 5 */
    return i;
}
*/
```

В предыдущем примере последние две строки функции и последний '* /' рассматриваются компилятором, поэтому он компилируется с ошибками. Более безопасный метод - использовать `#if 0` вокруг кода, который вы хотите заблокировать.

```
#if 0
/* #if 0 evaluates to false, so everything between here and the #endif are
 * removed by the preprocessor. */
int myUnusedFunction(void)
{
    int i = 5;
    return i;
}
```

```
#endif
```

Преимущество этого в том, что, когда вы хотите вернуться и найти код, гораздо проще выполнить поиск «#if 0», чем поиск всех ваших комментариев.

Еще одно очень важное преимущество заключается в том, что вы можете вставлять код комментария с `#if 0` . Это невозможно сделать с комментариями.

Альтернативой использованию `#if 0` является использование имени, которое не будет `#defined` но более подробно `#defined` почему код блокируется. Например, если есть функция, которая кажется бесполезным мертвым кодом, вы можете использовать `#if defined(POSSIBLE_DEAD_CODE)` или `#if defined(FUTURE_CODE_REL_020201)` для кода, необходимого после того, как будут установлены другие функции или что-то подобное. Затем, когда вы возвращаетесь, чтобы удалить или включить этот источник, эти разделы источника легко найти.

Маркировка токенов

Вставка меток позволяет склеить два макро аргумента. Например, `front##back frontback` . Известный пример - это заголовок `<TCHAR.H>` Win32. В стандарте C можно написать `L"string"` чтобы объявить широкую строку символов. Тем не менее, Windows API позволяет конвертировать между широкими символьными строками и узкими символьными строками просто `#define UNICODE` . Чтобы реализовать строковые литералы, `TCHAR.H` использует это

```
#ifdef UNICODE
#define TEXT(x) L##x
#endif
```

Всякий раз, когда пользователь записывает `TEXT("hello, world")` и UNICODE, препроцессор C объединяет `L` и аргумент макроса. `L` с `"hello, world"` дает `L"hello, world"` .

Предопределенные макросы

Предопределенный макрос - это макрос, который уже понимается процессором C без программы, требующей ее определения. Примеры включают

Обязательные предопределенные макросы

- `__FILE__` , который дает имя файла текущего исходного файла (строковый литерал),
- `__LINE__` для текущего номера строки (целочисленная константа),
- `__DATE__` для даты компиляции (строковый литерал)
- `__TIME__` для времени компиляции (строковый литерал).

Существует также связанный предопределенный идентификатор `__func__` (ISO / IEC 9899: 2011 §6.4.2.2), который *не* является макросом:

Идентификатор `__func__` должен быть неявно объявлен переводчиком так, как если бы сразу после открытия скобки каждого определения функции было объявлено:

```
static const char __func__[] = "function-name";
```

, где *function-name* - это имя лексически-охватывающей функции.

`__FILE__` , `__LINE__` и `__func__` особенно полезны для целей отладки. Например:

```
fprintf(stderr, "%s: %s: %d: Denominator is 0", __FILE__, __func__, __LINE__);
```

Компиляторы Pre-C99 могут поддерживать или не поддерживать `__func__` или могут иметь макрос, который действует так же, как и по-другому. Например, gcc использовал `__FUNCTION__` в режиме C89.

В приведенных ниже макросах можно задать подробные сведения о реализации:

- `__STDC_VERSION__` версия стандарта C. Это постоянное целое число, использующее формат `yyyymmL` (значение `201112L` для C11, значение `199901L` для C99, оно не было определено для C89 / C90)
- `__STDC_HOSTED__` `1` если это хостинг-реализация, иначе `0` .
- `__STDC__` Если `1` , реализация соответствует стандарту C.

Другие предварительно определенные макросы (не обязательно)

ISO / IEC 9899: 2011 §6.10.9.2 Макросы окружения:

- `__STDC_ISO_10646__` Целочисленная константа формы `yyyymmL` (например, 199712L). Если этот символ определен, то каждый символ в кодировке Unicode, заданный при хранении в объекте типа `wchar_t` , имеет то же значение, что и короткий идентификатор этого символа. Требуемый Unicode набор состоит из всех символов, которые определены ISO / IEC 10646, а также всех поправок и технических исправлений на указанный год и месяц. Если используется некоторая другая кодировка, макрос не должен определяться, а фактическая кодировка используется для реализации.

- `__STDC_MB_MIGHT_NEQ_WC__` Целочисленная константа 1, предназначенная для указания того, что в кодировке для `wchar_t` член базового набора символов не должен иметь кодового значения, равного его значению, когда он используется как одиночный символ в целочисленной символьной константе.

- `__STDC_UTF_16__` Целочисленная константа 1, предназначенная для указания того, что значения типа `char16_t` кодируются в кодировке UTF-16. Если используется некоторая другая кодировка, макрос не должен определяться, а фактическая кодировка используется для реализации.

- `__STDC_UTF_32__` Целая константа 1, предназначенная для указания того, что значения типа `char32_t` кодируются в кодировке UTF-32. Если используется некоторая другая кодировка, макрос не должен определяться, а фактическая кодировка используется для реализации.

ISO / IEC 9899: 2011 §6.10.8.3. Условные функциональные макросы

- `__STDC_ANALYZABLE__` Целая константа 1, предназначенная для указания соответствия спецификациям в приложении L (Анализируемость).
- `__STDC_IEC_559__` Целая константа 1, предназначенная для указания соответствия спецификациям в приложении F (IEC 60559 с плавающей точкой арифметики).
- `__STDC_IEC_559_COMPLEX__` Целочисленная константа 1, предназначенная для указания соответствия спецификациям в приложении G (совместимая с IEC 60559 комплексная арифметика).
- `__STDC_LIB_EXT1__` Целая константа `201112L` , предназначенная для указания поддержки расширений, определенных в приложении K (Интерфейсы проверки границ).
- `__STDC_NO_ATOMICS__` Целая константа 1, предназначенная для указания того, что реализация не поддерживает атомные типы (включая классификатор типа `_Atomic`) и заголовок `<stdatomic.h>` .
- `__STDC_NO_COMPLEX__` Целочисленная константа 1, предназначенная для указания того, что реализация не поддерживает сложные типы или заголовок `<complex.h>` .
- `__STDC_NO_THREADS__` Целочисленная константа 1, предназначенная для указания того, что реализация не поддерживает заголовок `<threads.h>` .
- `__STDC_NO_VLA__` Целочисленная константа 1, предназначенная для указания того, что реализация не поддерживает массивы переменной длины или измененные типы.

Заголовки включают охранников

Практически каждый заголовочный файл должен следить за идиомой include guard :

мой-заголовок-file.h

```
#ifndef MY_HEADER_FILE_H
#define MY_HEADER_FILE_H

// Code body for header file
```

```
#endif
```

Это гарантирует, что когда вы `#include "my-header-file.h"` в нескольких местах, вы не получаете дубликатов деклараций функций, переменных и т. Д. Представьте себе следующую иерархию файлов:

Заголовок-1.h

```
typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);
```

Заголовок-2.h

```
#include "header-1.h"

int myFunction2(MyStruct *value);
```

main.c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

Этот код имеет серьезную проблему: подробное содержимое `MyStruct` определено дважды, что недопустимо. Это приведет к ошибке компиляции, которую трудно отследить, поскольку один заголовочный файл содержит другой. Если вы сделали это с помощью защиты заголовков:

Заголовок-1.h

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif
```

Заголовок-2.h

```
#ifndef HEADER_2_H
```

```
#define HEADER_2_H

#include "header-1.h"

int myFunction2(MyStruct *value);

#endif
```

main.c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

Затем это расширилось бы до:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif

#ifndef HEADER_2_H
#define HEADER_2_H

#ifndef HEADER_1_H // Safe, since HEADER_1_H was #define'd before.
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif

int myFunction2(MyStruct *value);

#endif

int main() {
    // do something
}
```

Когда компилятор достигает второго включения **header-** `HEADER_1_H` , `HEADER_1_H` уже был определен предыдущим включением. Эрго, это сводится к следующему:

```
#define HEADER_1_H
```

```
typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#define HEADER_2_H

int myFunction2(MyStruct *value);

int main() {
    // do something
}
```

И, следовательно, ошибка компиляции отсутствует.

Примечание. Для обозначения защиты заголовков существует несколько различных соглашений. Некоторым людям нравится называть его `HEADER_2_H_` , некоторые включают название проекта, например `MY_PROJECT_HEADER_2_H` . Важно следить за тем, чтобы принятая вами конвенция делала так, чтобы каждый файл в вашем проекте имел уникальную защиту заголовка.

Если детали структуры не были включены в заголовок, объявленный тип был бы неполным или непрозрачным . Такие типы могут быть полезными, скрывая детали реализации от пользователей функций. Для многих целей тип `FILE` в стандартной библиотеке C можно рассматривать как непрозрачный тип (хотя он обычно не является непрозрачным, так что реализация макросов стандартных функций ввода-вывода может использовать внутренние элементы структуры). В этом случае `header-1.h` может содержать:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct MyStruct MyStruct;

int myFunction(MyStruct *value);

#endif
```

Обратите внимание, что структура должна иметь имя тега (здесь `MyStruct` - это пространство имен тегов, отдельно от пространства имен обычных идентификаторов имени typedef `MyStruct`) и что `{ … }` опущен. Это говорит о том, что существует структура `struct MyStruct` и для нее есть псевдоним `MyStruct` .

В файле реализации детали структуры могут быть определены для завершения типа:

```
struct MyStruct {
    …
};
```

Если вы используете C11, вы можете повторить `typedef struct MyStruct MyStruct;`

объявление, не вызывая ошибки компиляции, но более ранние версии C будут жаловаться. Следовательно, по-прежнему лучше использовать идиому include guard, хотя в этом примере было бы необязательно, если бы код был только компилирован с компиляторами, поддерживающими C11.

Многие компиляторы поддерживают директиву `#pragma once` , которая имеет те же результаты:

мой-заголовок-file.h

```
#pragma once

// Code for header file
```

Однако `#pragma once` не является частью стандарта C, поэтому код менее портативен, если вы его используете.

Несколько заголовков не используют идиому include guard. Один конкретный пример - стандартный заголовок `<assert.h>` . Он может быть включен несколько раз в одну единицу перевода, и эффект от этого зависит от того, определяется ли макрос `NDEBUG` каждый раз, когда заголовок включен. Иногда у вас может быть аналогичное требование; таких случаев будет мало и далеко. Как правило, ваши заголовки должны быть защищены включением охранной идиомы.

Внедрение

Мы также можем использовать макросы для упрощения чтения и записи кода. Например, мы можем реализовать макросы для реализации конструкции `foreach` в C для некоторых структур данных, таких как одиночные и двусвязные списки, очереди и т. Д.

Вот небольшой пример.

```
#include <stdio.h>
#include <stdlib.h>

struct LinkedListNode
{
    int data;
    struct LinkedListNode *next;
};

#define FOREACH_LIST(node, list) \
     for (node=list; node; node=node->next)

/* Usage */
int main(void)
{
    struct LinkedListNode *list, **plist = &list, *node;
```

```
    int i;

    for (i=0; i<10; i++)
    {
         *plist = malloc(sizeof(struct LinkedListNode));
         (*plist)->data = i;
         (*plist)->next = NULL;
         plist          = &(*plist)->next;
    }

    /* printing the elements here */
    FOREACH_LIST(node, list)
    {
        printf("%d\n", node->data);
    }
}
```

Вы можете создать стандартный интерфейс для таких структур данных и написать общую реализацию `FOREACH` как:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct CollectionItem_
{
    int data;
    struct CollectionItem_ *next;
} CollectionItem;

typedef struct Collection_
{
    /* interface functions */
    void* (*first)(void *coll);
    void* (*last) (void *coll);
    void* (*next) (void *coll, CollectionItem *currItem);

    CollectionItem *collectionHead;
    /* Other fields */
} Collection;

/* must implement */
void *first(void *coll)
{
    return ((Collection*)coll)->collectionHead;
}

/* must implement */
void *last(void *coll)
{
    return NULL;
}

/* must implement */
void *next(void *coll, CollectionItem *curr)
{
    return curr->next;
}

CollectionItem *new_CollectionItem(int data)
```

```
{
    CollectionItem *item = malloc(sizeof(CollectionItem));
    item->data = data;
    item->next = NULL;
    return item;
}

void Add_Collection(Collection *coll, int data)
{
    CollectionItem **item = &coll->collectionHead;
    while(*item)
        item = &(*item)->next;
    (*item) = new_CollectionItem(data);
}

Collection *new_Collection()
{
    Collection *nc = malloc(sizeof(Collection));
    nc->first = first;
    nc->last  = last;
    nc->next  = next;
    return nc;
}

/* generic implementation */
#define FOREACH(node, collection)                       \
    for (node  = (collection)->first(collection);       \
         node != (collection)->last(collection);        \
         node  = (collection)->next(collection, node))

int main(void)
{
    Collection *coll = new_Collection();
    CollectionItem *node;
    int i;

    for(i=0; i<10; i++)
    {
         Add_Collection(coll, i);
    }

    /* printing the elements here */
    FOREACH(node, coll)
    {
        printf("%d\n", node->data);
    }
}
```

Чтобы использовать эту общую реализацию, просто выполните эти функции для своей структуры данных.

```
1.  void* (*first)(void *coll);
2.  void* (*last) (void *coll);
3.  void* (*next) (void *coll, CollectionItem *currItem);
```

__cplusplus для использования C-externals в коде C ++, скомпилированном с C ++-name mangling

Бывают случаи, когда включаемый файл должен генерировать другой вывод из препроцессора в зависимости от того, является ли компилятор компилятором C или компилятором C ++ из-за различий в языке.

Например, функция или другая внешняя определяется в исходном файле C, но используется в исходном файле C ++. Поскольку C ++ использует управление именами (или украшение имен), чтобы генерировать уникальные имена функций на основе типов аргументов функции, объявление функции C, используемое в исходном файле C ++, приведет к ошибкам ссылок. Компилятор C ++ изменит указанное внешнее имя для выхода компилятора, используя правила управления именами для C ++. Результатом являются ошибки ссылок из-за внешних символов, которые не найдены, когда вывод компилятора C ++ связан с выходом компилятора C.

Поскольку компиляторы C не обрабатывают имя, но компиляторы C ++ делают для всех внешних ярлыков (имен функций или имен переменных), сгенерированных компилятором C ++, был введен предопределенный макрос препроцессора `__cplusplus` , чтобы разрешить обнаружение компилятора.

Чтобы обойти эту проблему несовместимого вывода компилятора для внешних имен между C и C ++, макрос `__cplusplus` определен в препроцессоре C ++ и не определен в препроцессоре C. Это имя макроса можно использовать с условной препроцессором директивы `#ifdef` или `#if` с помощью оператора `defined()` чтобы определить, компилируется ли исходный код или файл include как C ++ или C.

```
#ifdef __cplusplus
printf("C++\n");
#else
printf("C\n");
#endif
```

Или вы можете использовать

```
#if defined(__cplusplus)
printf("C++\n");
#else
printf("C\n");
#endif
```

Чтобы указать правильное имя функции функции из исходного файла C, скомпилированного с помощью компилятора C, который используется в исходном файле C ++, вы можете проверить определенную константу `__cplusplus` , чтобы вызвать `extern "C" { /* ... */ };` для использования, чтобы объявить C внешними, когда заголовочный файл включен в исходный файл на C ++. Однако при компиляции с компилятором C `extern "C" { */ ... */ };` не используется. Эта условная компиляция необходима, потому что `extern "C" { /* ... */ };` действителен в C ++, но не в C.

```
#ifdef __cplusplus
```

```
// if we are being compiled with a C++ compiler then declare the
// following functions as C functions to prevent name mangling.
extern "C" {
#endif

// exported C function list.
int foo (void);

#ifdef __cplusplus
// if this is a C++ compiler, we need to close off the extern declaration.
};
#endif
```

Функциональные макросы

Функциональные макросы аналогичны `inline` функциям, они полезны в некоторых случаях, например, временный журнал отладки:

```
#ifdef DEBUG
# define LOGFILENAME "/tmp/logfile.log"

# define LOG(str) do {                             \
  FILE *fp = fopen(LOGFILENAME, "a");             \
  if (fp) {                                        \
    fprintf(fp, "%s:%d %s\n", __FILE__, __LINE__, \
                /* don't print null pointer */    \
                str ?str :"<null>");               \
    fclose(fp);                                    \
  }                                                \
  else {                                           \
    perror("Opening '" LOGFILENAME "' failed");    \
  }                                                \
} while (0)
#else
  /* Make it a NOOP if DEBUG is not defined. */
# define LOG(LINE) (void)0
#endif


#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc > 1)
        LOG("There are command line arguments");
    else
        LOG("No command line arguments");
    return 0;
}
```

Здесь в обоих случаях (с `DEBUG` или нет) вызов ведет себя так же, как функция с типом возврата `void` . Это гарантирует, что условия `if/else` интерпретируются как ожидаемые.

В случае `DEBUG` это реализуется через конструкцию `do { ... } while(0)` . В другом случае `(void)0` - это оператор без побочного эффекта, который просто игнорируется.

Альтернативой для последнего было бы

```
#define LOG(LINE) do { /* empty */ } while (0)
```

так что он во всех случаях синтаксически эквивалентен первому.

Если вы используете GCC, вы также можете реализовать макрос функции, который возвращает результат с использованием выражений выражения нестандартного выражения GNU. Например:

```
#include <stdio.h>

#define POW(X, Y) \
({ \
        int i, r = 1; \
        for (i = 0; i < Y; ++i) \
                r *= X; \
        r; \ // returned value is result of last operation
})

int main(void)
{
        int result;

        result = POW(2, 3);
        printf("Result: %d\n", result);
}
```

Матричные аргументы Variadic

C99

Макросы с переменными аргументами:

Предположим, вы хотите создать некоторый макрос печати для отладки вашего кода, давайте возьмем этот макрос в качестве примера:

```
#define debug_print(msg) printf("%s:%d %s", __FILE__, __LINE__, msg)
```

Некоторые примеры использования:

Функция `somefunc()` возвращает -1, если не удалось, и 0, если она выполнена успешно, и вызывается из множества разных мест внутри кода:

```
int retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}

/* some other code */
```

```
 retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}
```

Что произойдет, если реализация `somefunc()` изменится, и теперь она возвращает разные значения, соответствующие различным возможным типам ошибок? Вы все еще хотите использовать макрос отладки и напечатать значение ошибки.

```
debug_printf(retVal);      /* this would obviously fail */
debug_printf("%d",retVal); /* this would also fail */
```

Для решения этой проблемы был `__VA_ARGS__` макрос `__VA_ARGS__` . Этот макрос позволяет использовать несколько параметров X-macro:

Пример:

```
 #define debug_print(msg, ...) printf(msg, __VA_ARGS__) \
                               printf("\nError occurred in file:line (%s:%d)\n", __FILE__,
__LINE)
```

Использование:

```
int retVal = somefunc();

debug_print("retVal of somefunc() is-> %d", retVal);
```

Этот макрос позволяет передавать несколько параметров и печатать их, но теперь он запрещает вам отправлять какие-либо параметры вообще.

```
debug_print("Hey");
```

Это вызовет некоторую синтаксическую ошибку, поскольку макрос ожидает хотя бы еще одного аргумента, а препроцессор не будет игнорировать отсутствие запятой в `debug_print()` . Также `debug_print("Hey",);` приведет к возникновению синтаксической ошибки, поскольку вы не можете оставить аргумент, переданный макросу пустым.

Чтобы решить эту проблему, был введен макрос `##__VA_ARGS__` , этот макрос утверждает, что если переменные аргументы отсутствуют, запятая удаляется препроцессором из кода.

Пример:

```
 #define debug_print(msg, ...) printf(msg, ##__VA_ARGS__) \
                               printf("\nError occured in file:line (%s:%d)\n", __FILE__,
__LINE)
```

Использование:

```
debug_print("Ret val of somefunc()?");
debug_print("%d",somefunc());
```

Прочитайте Препроцессор и макросы онлайн: https://riptutorial.com/ru/c/topic/447/препроцессор-и-макросы

глава 44: Прокладка и упаковка структуры

Вступление

По умолчанию компиляторы C выстраивают структуры так, чтобы к каждому участнику можно было получить быстрый доступ, без каких-либо штрафов за «неудовлетворенный доступ», проблему с RISC-машинами, такими как DEC Alpha и некоторые ARM-процессоры.

В зависимости от архитектуры процессора и компилятора структура может занимать больше места в памяти, чем сумма размеров ее компонентов. Компилятор может добавлять дополнения между членами или в конце структуры, но не в начале.

Упаковка отменяет заполнение по умолчанию.

замечания

У Эрика Раймонда есть статья о The Lost Art of C Structure Packing, которая является полезным чтением.

Examples

Упаковочные конструкции

По умолчанию структуры заполняются на C. Если вы хотите избежать такого поведения, вы должны явно запросить его. В GCC это `__attribute__((__packed__))` . Рассмотрим этот пример на 64-битной машине:

```
struct foo {
    char *p;  /* 8 bytes */
    char c;   /* 1 byte  */
    long x;   /* 8 bytes */
};
```

Структура будет автоматически дополнена, чтобы иметь `8-byte` выравнивание и будет выглядеть так:

```
struct foo {
    char *p;     /* 8 bytes */
    char c;      /* 1 byte  */

    char pad[7]; /* 7 bytes added by compiler */

    long x;      /* 8 bytes */
};
```

Поэтому `sizeof(struct foo)` даст нам `24` вместо `17` . Это произошло из-за 64-битного компилятора чтения / записи из / в память в 8 байтах слова на каждом шаге и очевидного при попытке написать `char c;` один байт в памяти получает 8 байтов (т.е. слово) и потребляет только первый байт, а семь его байтов остаются пустыми и недоступны для любой операции чтения и записи для заполнения структуры.

Структурная упаковка

Но если вы добавите атрибут `packed` , компилятор не добавит дополнения:

```
struct __attribute__((__packed__)) foo {
    char *p;  /* 8 bytes */
    char c;   /* 1 byte  */
    long x;   /* 8 bytes */
};
```

Теперь `sizeof(struct foo)` вернет `17` .

Используются обычно упакованные структуры:

- Чтобы сэкономить место.
- Форматировать структуру данных для передачи по сети без зависимости от каждого выравнивания архитектуры каждого узла сети.

Следует иметь в виду, что некоторые процессоры, такие как ARM Cortex-M0, не позволяют использовать неравномерный доступ к памяти; в таких случаях структура упаковки может привести к *неопределенному поведению* и может привести к сбою процессора.

Наложение структуры

Предположим, что эта `struct` определена и скомпилирована с 32-битным компилятором:

```
struct test_32 {
    int a;      // 4 byte
    short b;    // 2 byte
    int c;      // 4 byte
} str_32;
```

Мы могли бы ожидать, что эта `struct` займет всего 10 байт памяти, но, печатая `sizeof(str_32)` мы видим, что она использует 12 байтов.

Это произошло потому, что компилятор выравнивает переменные для быстрого доступа. Общий шаблон состоит в том, что когда базовый тип занимает N байтов (где N - это мощность 2, такая как 1, 2, 4, 8, 16 - и редко больше), переменная должна быть выровнена на N-байтовой границе (кратное N байтам).

Для структуры, показанной с `sizeof(int) == 4` и `sizeof(short) == 2` , общий макет:

- `int a;` хранится при смещении 0; размер 4.
- `short b;` хранится со смещением 4; размер 2.
- неназванное заполнение со смещением 6; размер 2.
- `int c;` хранится со смещением 8; размер 4.

Таким образом, `struct test_32` занимает 12 байт памяти. В этом примере нет отступающих отступов.

Компилятор обеспечит сохранение любых переменных `struct test_32` , начиная с 4-байтной границы, так что члены внутри структуры будут правильно выровнены для быстрого доступа. Функции распределения памяти, такие как `malloc()` , `calloc()` и `realloc()` , необходимы для обеспечения того, чтобы возвращаемый указатель был достаточно хорошо выровнен для использования с любым типом данных, поэтому динамически распределенные структуры также будут правильно выровнены.

Вы можете столкнуться с нечетными ситуациями, например, на 64-разрядном процессоре Intel x86_64 (например, Intel Core i7 - Mac, работающем под управлением MacOS Sierra или Mac OS X), где при компиляции в 32-разрядном режиме компиляторы размещают `double` выравнивание на 4-байтовая граница; но, на том же аппаратном обеспечении, при компиляции в 64-битном режиме компиляторы располагают `double` выравниванием по 8-байтовой границе.

Прочитайте Прокладка и упаковка структуры онлайн: https://riptutorial.com/ru/c/topic/4590/прокладка-и-упаковка-структуры

глава 45: Псевдонимы и эффективный тип

замечания

Нарушения правил псевдонимов и нарушения эффективного типа объекта - это две разные вещи, и их не следует смешивать.

- *Алиасирование* - это свойство двух указателей `a` и `b` , относящихся к одному и тому же объекту, то есть `a == b` .
- *Эффективный тип* объекта данных используется C для определения того, какие операции могут выполняться на этом объекте. В частности, эффективный тип используется для определения того, могут ли два указателя быть похожими друг на друга.

Aliasing может быть проблемой для оптимизации, так как изменение объекта через один указатель, `a` говорят, может изменить объект , который виден через другой указатель, `b` . Если ваш компилятор C должен будет предположить, что указатели всегда могут быть псевдонимом друг друга, независимо от их типа и происхождения, многие возможности оптимизации будут потеряны, а многие программы будут работать медленнее.

Правила строгого сглаживания C относятся к случаям в компиляторе, которые *могут предполагать,* какие объекты выполняют (или не делают) псевдонимы друг другу. Существуют два эмпирических правила, которые вы всегда должны иметь в виду для указателей данных.

> Если не указано иное, два указателя с одинаковым базовым типом могут быть псевдонимом.
>
> Два указателя с различным базовым типом не могут быть псевдонимом, если только один из двух типов не является типом символа.

Здесь *базовый тип* означает, что мы отбрасываем квалификацию типа, например `const` , например, если `a` является `double*` и `b` является `const double*` , компилятор *должен* обычно предполагать, что изменение `*a` может изменить `*b` .

Нарушение второго правила может привести к катастрофическим результатам. Здесь нарушение правила строгого псевдонима означает, что вы представляете компилятору два указателя `a` и `b` различного типа, которые на самом деле указывают на один и тот же объект. Тогда компилятор всегда может предположить, что эти два указывают на разные объекты и не будут обновлять его идею `*b` если вы изменили объект на `*a` .

Если вы это сделаете, поведение вашей программы станет неопределенным. Таким образом, C ставит довольно жесткие ограничения на конверсии указателей, чтобы помочь

вам избежать случайной ситуации.

> Если исходный или целевой тип `void` , все преобразования указателей между указателями с различным базовым типом должны быть *явными* .

Или, другими словами, им нужен *актерский состав* , если вы не делаете преобразования, которое просто добавляет к целевому типу квалификатор, такой как `const` .

Избегание конверсий указателей в целом и бросков, в частности, защищает вас от проблем с псевдонимом. Если вы им действительно не нужны, и эти случаи очень особенные, вы должны избегать их, как можете.

Examples

К символьным типам нельзя обращаться через несимвольные типы.

Если объект определен со статической, потоковой или автоматической продолжительностью хранения, и он имеет тип символа, либо: `char` , `unsigned char` или `signed char` , он может быть недоступен несимвольным типом. В приведенном ниже примере массив `char` переинтерпретируется как тип `int` , и поведение не определено при каждом разыменовании указателя `int b` .

```
int main( void )
{
    char a[100];
    int* b = ( int* )&a;
    *b = 1;

    static char c[100];
    b = ( int* )&c;
    *b = 2;

    _Thread_local char d[100];
    b = ( int* )&d;
    *b = 3;
}
```

Это не определено, поскольку оно нарушает правило «эффективного типа», ни один объект данных, у которого есть эффективный тип, может быть доступен через другой тип, который не является типом символа. Так как другой тип здесь `int` , это недопустимо.

Даже если размеры выравнивания и указателя будут, как известно, соответствовать, это не освобождает от этого правила, поведение все равно будет неопределенным.

Это означает, в частности, что в стандартном C нет способа зарезервировать буферный объект типа символа, который может использоваться с помощью указателей с разными типами, так как вы использовали бы буфер, который был получен `malloc` или аналогичной функцией.

Правильный способ достижения той же цели, что и в приведенном выше примере, - это использовать `union` .

```
typedef union bufType bufType;
union bufType {
   char c[sizeof(int[25])];
   int i[25];
};

int main( void )
{
    bufType a = { .c = { 0 } }; // reserve a buffer and initialize
    int* b = a.i;      // no cast necessary
    *b = 1;

    static bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 2;

    _Thread_local bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 3;
}
```

Здесь `union` гарантирует, что компилятор с самого начала знает, что к буфере можно получить доступ через разные виды. Это также имеет то преимущество, что теперь в буфере есть «вид» `ai` который уже имеет тип `int` и не требуется преобразования указателя.

Эффективный тип

Эффективным типом объекта данных является информация последнего типа, связанная с ним, если таковая имеется.

```
// a normal variable, effective type uint32_t, and this type never changes
uint32_t a = 0.0;

// effective type of *pa is uint32_t, too, simply
// because *pa is the object a
uint32_t* pa = &a;

// the object pointed to by q has no effective type, yet
void* q = malloc(sizeof uint32_t);
// the object pointed to by q still has no effective type,
// because nobody has written to it
uint32_t* qb = q;
// *qb now has effective type uint32_t because a uint32_t value was written
*qb = 37;

// the object pointed to by r has no effective type, yet, although
// it is initialized
void* r = calloc(1, sizeof uint32_t);
// the object pointed to by r still has no effective type,
// because nobody has written to or read from it
uint32_t* rc = r;
```

```
// *rc now has effective type uint32_t because a value is read
// from it with that type. The read operation is valid because we used calloc.
// Now the object pointed to by r (which is the same as *rc) has
// gained an effective type, although we didn't change its value.
uint32_t c = *rc;

// the object pointed to by s has no effective type, yet.
void* s = malloc(sizeof uint32_t);
// the object pointed to by s now has effective type uint32_t
// because an uint32_t value is copied into it.
memcpy(s, r, sizeof uint32_t);
```

Обратите внимание, что для последнего не было необходимости, чтобы у нас даже был указатель `uint32_t*` на этот объект. Достаточно того факта, что мы скопировали другой объект `uint32_t` .

Нарушение строгих правил псевдонимов

В следующем коде предположим для простоты, что `float` и `uint32_t` имеют одинаковый размер.

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    float b = *f;
    print("%g should equal %g\n", a, b);
}
```

`u` и `f` имеют различный базовый тип, и, таким образом, компилятор может предположить, что они указывают на разные объекты. Нет никакой возможности, что `*f` мог бы измениться между двумя инициализациями `a` и `b` , и поэтому компилятор может оптимизировать код для чего-то эквивалентного

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    print("%g should equal %g\n", a, a);
}
```

То есть, вторая операция загрузки `*f` может быть полностью оптимизирована.

Если мы будем называть эту функцию «нормально»,

```
 float fval = 4;
 uint32_t uval = 77;
 fun(&uval, &fval);
```

все идет хорошо, и что-то вроде

> 4 должно равняться 4

печатается. Но если мы обманываем и передаем один и тот же указатель, после его преобразования,

```
 float fval = 4;
 uint32_t* up = (uint32_t*)&fval;
 fun(up, &fval);
```

мы нарушаем строгое правило сглаживания. Тогда поведение становится неопределенным. Вывод может быть таким, как указано выше, если компилятор оптимизировал второй доступ или что-то совершенно другое, и поэтому ваша программа попадает в совершенно ненадежное состояние.

ограничить квалификацию

Если у нас есть два аргумента указателя того же типа, компилятор не может делать никаких предположений и всегда должен будет предположить, что изменение на `*e` может измениться `*f` :

```
void fun(float* e, float* f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}

float fval = 4;
float eval = 77;
 fun(&eval, &fval);
```

все идет хорошо, и что-то вроде

> составляет 4, равный 4?

печатается. Если мы передадим один и тот же указатель, программа по-прежнему будет делать правильные действия и распечатать

> составляет 4, равный 22?

Это может оказаться неэффективным, если по какой-то внешней информации мы *знаем* , что `e` и `f` никогда не укажут на один и тот же объект данных. Мы можем отражать это знание, добавляя `restrict` к параметрам указателя:

```
void fan(float*restrict e, float*restrict f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}
```

Тогда компилятор всегда может предположить, что `e` и `f` указывают на разные объекты.

Изменение байтов

Когда объект имеет эффективный тип, вы не должны пытаться его модифицировать с помощью указателя другого типа, если только этот тип не является типом `char` , `char` , `signed char` символом или символом `unsigned char` .

```
#include <inttypes.h>
#include <stdio.h>

int main(void) {
  uint32_t a = 57;
  // conversion from incompatible types needs a cast !
  unsigned char* ap = (unsigned char*)&a;
  for (size_t i = 0; i < sizeof a; ++i) {
    /* set each byte of a to 42 */
    ap[i] = 42;
  }
  printf("a now has value %" PRIu32 "\n", a);
}
```

Это действительная программа, которая печатает

> а теперь имеет значение 707406378

Это работает, потому что:

- Доступ осуществляется к отдельным байтам, видимым с типом `unsigned char` поэтому каждая модификация хорошо определена.
- Два представления объекта, через `a` и через `*ap` , псевдоним, но поскольку `ap` является указателем на тип символа, правило строгого псевдонима не применяется. Таким образом, компилятор должен предположить, что значение `a` может быть изменено в цикле `for` . Модифицированное значение `a` должно быть построено из байтов, которые были изменены.
- Тип `a` , `uint32_t` не имеет битов заполнения. Все его биты представления подсчитываются для значения, здесь `707406378` , и не может быть ловушечного представления.

Прочитайте Псевдонимы и эффективный тип онлайн: https://riptutorial.com/ru/c/topic/1301/псевдонимы-и-эффективный-тип

глава 46: Связанные списки

замечания

Язык C не определяет структуру данных связанного списка. Если вы используете C и вам нужен связанный список, вам нужно либо использовать связанный список из существующей библиотеки (например, GLib), либо написать свой собственный интерфейс связанных списков. В этом разделе приведены примеры связанных списков и двойных связанных списков, которые можно использовать в качестве отправной точки для написания собственных связанных списков.

Одиночный список

Список содержит узлы, которые состоят из одной ссылки, называемой следующей.

Структура данных

```
struct singly_node
{
  struct singly_node * next;
};
```

Двусторонний список

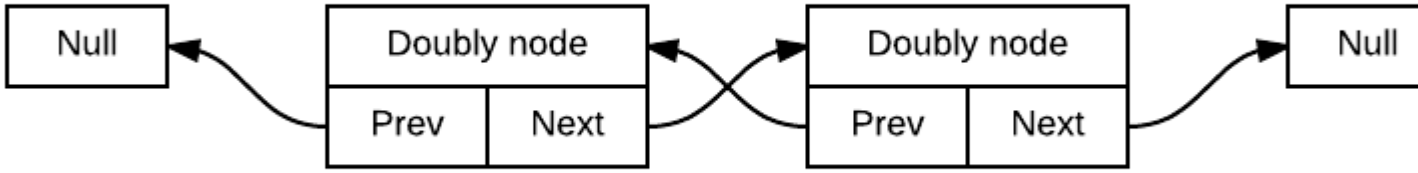
Список содержит узлы, которые состоят из двух ссылок, называемых предыдущими и последующими. Ссылки обычно ссылаются на узел с той же структурой.

Структура данных

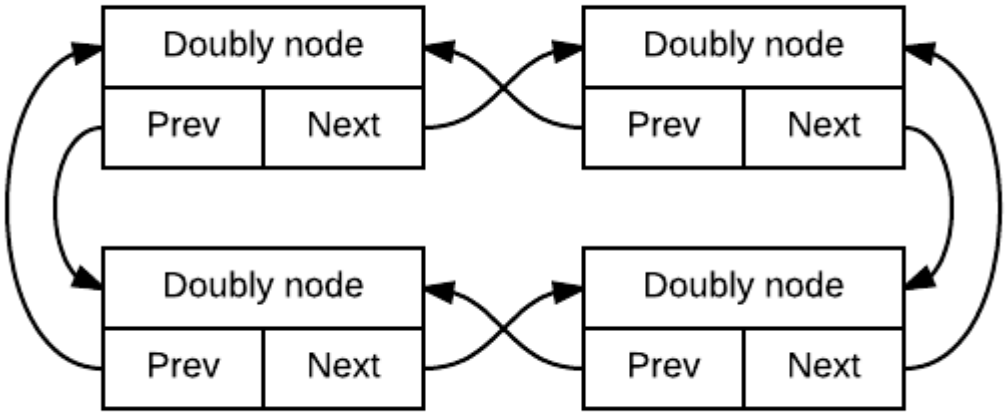
```
struct doubly_node
{
  struct doubly_node * prev;
  struct doubly_node * next;
};
```

Topoliges

Линейный или открытый



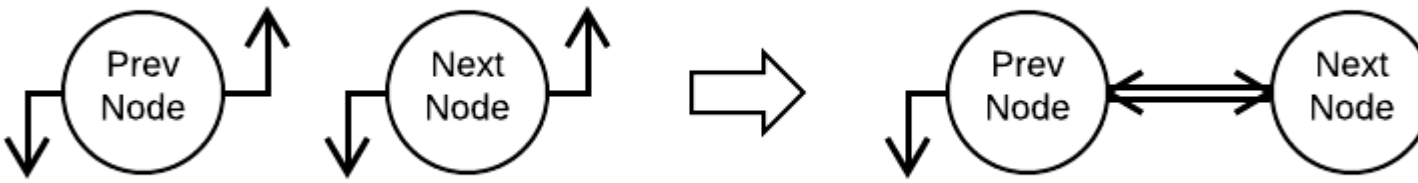
Круговое или кольцевое



процедуры

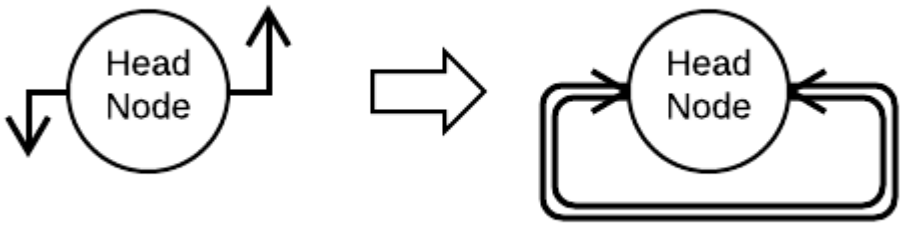
привязывать

Свяжите два узла вместе.



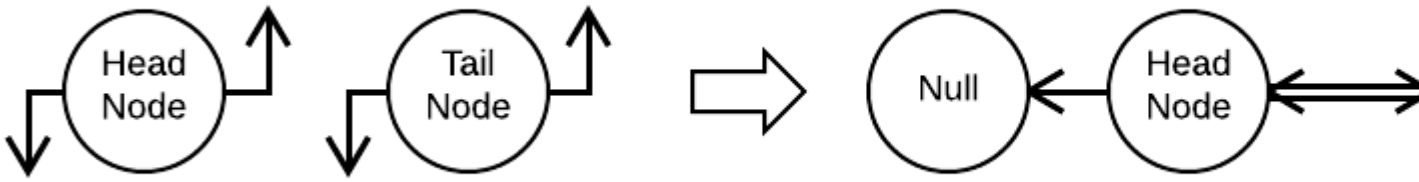

```
void doubly_node_bind (struct doubly_node * prev, struct doubly_node * next)
{
  prev->next = next;
  next->prev = prev;
}
```

Создание кругового списка




```
void doubly_node_make_empty_circularly_list (struct doubly_node * head)
{
  doubly_node_bind (head, head);
}
```

Создание линейно связанного списка




```
void doubly_node_make_empty_linear_list (struct doubly_node * head, struct doubly_node * tail)
{
  head->prev = NULL;
  tail->next = NULL;
  doubly_node_bind (head, tail);
}
```

вставка

Предположим, что пустой список всегда содержит один узел вместо NULL. Тогда процедуры ввода не должны принимать NULL.

```
void doubly_node_insert_between
(struct doubly_node * prev, struct doubly_node * next, struct doubly_node * insertion)
{
  doubly_node_bind (prev, insertion);
  doubly_node_bind (insertion, next);
}

void doubly_node_insert_before
(struct doubly_node * tail, struct doubly_node * insertion)
{
  doubly_node_insert_between (tail->prev, tail, insertion);
}

void doubly_node_insert_after
```

```
(struct doubly_node * head, struct doubly_node * insertion)
{
  doubly_node_insert_between (head, head->next, insertion);
}
```

Examples

Вставка узла в начале отдельного списка

Приведенный ниже код будет запрашивать цифры и продолжать добавлять их в начало связанного списка.

```
/* This program will demonstrate inserting a node at the beginning of a linked list */

#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};


void insert_node (struct Node **head, int nodeValue);
void print_list (struct Node *head);

int main(int argc, char *argv[]) {
  struct Node* headNode;
  headNode = NULL; /* Initialize our first node pointer to be NULL. */
  size_t listSize, i;
  do {
    printf("How many numbers would you like to input?\n");
  } while(1 != scanf("%zu", &listSize));

  for (i = 0; i < listSize; i++) {
    int numToAdd;
    do {
      printf("Enter a number:\n");
    } while (1 != scanf("%d", &numToAdd));

    insert_node (&headNode, numToAdd);
    printf("Current list after your inserted node: \n");
    print_list(headNode);
  }

  return 0;
}

void print_list (struct Node *head) {
  struct node* currentNode = head;

  /* Iterate through each link. */
  while (currentNode != NULL) {
      printf("Value: %d\n", currentNode->data);
      currentNode = currentNode -> next;
  }
}
```

```
void insert_node (struct Node **head, int nodeValue) {
  struct Node *currentNode = malloc(sizeof *currentNode);
  currentNode->data = nodeValue;
  currentNode->next = (*head);

  *head = currentNode;
}
```

Объяснение для вставки узлов

Чтобы понять, как мы добавляем узлы в начале, давайте рассмотрим возможные сценарии:

1. Список пуст, поэтому нам нужно добавить новый узел. В этом случае наша память выглядит так, где `HEAD` является указателем на первый узел:

```
| HEAD | --> NULL
```

Строка `currentNode->next = *headNode;` будет присваивать значение `currentNode->next` с `NULL` так как `headNode` изначально начинается со значения `NULL` .

Теперь мы хотим, чтобы наш указатель на главном узле указывал на наш текущий узел.

```
 -----      -------------
|HEAD | --> |CURRENTNODE| --> NULL /* The head node points to the current node */
 -----      -------------
```

Это делается с помощью `*headNode = currentNode;`

2. Список уже заполнен; нам нужно добавить новый узел в начало. Для простоты, давайте начнем с 1 узла:

```
-----    -----------
HEAD --> FIRST NODE --> NULL
-----    -----------
```

С `currentNode->next = *headNode` структура данных выглядит следующим образом:

```
---------        -----    ---------------------
currentNode -->  HEAD --> POINTER TO FIRST NODE --> NULL
---------        -----    ---------------------
```

Который, очевидно, должен быть изменен, так как `*headNode` должен указывать на `currentNode` .

```
----    -----------    ---------------
HEAD -> currentNode -->     NODE       -> NULL
----    -----------    ---------------
```

Это делается с помощью `*headNode = currentNode;`

Вставка узла в n-ое положение

До сих пор мы рассматривали вставку узла в начале отдельного списка . Однако в большинстве случаев вы захотите вставлять узлы в другое место. В приведенном ниже коде показано, как можно написать функцию `insert()` для вставки узлов в *любом* из связанных списков.

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};

struct Node* insert(struct Node* head, int value, size_t position);
void print_list (struct Node* head);

int main(int argc, char *argv[]) {
  struct Node *head = NULL; /* Initialize the list to be empty */

  /* Insert nodes at positions with values: */
  head = insert(head, 1, 0);
  head = insert(head, 100, 1);
  head = insert(head, 21, 2);
  head = insert(head, 2, 3);
  head = insert(head, 5, 4);
  head = insert(head, 42, 2);

  print_list(head);
  return 0;
}

struct Node* insert(struct Node* head, int value, size_t position) {
  size_t i = 0;
  struct Node *currentNode;

  /* Create our node */
  currentNode = malloc(sizeof *currentNode);
  /* Check for success of malloc() here! */

  /* Assign data */
  currentNode->data = value;

  /* Holds a pointer to the 'next' field that we have to link to the new node.
     By initializing it to &head we handle the case of insertion at the beginning. */
  struct Node **nextForPosition = &head;
  /* Iterate to get the 'next' field we are looking for.
     Note: Insert at the end if position is larger than current number of elements. */
  for (i = 0; i < position && *nextForPosition != NULL; i++) {
      /* nextForPosition is pointing to the 'next' field of the node.
         So *nextForPosition is a pointer to the next node.
         Update it with a pointer to the 'next' field of the next node. */
      nextForPosition = &(*nextForPosition)->next;
  }
```

```
  /* Here, we are taking the link to the next node (the one our newly inserted node should
  point to) by dereferencing nextForPosition, which points to the 'next' field of the node
  that is in the position we want to insert our node at.
  We assign this link to our next value. */
  currentNode->next = *nextForPosition;

  /* Now, we want to correct the link of the node before the position of our
  new node: it will be changed to be a pointer to our new node. */
  *nextForPosition = currentNode;

  return head;
}

void print_list (struct Node* head) {
  /* Go through the list of nodes and print out the data in each node */
  struct Node* i = head;
  while (i != NULL) {
    printf("%d\n", i->data);
    i = i->next;
  }
}
```

Смена связанного списка

Вы также можете выполнить эту задачу рекурсивно, но я выбрал в этом примере использовать итеративный подход. Эта задача полезна, если вы вставляете все свои узлы в начале связанного списка . Вот пример:

```
#include <stdio.h>
#include <stdlib.h>

#define NUM_ITEMS 10

struct Node {
  int data;
  struct Node *next;
};

void insert_node(struct Node **headNode, int nodeValue, int position);
void print_list(struct Node *headNode);
void reverse_list(struct Node **headNode);


int main(void) {
  int i;
  struct Node *head = NULL;

  for(i = 1; i <= NUM_ITEMS; i++) {
    insert_node(&head, i, i);
  }
  print_list(head);

  printf("I will now reverse the linked list\n");
  reverse_list(&head);
  print_list(head);
  return 0;
}
```

```
void print_list(struct Node *headNode) {
  struct Node *iterator;

  for(iterator = headNode; iterator != NULL; iterator = iterator->next) {
    printf("Value: %d\n", iterator->data);
  }
}

void insert_node(struct Node **headNode, int nodeValue, int position) {
  int i;
  struct Node *currentNode = (struct Node *)malloc(sizeof(struct Node));
  struct Node *nodeBeforePosition = *headNode;

  currentNode->data = nodeValue;

  if(position == 1) {
    currentNode->next = *headNode;
    *headNode = currentNode;
    return;
  }

  for (i = 0; i < position - 2; i++) {
    nodeBeforePosition = nodeBeforePosition->next;
  }

  currentNode->next = nodeBeforePosition->next;
  nodeBeforePosition->next = currentNode;
}

void reverse_list(struct Node **headNode) {
  struct Node *iterator = *headNode;
  struct Node *previousNode = NULL;
  struct Node *nextNode = NULL;

  while (iterator != NULL) {
    nextNode = iterator->next;
    iterator->next = previousNode;
    previousNode = iterator;
    iterator = nextNode;
  }

  /* Iterator will be NULL by the end, so the last node will be stored in
  previousNode.  We will set the last node to be the headNode */
  *headNode = previousNode;
}
```

Объяснение для метода обратного списка

Мы запускаем `previousNode` как `NULL` , так как мы знаем на первой итерации цикла, если мы ищем узел перед первым головным узлом, он будет `NULL` . Первый узел станет последним узлом в списке, а следующая переменная должна, естественно, быть `NULL` .

В принципе, концепция обратного связывания списка здесь заключается в том, что мы фактически реверсируем сами ссылки. Следующий член следующего узла станет узлом перед ним, например:

```
Head -> 1 -> 2 -> 3 -> 4 -> 5
```

Где каждый номер представляет собой узел. Этот список станет следующим:

```
1 <- 2 <- 3 <- 4 <- 5 <- Head
```

Наконец, голова должна указывать на 5-й узел вместо этого, и каждый узел должен указывать на предыдущий узел.

Узел 1 должен указывать на `NULL` поскольку перед ним ничего не было. Узел 2 должен указывать на узел 1, узел 3 должен указывать на узел 2 и т. Д.

Однако с этим методом существует *одна небольшая проблема* . Если мы разорвем ссылку на следующий узел и изменим ее на предыдущий узел, мы не сможем перейти к следующему узлу в списке, так как ссылка на него исчезла.

Решение этой проблемы состоит в том, чтобы просто сохранить следующий элемент в переменной (`nextNode`) перед изменением ссылки.

Список с двойной связью

Пример кода, показывающий, как узлы могут быть вставлены в двусвязный список, как список можно легко отменить и как его можно напечатать в обратном порядке.

```
#include <stdio.h>
#include <stdlib.h>

/* This data is not always stored in a structure, but it is sometimes for ease of use */
struct Node {
  /* Sometimes a key is also stored and used in the functions */
  int data;
  struct Node* next;
  struct Node* previous;
};

void insert_at_beginning(struct Node **pheadNode, int value);
void insert_at_end(struct Node **pheadNode, int value);

void print_list(struct Node *headNode);
void print_list_backwards(struct Node *headNode);

void free_list(struct Node *headNode);

int main(void) {
  /* Sometimes in a doubly linked list the last node is also stored */
  struct Node *head = NULL;

  printf("Insert a node at the beginning of the list.\n");
  insert_at_beginning(&head, 5);
  print_list(head);

  printf("Insert a node at the beginning, and then print the list backwards\n");
  insert_at_beginning(&head, 10);
```

```
  print_list_backwards(head);

  printf("Insert a node at the end, and then print the list forwards.\n");

  insert_at_end(&head, 15);
  print_list(head);

  free_list(head);

  return 0;
}

void print_list_backwards(struct Node *headNode) {
  if (NULL == headNode)
  {
    return;
  }
  /*
  Iterate through the list, and once we get to the end, iterate backwards to print
  out the items in reverse order (this is done with the pointer to the previous node).
  This can be done even more easily if a pointer to the last node is stored.
  */
  struct Node *i = headNode;
  while (i->next != NULL) {
    i = i->next; /* Move to the end of the list */
  }

  while (i != NULL) {
    printf("Value: %d\n", i->data);
    i = i->previous;
  }
}

void print_list(struct Node *headNode) {
  /* Iterate through the list and print out the data member of each node */
  struct Node *i;
  for (i = headNode; i != NULL; i = i->next) {
    printf("Value: %d\n", i->data);
  }
}

void insert_at_beginning(struct Node **pheadNode, int value) {
  struct Node *currentNode;

  if (NULL == pheadNode)
  {
    return;
  }
  /*
  This is done similarly to how we insert a node at the beginning of a singly linked
  list, instead we set the previous member of the structure as well
  */
  currentNode = malloc(sizeof *currentNode);

  currentNode->next = NULL;
  currentNode->previous = NULL;
  currentNode->data = value;

  if (*pheadNode == NULL) { /* The list is empty */
    *pheadNode = currentNode;
    return;
```

```
  }

  currentNode->next = *pheadNode;
  (*pheadNode)->previous = currentNode;
  *pheadNode = currentNode;
}

void insert_at_end(struct Node **pheadNode, int value) {
  struct Node *currentNode;

  if (NULL == pheadNode)
  {
    return;
  }

  /*
  This can, again be done easily by being able to have the previous element.  It
  would also be even more useful to have a pointer to the last node, which is commonly
  used.
  */

  currentNode = malloc(sizeof *currentNode);
  struct Node *i = *pheadNode;

  currentNode->data = value;
  currentNode->next = NULL;
  currentNode->previous = NULL;

  if (*pheadNode == NULL) {
    *pheadNode = currentNode;
    return;
  }

  while (i->next != NULL) { /* Go to the end of the list */
    i = i->next;
  }

  i->next = currentNode;
  currentNode->previous = i;
}

void free_list(struct Node *node) {
  while (node != NULL) {
    struct Node *next = node->next;
    free(node);
    node = next;
  }
}
```

Обратите внимание, что иногда полезно хранить указатель на последний узел (более эффективно просто прыгать прямо до конца списка, чем нужно перебирать до конца):

```
struct Node *lastNode = NULL;
```

В этом случае необходимо обновить его при внесении изменений в список.

Иногда ключ используется для идентификации элементов. Он просто является членом структуры узла:

```
struct Node {
  int data;
  int key;
  struct Node* next;
  struct Node* previous;
};
```

Затем ключ используется, когда любые задачи выполняются над конкретным элементом, например удалением элементов.

Прочитайте Связанные списки онлайн: https://riptutorial.com/ru/c/topic/560/связанные-списки

глава 47: Создание и включение файлов заголовков

Вступление

В современных C заголовочные файлы являются важными инструментами, которые необходимо разрабатывать и использовать правильно. Они позволяют компилятору перекрестно проверять самостоятельно скомпилированные части программы.

Заголовки объявляют типы, функции, макросы и т. Д., Которые нужны потребителям набора средств. Весь код, который использует любой из этих объектов, включает заголовок. Весь код, определяющий эти объекты, включает заголовок. Это позволяет компилятору проверить соответствие совпадений и определений.

Examples

Вступление

При создании и использовании файлов заголовков в проекте C существует ряд рекомендаций:

- Idemopotence

  Если заголовочный файл несколько раз включается в блок трансляции (TU), он не должен нарушать сборки.

- Автономность

  Если вам нужны средства, объявленные в файле заголовка, вам не нужно включать какие-либо другие заголовки явно.

- Минимальность

  Вы не должны удалять информацию из заголовка, не вызывая сбоев сборки.

- Включить то, что вы используете (IWYU)

  Больше заботы о C ++, чем C, но тем не менее важны и для C. Если код в TU (вызовите его `code.c`) напрямую использует функции, объявленные заголовком (назовите его `"headerA.h"`), тогда `code.c` должен `#include "headerA.h"` напрямую, даже если TU включает другой заголовок (назовите его `"headerB.h"`), который в настоящий момент происходит с включением `"headerA.h"` .

Иногда бывает достаточно оснований для нарушения одного или нескольких из этих рекомендаций, но вы должны знать, что нарушаете правило и осознаете последствия этого, прежде чем нарушить его.

идемпотентность

Если конкретный заголовочный файл включен более одного раза в блок трансляции (TU), не должно быть проблем с компиляцией. Это называется «идемпотенция»; ваши заголовки должны быть идемпотентными. Подумайте, насколько сложной была бы жизнь, если бы вы должны были убедиться, что `#include <stdio.h>` был включен только один раз.

Существует два способа достижения идемпотенциала: защита заголовков и директива `#pragma once` .

Защитные кожухи

Защитные решетки просты и надежны и соответствуют стандарту C. Первые строки без комментариев в файле заголовка должны иметь следующий вид:

```
#ifndef UNIQUE_ID_FOR_HEADER
#define UNIQUE_ID_FOR_HEADER
```

Последняя строка без комментария должна быть `#endif` , необязательно с комментарием после нее:

```
#endif /* UNIQUE_ID_FOR_HEADER */
```

Весь операционный код, включая другие директивы `#include` , должен находиться между этими строками.

Каждое имя должно быть уникальным. Часто используется схема имен, такая как `HEADER_H_INCLUDED` . В некотором более старом коде используется символ, обозначенный как защита заголовка (например, `#ifndef BUFSIZ` в `<stdio.h>`), но он не так надежен, как уникальное имя.

Один из вариантов заключается в использовании сгенерированного хеша MD5 (или другого) для имени защитника заголовка. Вам следует избегать эмуляции схем, используемых заголовками системы, которые часто используют имена, зарезервированные для имен реализации, начиная с символа подчеркивания, за которым следует либо другое подчеркивание, либо буква верхнего регистра.

Директива `#pragma once`

В качестве альтернативы, некоторые компиляторы поддерживают директиву `#pragma once`

которая имеет тот же эффект, что и три строки, показанные для защиты заголовков.

```
#pragma once
```

Компиляторы, которые поддерживают `#pragma once` включают MS Visual Studio и GCC и Clang. Однако, если переносимость вызывает беспокойство, лучше использовать защиту заголовков или использовать оба варианта. Современные компиляторы (поддерживающие C89 или более поздние) должны игнорировать, без комментариев, прагмы, которые они не распознают («Любая такая прагма, которая не распознается реализацией, игнорируется»), но старые версии GCC не были настолько снисходительными.

Автономность

Современные заголовки должны быть автономными, а это значит, что программа, которая должна использовать средства, определенные `header.h` может включать этот заголовок (`#include "header.h"`) и не беспокоиться о том, нужно ли сначала включать другие заголовки.

Рекомендация: Файлы заголовков должны быть автономными.

Исторические правила

Исторически сложилось так, что это был спорный спорный вопрос.

Однажды на другое тысячелетие, AT & T Indian Hill C Стиль и стандарты кодирования заявили:

> Файлы заголовков не должны быть вложенными. Поэтому пролог для файла заголовка должен описывать, какие другие заголовки должны быть `#include` d для того, чтобы заголовок был функциональным. В крайних случаях, когда большое количество файлов заголовков должно быть включено в несколько разных исходных файлов, допустимо поместить все обычные `#include` s в один файл include.

Это противоположность самоограничения.

Современные правила

Однако с тех пор мнение имеет тенденцию в противоположном направлении. Если исходный файл должен использовать средства, объявленные заголовком `header.h` , программист должен иметь возможность писать:

```
#include "header.h"
```

и (при условии наличия правильных путей поиска, установленных в командной строке), любые необходимые предварительные заголовки будут включены `header.h` без необходимости добавления дополнительных заголовков в исходный файл.

Это обеспечивает лучшую модульность исходного кода. Он также защищает источник от «догадки, почему этот заголовок был добавлен», головоломка, которая возникает после того, как код был изменен и взломан на десятилетие или два.

Стандарты кодирования космического полета NASA Goddard (GSFC) для C - один из самых современных стандартов, но сейчас его трудно отследить. В нем говорится, что заголовки должны быть автономными. Он также обеспечивает простой способ гарантировать, что заголовки являются автономными: файл реализации для заголовка должен включать заголовок в качестве первого заголовка. Если он не является самодостаточным, этот код не будет компилироваться.

Обоснование, данное GSFC, включает:

> §2.1.1 Заголовок включает в себя обоснование
>
> Этот стандарт требует, чтобы заголовок блока содержал инструкции `#include` для всех других заголовков, требуемых заголовком блока. Размещение `#include` для заголовка блока сначала в блоке устройства позволяет компилятору проверить, содержит ли заголовок все необходимые инструкции `#include` .
>
> Альтернативный дизайн, не разрешенный этим стандартом, не допускает операторов `#include` в заголовках; все `#includes` производятся в файлах body. Заголовочные файлы блока должны содержать инструкции #ifdef, которые проверяют, что требуемые заголовки включены в правильный порядок.
>
> Одно из преимуществ альтернативного дизайна заключается в том, что список `#include` в основном файле - это список зависимостей, необходимый в make-файле, и этот список проверяется компилятором. При стандартном дизайне инструмент должен использоваться для создания списка зависимостей. Тем не менее, все рекомендованные отраслью среды разработки предоставляют такой инструмент.
>
> Основным недостатком альтернативного дизайна является то, что если список требуемых заголовков блока изменяется, каждый файл, который использует этот модуль, должен быть отредактирован, чтобы обновить список операторов `#include` . Кроме того, требуемый список заголовков для библиотеки библиотеки компилятора может отличаться для разных целей.
>
> Другим недостатком альтернативного дизайна является то, что файлы заголовков библиотеки компилятора и другие файлы сторонних разработчиков должны быть изменены для добавления необходимых операторов `#ifdef` .

Таким образом, самоограничение означает, что:

- Если заголовку `header.h` нужен новый вложенный заголовок `extra.h` , вам не нужно проверять каждый исходный файл, который использует `header.h` чтобы узнать, нужно ли вам добавлять `extra.h` .
- Если заголовку `header.h` больше не нужно включать определенный заголовок `notneeded.h` , вам не нужно проверять каждый исходный файл, который использует `header.h` чтобы увидеть, можете ли вы безопасно удалить `notneeded.h` (но см. `notneeded.h` Включить то, что вы используете» .
- Вам не нужно устанавливать правильную последовательность для включения необходимых заголовков (что требует топологической сортировки для правильной работы).

Проверка самоограничения

См. Ссылку на статическую библиотеку для скрипта `chkhdr` который может использоваться для проверки идемпотенции и самоограничения файла заголовка.

Минимальность

Заголовки - это решающий механизм проверки согласованности, но они должны быть как можно меньше. В частности, это означает, что заголовок не должен включать другие заголовки только потому, что для файла реализации понадобятся другие заголовки. Заголовок должен содержать только те заголовки, которые необходимы для потребителя описанных услуг.

Например, заголовок проекта не должен включать `<stdio.h>` если только один из функциональных интерфейсов не использует тип `FILE *` (или один из других типов, определенных исключительно в `<stdio.h>`). Если интерфейс использует `size_t` , наименьший заголовок, который достаточно, `<stddef.h>` . Очевидно, что если включен другой заголовок, который определяет `size_t` , нет необходимости включать `<stddef.h>` .

Если заголовки минимальны, то это также приводит к минимуму времени компиляции.

Можно создать заголовки, единственной целью которых является включение множества других заголовков. Они редко оказываются хорошей идеей в долгосрочной перспективе, потому что для нескольких исходных файлов понадобятся все средства, описанные всеми заголовками. Например, можно было бы разработать `<standard-ch>` , который включает все стандартные заголовки C - с осторожностью, поскольку некоторые заголовки не всегда присутствуют. Однако очень немногие программы фактически используют средства `<locale.h>` или `<tgmath.h>` .

- См. Также Как связать несколько файлов реализации в C?

Включить то, что вы используете (IWYU)

Проект Google Include What You Use , или IWYU, гарантирует, что исходные файлы включают все заголовки, используемые в коде.

Предположим, что исходный файл `source.c` содержит заголовок `arbitrary.h` который, в свою очередь, случайно включает `freeloader.h` , но исходный файл также явно и независимо использует средства из `freeloader.h` . Все хорошо начать. Затем в один прекрасный день `arbitrary.h` изменяется, поэтому его клиентам больше не нужны средства `freeloader.h` . Внезапно `source.c` прекращает компиляцию - поскольку он не соответствует критериям IWYU. Поскольку код в `source.c` явно использовал возможности `freeloader.h` , он должен был включить то, что он использует - в источнике также должен был быть явно `#include "freeloader.h"` . (Идемпотентность гарантировала бы, что проблем не было.)

Философия IWYU максимизирует вероятность того, что код продолжает компилироваться даже при разумных изменениях, внесенных в интерфейсы. Очевидно, что если ваш код вызывает функцию, которая впоследствии удаляется из опубликованного интерфейса, никакая подготовка не может препятствовать изменениям. Вот почему, когда это возможно, избегаются изменения API-интерфейсов и почему существуют циклы отказов по нескольким выпускам и т. Д.

Это особая проблема в C ++, потому что стандартным заголовкам разрешено включать друг друга. Исходный файл `file.cpp` может включать один заголовок `header1.h` который на одной платформе включает другой заголовок `header2.h` . `file.cpp` может использовать средства `header2.h` . Первоначально это не было проблемой - код будет компилироваться, поскольку `header1.h` включает `header2.h` . На другой платформе или обновлении текущей платформы `header1.h` можно было бы пересмотреть, чтобы она больше не включала `header2.h` , а затем `file.cpp` прекратил компиляцию в результате.

IWYU обнаружит проблему и порекомендует, что `header2.h` будет включен непосредственно в `file.cpp` . Это обеспечило бы ее компиляцию. Аналогичные соображения также относятся к коду C.

Обозначение и сборник

В стандарте C говорится, что между обозначениями `#include <header.h>` и `#include "header.h"` очень мало различий.

> [`#include <header.h>`] ищет последовательность определённых реализацией мест для заголовка, идентифицированного однозначно указанной последовательностью между разделителями `<` и `>` и вызывает замену этой директивы на все содержимое заголовка. Как указано места, или идентифицированный заголовок определяется реализацией.
>
> [`#include "header.h"`] вызывает замену этой директивы всем содержимым исходного файла, идентифицированного указанной последовательностью между разделителями `"…"` . Именованный исходный файл выполняется

поисковым способом. Если этот поиск не поддерживается или если поиск не выполняется, директива перерабатывается, как если бы она читала [`#include <header.h>`] ...

Таким образом, двойная кавычка может выглядеть в большем количестве мест, чем форма с угловыми скобками. Стандарт указывает на пример, что стандартные заголовки должны быть включены в угловые скобки, даже если компиляция работает, если вместо этого использовать двойные кавычки. Аналогично, такие стандарты, как POSIX, используют формат с угловым скобкой - и вам тоже нужно. Зарезервировать заголовки с двойными кавычками для заголовков, определенных проектом. Для внешних заголовков (включая заголовки других проектов, на которые опирается ваш проект), наиболее подходящим является обозначение углового кронштейна.

Обратите внимание, что между `#include` и заголовком должно быть пробел, хотя компиляторы не будут там места. Пробелы дешевы.

В ряде проектов используются такие обозначения, как:

```
#include <openssl/ssl.h>
#include <sys/stat.h>
#include <linux/kernel.h>
```

Вы должны подумать, следует ли использовать этот контроль пространства имен в вашем проекте (это, скорее всего, хорошая идея). Вы должны избегать имен, используемых существующими проектами (в частности, как `sys` и `linux` будут плохими выборами).

Если вы используете это, ваш код должен быть осторожным и последовательным в использовании обозначений.

Не используйте обозначения `#include "../include/header.h"` .

Заголовочные файлы должны редко задавать переменные. Хотя вы сохраните глобальные переменные до минимума, если вам нужна глобальная переменная, вы объявите ее в заголовке и определите ее в одном подходящем исходном файле, и этот исходный файл будет включать заголовок для перекрестной проверки декларации и определения , и все исходные файлы, которые используют переменную, будут использовать заголовок, чтобы объявить его.

Следствие: вы не будете объявлять глобальные переменные в исходном файле - исходный файл будет содержать только определения.

Заголовочные файлы должны редко объявлять `static` функции с заметным исключением `static inline` функций, которые будут определены в заголовках, если эта функция необходима в нескольких исходных файлах.

- Исходные файлы определяют глобальные переменные и глобальные функции.

- Исходные файлы не объявляют о существовании глобальных переменных или функций; они включают заголовок, который объявляет переменную или функцию.
- Заголовочные файлы объявляют глобальную переменную и функции (и типы и другие вспомогательные материалы).
- Заголовочные файлы не определяют переменные или любые функции, кроме (`static`) `inline` функций.

Перекрестные ссылки

- Где записывать функции в C?
- Список стандартных файлов заголовков в C и C ++
- Является ли `inline` не `static` или `extern` когда-либо полезным в C99?
- Как использовать `extern` для обмена переменными между исходными файлами?
- Каковы преимущества относительного пути, например `"../include/header.h"` для заголовка?
- Оптимизация включения заголовка
- Должен ли я включать каждый заголовок?

Прочитайте Создание и включение файлов заголовков онлайн:
https://riptutorial.com/ru/c/topic/6257/создание-и-включение-файлов-заголовков

глава 48: Составные литералы

Синтаксис

- (тип) {initializer-list}

замечания

Стандарт C говорит в C11-§6.5.2.5 / 3:

> Постфиксное выражение, состоящее из имени типа в скобках, за которым следует скопированный список инициализаторов, является *сложным литералом* . Он предоставляет неназванный объект, значение которого задается в списке инициализаторов. [99)]

и сноска 99 гласит:

> Обратите внимание, что это отличается от литого выражения. Например, приведение задает преобразование только в скалярные типы или только **void** , а результат литого выражения не является значением lvalue.

Обратите внимание, что:

> Строковые литералы и составные литералы с категориальными типами, не обязательно обозначают разные объекты. [101)]
>
> 101). Это позволяет реализациям совместно использовать хранилище для строковых литералов и постоянных составных литералов с одинаковыми или перекрывающимися представлениями.

Пример приведен в стандарте:
C11-§6.5.2.5 / 13:

> Подобно строковым литералам, const-квалифицированные составные литералы могут быть помещены в постоянную память и могут быть разделены. Например,
>
> ```
> (const char []){"abc"} == "abc"
> ```
>
> может дать 1, если хранилище литералов разделено.

Examples

Определение / Инициализация сложных литералов

Комбинированный литерал является неназванным объектом, который создается в области,

где определено. Концепция была впервые введена в стандарте C99. Примером для составного литерала является

Примеры из стандарта C, C11-§6.5.2.5 / 9:

```
int *p = (int [2]){ 2, 4 };
```

`p` инициализируется адресом первого элемента неназванного массива из двух целых чисел.

Смежный литерал - это значение lvalue. Длительность хранения неназванного объекта является либо статической (если литерал появляется в области файла), либо автоматически (если литерал появляется в области блока), а в последнем случае время жизни объекта заканчивается, когда элемент управления выходит из закрывающего блока.

```
void f(void)
{
    int *p;
    /*...*/
    p = (int [2]){ *p };
    /*...*/
}
```

> `p` присваивается адрес первого элемента массива из двух ints, первый из которых имеет значение, ранее указанное `p` а второе - ноль. [...]

Здесь `p` остается действительным до конца блока.

Комбинированный литерал с указателями

(также от C11)

```
 struct point {
   unsigned x;
   unsigned y;
 };

 extern void drawline(struct point, struct point);

// used somewhere like this
drawline((struct point){.x=1, .y=1}, (struct point){.x=3, .y=4});
```

`drawline` функция `drawline` получает два аргумента типа `struct point` . Первый имеет координаты `x == 1` и `y == 1` , тогда как второй имеет `x == 3` и `y == 4`

Составной литерал без указания длины массива

```
int *p = (int []){ 1, 2, 3};
```

В этом случае размер массива не указан, тогда он будет определяться длиной инициализатора.

Составной литерал, имеющий длину инициализатора меньше указанного размера массива

```
int *p = (int [10]){1, 2, 3};
```

остальные элементы составного литерала будут инициализированы до `0` неявно.

Только для чтения составной литерал

Обратите внимание, что составной литерал является lvalue, и поэтому его элементы могут быть модифицируемыми. Литерал соединения *только для чтения* может быть указан с использованием спецификатора `const` as `(const int[]){1,2}` .

Комбинированный литерал, содержащий произвольные выражения

Внутри функции составной литерал, как и для любой инициализации с C99, может иметь произвольные выражения.

```
void foo()
{
    int *p;
    int i = 2; j = 5;
    /*...*/
    p = (int [2]){ i+j, i*j };
    /*...*/
}
```

Прочитайте Составные литералы онлайн: https://riptutorial.com/ru/c/topic/4135/составные-литералы

глава 49: Союзы

Examples

Разница между структурой и объединением

Это иллюстрирует, что члены профсоюза разделяют память и члены структуры не обмениваются памятью.

```
#include <stdio.h>
#include <string.h>

union My_Union
{
  int variable_1;
  int variable_2;
};

struct My_Struct
{
  int variable_1;
  int variable_2;
};

int main (void)
{
  union My_Union u;
  struct My_Struct s;
  u.variable_1 = 1;
  u.variable_2 = 2;
  s.variable_1 = 1;
  s.variable_2 = 2;
  printf ("u.variable_1: %i\n", u.variable_1);
  printf ("u.variable_2: %i\n", u.variable_2);
  printf ("s.variable_1: %i\n", s.variable_1);
  printf ("s.variable_2: %i\n", s.variable_2);
  printf ("sizeof (union My_Union): %i\n", sizeof (union My_Union));
  printf ("sizeof (struct My_Struct): %i\n", sizeof (struct My_Struct));
  return 0;
}
```

Использование союзов для повторной интерпретации значений

Некоторые реализации C позволяют коду писать одному члену типа объединения, а затем читать от другого, чтобы выполнить своего рода реинтерпретацию (разбор нового типа в виде битового представления старого).

Однако важно отметить, что это не разрешено стандартным током C или прошлым и приведет к неопределенному поведению, тем не менее, это очень распространенное расширение, предлагаемое компиляторами (так что проверьте свои документы компилятора, если вы планируете это делать) ,

Одним из реальных примеров этого метода является алгоритм «Быстрый обратный квадратный корень», который опирается на детали реализации чисел с плавающей запятой IEEE 754 для выполнения обратного квадратного корня быстрее, чем при использовании операций с плавающей запятой, этот алгоритм может выполняться либо путем кастования указателя (что очень опасно и нарушает правило строгого псевдонима) или через объединение (которое по-прежнему является неопределенным поведением, но работает во многих компиляторах):

```
union floatToInt
{
    int32_t intMember;
    float floatMember; /* Float must be 32 bits IEEE 754 for this to work */
};

float inverseSquareRoot(float input)
{
    union floatToInt x;
    int32_t i;
    float f;
    x.floatMember = input;     /* Assign to the float member */
    i = x.intMember;           /* Read back from the integer member */
    i = 0x5f3759df - (i >> 1);
    x.intMember = i;           /* Assign to the integer member */
    f = x.floatMember;         /* Read back from the float member */
    f = f * (1.5f - input * 0.5f * f * f);
    return f * (1.5f - input * 0.5f * f * f);
}
```

Этот метод широко использовался в компьютерной графике и играх в прошлом из-за его большей скорости по сравнению с использованием операций с плавающей запятой и является очень компромиссным, теряя определенную точность и очень не переносимый в обмен на скорость.

Запись одному члену профсоюза и чтение от другого

Члены профсоюза имеют одинаковое пространство в памяти. Это означает, что запись в один член перезаписывает данные во всех других членах и что чтение из одного члена приводит к тем же данным, что и чтение от всех других членов. Однако, поскольку члены профсоюза могут иметь разные типы и размеры, данные, которые читаются, могут интерпретироваться по-разному, см. Http://www.riptutorial.com/c/example/9399/using-unions-to-reinterpret-values

Простой пример ниже демонстрирует объединение с двумя членами, одинаковыми типами. Он показывает, что запись в член `m_1` приводит к тому, что записанное значение считывается из члена `m_2` а запись в член `m_2` приводит к тому, что записанное значение считывается из члена `m_1` .

```
#include <stdio.h>

union my_union /* Define union */
```

```
{
    int m_1;
    int m_2;
};

int main (void)
{
    union my_union u;              /* Declare union */
    u.m_1 = 1;                     /* Write to m_1 */
    printf("u.m_2: %i\n", u.m_2); /* Read from m_2 */
    u.m_2 = 2;                     /* Write to m_2 */
    printf("u.m_1: %i\n", u.m_1); /* Read from m_1 */
    return 0;
}
```

Результат

```
u.m_2: 1
u.m_1: 2
```

Прочитайте Союзы онлайн: https://riptutorial.com/ru/c/topic/7645/союзы

глава 50: Стандартная математика

Синтаксис

- #include <math.h>
- double pow (double x, double y);
- float powf (float x, float y);
- long double powl (long double x, long double y);

замечания

1. Для связи с библиотекой math используйте `-lm` с gcc-флагами.
2. Портативная программа, которая должна проверять ошибку от математической функции, должна установить `errno` в ноль и сделать следующий вызов `feclearexcept(FE_ALL_EXCEPT);` перед вызовом математической функции. По возвращении из математической функции, если `errno` является ненулевым, или следующий вызов возвращает ненулевое значение `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW);` то в математической функции произошла ошибка. Прочтите manpage math_error для получения дополнительной информации.

Examples

Двойной предел с плавающей запятой: fmod ()

Эта функция возвращает остаток с плавающей запятой деления `x/y` . Возвращаемое значение имеет тот же знак, что и x.

```
#include <math.h> /* for fmod() */
#include <stdio.h> /* for printf() */

int main(void)
{
    double x = 10.0;
    double y = 5.1;

    double modulus = fmod(x, y);

    printf("%lf\n", modulus); /* f is the same as lf. */

    return 0;
}
```

Выход:

```
4.90000
```

Важно: используйте эту функцию с осторожностью, так как она может возвращать неожиданные значения из-за работы значений с плавающей запятой.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    printf("%f\n", fmod(1, 0.1));
    printf("%19.17f\n", fmod(1, 0.1));
    return 0;
}
```

Выход:

```
0.1
0.09999999999999995
```

Одиночная и длинная двойные точки с плавающей запятой: fmodf (), fmodl ()

C99

Эти функции возвращают остаток с плавающей запятой деления `x/y` . Возвращаемое значение имеет тот же знак, что и x.

Единая точность:

```
#include <math.h> /* for fmodf() */
#include <stdio.h> /* for printf() */

int main(void)
{
    float x = 10.0;
    float y = 5.1;

    float modulus = fmodf(x, y);

    printf("%f\n", modulus); /* lf would do as well as modulus gets promoted to double. */
}
```

Выход:

```
4.90000
```

Двойная двойная точность:

```
#include <math.h> /* for fmodl() */
#include <stdio.h> /* for printf() */

int main(void)
{
```

```
    long double x = 10.0;
    long double y = 5.1;

    long double modulus = fmodl(x, y);

    printf("%Lf\n", modulus); /* Lf is for long double. */
}
```

Выход:

```
4.90000
```

Функции мощности - pow (), powf (), powl ()

Следующий примерный код вычисляет сумму рядов *1 + 4 (3 + 3 ^ 2 + 3 ^ 3 + 3 ^ 4 + ... + 3 ^ N)*, используя семейство стандартной математической библиотеки pow ().

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <fenv.h>

int main()
{
        double pwr, sum=0;
        int i, n;

        printf("\n1+4(3+3^2+3^3+3^4+...+3^N)=?\nEnter N:");
        scanf("%d",&n);
        if (n<=0) {
                printf("Invalid power N=%d", n);
                return -1;
        }

        for (i=0; i<n+1; i++) {
                errno = 0;
                feclearexcept(FE_ALL_EXCEPT);
                pwr = powl(3,i);
                if (fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW |
                        FE_UNDERFLOW)) {
                        perror("Math Error");
                }
                sum += i ? pwr : 0;
                printf("N= %d\tS= %g\n", i, 1+4*sum);
        }

        return 0;
}
```

Пример:

```
1+4(3+3^2+3^3+3^4+...+3^N)=?
Enter N:10
N= 0    S= 1
N= 1    S= 13
```

```
N= 2    S= 49
N= 3    S= 157
N= 4    S= 481
N= 5    S= 1453
N= 6    S= 4369
N= 7    S= 13117
N= 8    S= 39361
N= 9    S= 118093
N= 10    S= 354289
```

Прочитайте Стандартная математика онлайн: https://riptutorial.com/ru/c/topic/3170/стандартная-математика

глава 51: Структуры

Вступление

Структуры обеспечивают способ группировать набор связанных переменных различных типов в единую единицу памяти. К структуре в целом можно отнести одно имя или указатель; к элементам структуры также можно обращаться индивидуально. Структуры могут быть переданы в функции и возвращены из функций. Они определяются с помощью ключевого слова `struct` .

Examples

Простые структуры данных

Структурные типы данных полезны для упаковки связанных данных и ведут себя как одна переменная.

Объявление простой `struct` , содержащей два члена `int` :

```
struct point
{
    int x;
    int y;
};
```

`x` и `y` называются *членами* (или *полями*) `point` структуры.

Определение и использование структур:

```
struct point p;    // declare p as a point struct
p.x = 5;           // assign p member variables
p.y = 3;
```

Структуры могут быть инициализированы при определении. Вышеупомянутое эквивалентно:

```
struct point p = {5, 3};
```

Структуры также могут быть инициализированы с использованием назначенных инициализаторов .

Доступ к полям также выполняется с помощью . оператор

```
printf("point is (x = %d, y = %d)", p.x, p.y);
```

Typedef Structs

Объединение `typedef` со `struct` может сделать код более понятным. Например:

```
typedef struct
{
    int x, y;
} Point;
```

в отличие от:

```
struct Point
{
    int x, y;
};
```

может быть объявлено как:

```
Point point;
```

вместо:

```
struct Point point;
```

Еще лучше использовать следующие

```
typedef struct Point Point;

struct Point
{
    int x, y;
};
```

иметь преимущество обоих возможных определений `point` . Такое объявление наиболее удобно, если вы сначала изучили C ++, где вы можете опустить ключевое слово `struct` если имя не является двусмысленным.

`typedef` имена для структур могут быть в конфликте с другими идентификаторами других частей программы. Некоторые считают это недостатком, но для большинства людей, имеющих `struct` и другой идентификатор, это очень тревожно. Печально известный, например, POSIX ' `stat`

```
int stat(const char *pathname, struct stat *buf);
```

где вы видите функцию `stat` которой есть один аргумент, который является `struct stat` .

`typedef` 'd structs без имени тега всегда налагают, что все объявление `struct` видимо для кода, который его использует. Затем все объявление `struct` должно быть помещено в файл

заголовка.

Рассматривать:

```
#include "bar.h"

struct foo
{
    bar *aBar;
};
```

Таким образом, с помощью `typedef` d `struct` которая не имеет имени тега, файл `bar.h` всегда должен включать в себя все определение `bar` . Если мы используем

```
typedef struct bar bar;
```

в `bar.h` , детали структуры `bar` могут быть скрыты.

Посмотреть Typedef

Указатели на структуры

Когда у вас есть переменная, содержащая `struct` , вы можете получить доступ к своим полям, используя оператор точки (`.`). Однако, если у вас есть указатель на `struct` , это не сработает. Вы должны использовать оператор стрелки (`->`) для доступа к своим полям. Вот пример ужасно простой (некоторые могут сказать «ужасной и простой») реализации стека, который использует указатели на `struct` s и демонстрирует оператор стрелки.

```
#include <stdlib.h>
#include <stdio.h>

/* structs */
struct stack
{
    struct node *top;
    int size;
};

struct node
{
    int data;
    struct node *next;
};

/* function declarations */
int push(int, struct stack*);
int pop(struct stack*);
void destroy(struct stack*);

int main(void)
{
    int result = EXIT_SUCCESS;
```

```
    size_t i;

    /* allocate memory for a struct stack and record its pointer */
    struct stack *stack = malloc(sizeof *stack);
    if (NULL == stack)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }

    /* initialize stack */
    stack->top = NULL;
    stack->size = 0;

    /* push 10 ints */
    {
        int data = 0;
        for(i = 0; i < 10; i++)
        {
            printf("Pushing: %d\n", data);
            if (-1 == push(data, stack))
            {
                perror("push() failed");
                result = EXIT_FAILURE;
                break;
            }

            ++data;
        }
    }

    if (EXIT_SUCCESS == result)
    {
        /* pop 5 ints */
        for(i = 0; i < 5; i++)
        {
            printf("Popped: %i\n", pop(stack));
        }
    }

    /* destroy stack */
    destroy(stack);

    return result;
}

/* Push a value onto the stack. */
/* Returns 0 on success and -1 on failure. */
int push(int data, struct stack *stack)
{
    int result = 0;

    /* allocate memory for new node */
    struct node *new_node = malloc(sizeof *new_node);
    if (NULL == new_node)
    {
        result = -1;
    }
    else
    {
        new_node->data = data;
```

```
        new_node->next = stack->top;
        stack->top = new_node;
        stack->size++;
    }

    return result;
}

/* Pop a value off of the stack. */
/* Returns the value popped off the stack */
int pop(struct stack *stack)
{
    struct node *top = stack->top;
    int data = top->data;
    stack->top = top->next;
    stack->size--;
    free(top);
    return data;
}

/* destroy the stack */
void destroy(struct stack *stack)
{
    /* free all pointers */
    while(stack->top != NULL)
    {
        pop(stack);
    }
}
```

Гибкие члены массива

C99

Объявление типа

Структура *с по меньшей мере одним элементом* может дополнительно содержать один элемент массива неопределенной длины в конце структуры. Это называется гибким элементом массива:

```
struct ex1
{
    size_t foo;
    int flex[];
};

struct ex2_header
{
    int foo;
    char bar;
};

struct ex2
{
    struct ex2_header hdr;
    int flex[];
```

```
};

/* Merged ex2_header and ex2 structures. */
struct ex3
{
    int foo;
    char bar;
    int flex[];
};
```

Эффекты на размер и заполнение

Элемент гибкого массива рассматривается как не имеющий размера при вычислении размера структуры, хотя дополнение между этим элементом и предыдущим элементом структуры может все еще существовать:

```
/* Prints "8,8" on my machine, so there is no padding. */
printf("%zu,%zu\n", sizeof(size_t), sizeof(struct ex1));

/* Also prints "8,8" on my machine, so there is no padding in the ex2 structure itself. */
printf("%zu,%zu\n", sizeof(struct ex2_header), sizeof(struct ex2));

/* Prints "5,8" on my machine, so there are 3 bytes of padding. */
printf("%zu,%zu\n", sizeof(int) + sizeof(char), sizeof(struct ex3));
```

Элемент гибкого массива считается неполным типом массива, поэтому его размер не может быть рассчитан с использованием `sizeof` .

использование

Вы можете объявить и инициализировать объект с типом структуры, содержащим гибкий член массива, но вы не должны пытаться инициализировать элемент гибкого массива, поскольку он рассматривается так, как если бы он не существовал. Запрещается пытаться это сделать, и результатом будут ошибки компиляции.

Точно так же вы не должны пытаться присвоить значение любому элементу гибкого элемента массива при объявлении структуры таким образом, поскольку в конце структуры может быть недостаточно заполнения, чтобы разрешить любые объекты, требуемые элементом гибкого массива. Однако компилятор не обязательно помешает вам сделать это, так что это может привести к неопределенному поведению.

```
/* invalid: cannot initialize flexible array member */
struct ex1 e1 = {1, {2, 3}};
/* invalid: hdr={foo=1, bar=2} OK, but cannot initialize flexible array member */
struct ex2 e2 = {{1, 2}, {3}};
/* valid: initialize foo=1, bar=2 members */
struct ex3 e3 = {1, 2};

e1.flex[0] = 3; /* undefined behavior, in my case */
e3.flex[0] = 2; /* undefined behavior again */
```

```
e2.flex[0] = e3.flex[0]; /* undefined behavior */
```

Вместо этого вы можете использовать `malloc` , `calloc` или `realloc` для распределения структуры с дополнительным хранилищем, а затем бесплатно, что позволяет вам использовать гибкий член массива по своему усмотрению:

```
/* valid: allocate an object of structure type `ex1` along with an array of 2 ints */
struct ex1 *pe1 = malloc(sizeof(*pe1) + 2 * sizeof(pe1->flex[0]));

/* valid: allocate an object of structure type ex2 along with an array of 4 ints */
struct ex2 *pe2 = malloc(sizeof(struct ex2) + sizeof(int[4]));

/* valid: allocate 5 structure type ex3 objects along with an array of 3 ints per object */
struct ex3 *pe3 = malloc(5 * (sizeof(*pe3) + sizeof(int[3])));

pe1->flex[0] = 3; /* valid */
pe3[0]->flex[0] = pe1->flex[0]; /* valid */
```

C99

«Структурный взлом»

Элементы гибкого массива не существовали до C99 и рассматриваются как ошибки. Общим обходным путем является объявление массива длиной 1, метод, называемый «struct hack»:

```
struct ex1
{
    size_t foo;
    int flex[1];
};
```

Однако это повлияет на размер структуры, в отличие от истинного элемента гибкого массива:

```
/* Prints "8,4,16" on my machine, signifying that there are 4 bytes of padding. */
printf("%d,%d,%d\n", (int)sizeof(size_t), (int)sizeof(int[1]), (int)sizeof(struct ex1));
```

Чтобы использовать элемент `flex` как элемент гибкого массива, вы должны выделить его с помощью `malloc` как показано выше, за исключением того, что `sizeof(*pe1)` (или эквивалентный `sizeof(struct ex1)`) будет заменен на `offsetof(struct ex1, flex)` или более длинный, тип-агностический `sizeof(*pe1)-sizeof(pe1->flex)` выражения `sizeof(*pe1)-sizeof(pe1->flex)` . В качестве альтернативы вы можете вычесть 1 из требуемой длины «гибкого» массива, поскольку она уже включена в размер структуры, если желаемая длина больше 0. Такая же логика может быть применена к другим примерам использования.

Совместимость

Если требуется совместимость с компиляторами, которые не поддерживают гибкие

элементы массива, вы можете использовать макрос, определенный как `FLEXMEMB_SIZE` ниже:

```
#if __STDC_VERSION__ < 199901L
#define FLEXMEMB_SIZE 1
#else
#define FLEXMEMB_SIZE /* nothing */
#endif

struct ex1
{
    size_t foo;
    int flex[FLEXMEMB_SIZE];
};
```

При распределении объектов вы должны использовать форму `offsetof(struct ex1, flex)` чтобы ссылаться на размер структуры (за исключением элемента гибкого массива), поскольку это единственное выражение, которое останется согласованным между компиляторами, которые поддерживают гибкие члены массива и компиляторы, которые делают не:

```
struct ex1 *pe10 = malloc(offsetof(struct ex1, flex) + n * sizeof(pe10->flex[0]));
```

Альтернативой является использование препроцессора для условного вычитания 1 из указанной длины. Из-за возросшего потенциала несогласованности и общей человеческой ошибки в этой форме я переместил логику в отдельную функцию:

```
struct ex1 *ex1_alloc(size_t n)
{
    struct ex1 tmp;
#if __STDC_VERSION__ < 199901L
    if (n != 0)
        n--;
#endif
    return malloc(sizeof(tmp) + n * sizeof(tmp.flex[0]));
}
...

/* allocate an ex1 object with "flex" array of length 3 */
struct ex1 *pe1 = ex1_alloc(3);
```

Передача структур в функции

В C все аргументы передаются функциям по значению, включая структуры. Для небольших структур это хорошо, так как это означает, что нет никаких накладных расходов на доступ к данным через указатель. Тем не менее, также очень легко случайно передать огромную структуру, что приводит к низкой производительности, особенно если программист используется для других языков, где аргументы передаются по ссылке.

```
struct coordinates
{
    int x;
```

```
    int y;
    int z;
};

// Passing and returning a small struct by value, very fast
struct coordinates move(struct coordinates position, struct coordinates movement)
{
    position.x += movement.x;
    position.y += movement.y;
    position.z += movement.z;
    return position;
}

// A very big struct
struct lotsOfData
{
    int param1;
    char param2[80000];
};

// Passing and returning a large struct by value, very slow!
// Given the large size of the struct this could even cause stack overflow
struct lotsOfData doubleParam1(struct lotsOfData value)
{
    value.param1 *= 2;
    return value;
}

// Passing the large struct by pointer instead, fairly fast
void doubleParam1ByPtr(struct lotsOfData *value)
{
    value->param1 *= 2;
}
```

Объектно-ориентированное программирование с использованием структур

Структуры могут использоваться для реализации кода объектно-ориентированным способом. Структура похожа на класс, но в ней отсутствуют функции, которые обычно также являются частью класса, мы можем добавить их как переменные-члены указателя функции. Чтобы остановиться на нашем примере с координатами:

```
/* coordinates.h */

typedef struct coordinate_s
{
    /* Pointers to method functions */
    void (*setx)(coordinate *this, int x);
    void (*sety)(coordinate *this, int y);
    void (*print)(coordinate *this);
    /* Data */
    int x;
    int y;
} coordinate;

/* Constructor */
coordinate *coordinate_create(void);
```

```
/* Destructor */
void coordinate_destroy(coordinate *this);
```

И теперь реализующий файл C:

```
/* coordinates.c */

#include "coordinates.h"
#include <stdio.h>
#include <stdlib.h>

/* Constructor */
coordinate *coordinate_create(void)
{
    coordinate *c = malloc(sizeof(*c));
    if (c != 0)
    {
        c->setx = &coordinate_setx;
        c->sety = &coordinate_sety;
        c->print = &coordinate_print;
        c->x = 0;
        c->y = 0;
    }
    return c;
}

/* Destructor */
void coordinate_destroy(coordinate *this)
{
    if (this != NULL)
    {
        free(this);
    }
}

/* Methods */
static void coordinate_setx(coordinate *this, int x)
{
    if (this != NULL)
    {
        this->x = x;
    }
}

static void coordinate_sety(coordinate *this, int y)
{
    if (this != NULL)
    {
        this->y = y;
    }
}

static void coordinate_print(coordinate *this)
{
    if (this != NULL)
    {
        printf("Coordinate: (%i, %i)\n", this->x, this->y);
    }
    else
    {
```

```
        printf("NULL pointer exception!\n");
    }
}
```

Пример использования нашего координатного класса:

```
/* main.c */

#include "coordinates.h"
#include <stddef.h>

int main(void)
{
    /* Create and initialize pointers to coordinate objects */
    coordinate *c1 = coordinate_create();
    coordinate *c2 = coordinate_create();

    /* Now we can use our objects using our methods and passing the object as parameter */
    c1->setx(c1, 1);
    c1->sety(c1, 2);

    c2->setx(c2, 3);
    c2->sety(c2, 4);

    c1->print(c1);
    c2->print(c2);

    /* After using our objects we destroy them using our "destructor" function */
    coordinate_destroy(c1);
    c1 = NULL;
    coordinate_destroy(c2);
    c2 = NULL;

    return 0;
}
```

Прочитайте Структуры онлайн: https://riptutorial.com/ru/c/topic/1119/структуры

глава 52: Структуры тестирования

Вступление

Многие разработчики используют модульные тесты для проверки работоспособности своего программного обеспечения. Единичные тесты проверяют небольшие единицы более крупных программных продуктов и обеспечивают соответствие результатов ожиданиям. Структуры тестирования облегчают модульное тестирование, предоставляя услуги по настройке / разрыву и координируя тесты.

Для C. существует множество модульных модулей тестирования. Например, Unity - это чистая C-структура. Люди довольно часто используют C ++-тестирование для проверки кода C; существует множество тестовых рамок C ++.

замечания

Жгут проводов:

TDD - разработка, управляемая тестированием:

Тестирование двойных механизмов в C:

1. Замена ссылки
2. Подстановка указателя функции
3. Замена препроцессора
4. Комбинированная замещающая ссылка и функция

Замечание о средах тестирования C ++, используемых в C: Использование фреймворков C ++ для тестирования программы на C - довольно распространенная практика, как описано здесь .

Examples

CppUTest

CppUTest - это инфраструктура xUnit для модульного тестирования C и C ++. Он написан на C ++ и нацелен на переносимость и простоту в дизайне. Он поддерживает обнаружение утечки памяти, создает издевательства и выполняет свои тесты вместе с тестом Google. Поставляется со вспомогательными скриптами и примерами проектов для Visual Studio и Eclipse CDT.

```
#include <CppUTest/CommandLineTestRunner.h>
#include <CppUTest/TestHarness.h>
```

```
TEST_GROUP(Foo_Group) {}

TEST(Foo_Group, Foo_TestOne) {}

/* Test runner may be provided options, such
   as to enable colored output, to run only a
   specific test or a group of tests, etc. This
   will return the number of failed tests. */

int main(int argc, char ** argv)
{
    RUN_ALL_TESTS(argc, argv);
}
```

У тестовой группы может быть метод `setup()` и `teardown()` . Метод `setup` вызывается перед каждым тестом, после чего `teardown()` метод `teardown()` . Оба они являются необязательными и могут быть опущены независимо. Другие методы и переменные также могут быть объявлены внутри группы и будут доступны для всех тестов этой группы.

```
TEST_GROUP(Foo_Group)
{
    size_t data_bytes = 128;
    void * data;

    void setup()
    {
        data = malloc(data_bytes);
    }

    void teardown()
    {
        free(data);
    }

    void clear()
    {
        memset(data, 0, data_bytes);
    }
}
```

Схема тестирования Unity

Unity - это тестовая платформа xUnit для модульного тестирования C. Она полностью написана на C и является переносной, быстрой, простой, выразительной и расширяемой. Он специально предназначен для модульного тестирования встроенных систем.

Простой тестовый пример, который проверяет возвращаемое значение функции, может выглядеть следующим образом

```
void test_FunctionUnderTest_should_ReturnFive(void)
{
    TEST_ASSERT_EQUAL_INT( 5, FunctionUnderTest() );
```

```
}
```

Полный тестовый файл может выглядеть так:

```
#include "unity.h"
#include "UnitUnderTest.h" /* The unit to be tested. */

void setUp (void) {} /* Is run before every test, put unit init calls here. */
void tearDown (void) {} /* Is run after every test, put unit clean-up calls here. */

void test_TheFirst(void)
{
    TEST_IGNORE_MESSAGE("Hello world!"); /* Ignore this test but print a message. */
}

int main (void)
{
    UNITY_BEGIN();
    RUN_TEST(test_TheFirst); /* Run the test. */
    return UNITY_END();
}
```

Unity поставляется с примерами проектов, make-файлов и некоторых скриптов Ruby rake, которые упрощают создание более длинных тестовых файлов.

CMocka

CMocka - это элегантная модульная система тестирования для C с поддержкой макетных объектов. Он требует только стандартной библиотеки C, работает на ряде вычислительных платформ (включая встроенные) и с разными компиляторами. В нем есть учебное пособие по тестированию с помощью mocks, документации по API и множеству примеров .

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>

void null_test_success (void ** state) {}

void null_test_fail (void ** state)
{
    assert_true (0);
}

/* These functions will be used to initialize
   and clean resources up after each test run */
int setup (void ** state)
{
    return 0;
}

int teardown (void ** state)
{
    return 0;
```

```
}


int main (void)
{
    const struct CMUnitTest tests [] =
    {
        cmocka_unit_test (null_test_success),
        cmocka_unit_test (null_test_fail),
    };

    /* If setup and teardown functions are not
       needed, then NULL may be passed instead */

    int count_fail_tests =
        cmocka_run_group_tests (tests, setup, teardown);

    return count_fail_tests;
}
```

Прочитайте Структуры тестирования онлайн: https://riptutorial.com/ru/c/topic/6779/структуры-тестирования

глава 53: Струны

Вступление

В C строка не является внутренним типом. C-строка - это соглашение, чтобы иметь одномерный массив символов, который заканчивается нулевым символом, с помощью `'\0'` .

Это означает, что C-строка с содержанием `"abc"` будет иметь четыре символа `'a'` , `'b'` , `'c'` и `'\0'` .

См. Базовое введение в пример строк .

Синтаксис

- char str1 [] = "Привет, мир!"; / * Модифицируемая * /
- char str2 [14] = «Привет, мир!»; / * Модифицируемая * /
- char * str3 = «Привет, мир!»; / * Немодифицируемые * /

Examples

Вычислить длину: strlen ()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        puts("Argument missing.");
        return EXIT_FAILURE;
    }

    size_t len = strlen(argv[1]);
    printf("The length of the second argument is %zu.\n", len);

    return EXIT_SUCCESS;
}
```

Эта программа вычисляет длину своего второго входного аргумента и сохраняет результат в `len` . Затем он печатает эту длину на терминале. Например, при запуске с параметрами `program_name "Hello, world!"` , программа выведет `The length of the second argument is 13.` потому что строка `Hello, world!` имеет длину 13 символов.

`strlen` подсчитывает все **байты** от начала строки до, но не включает завершающий символ

NUL, `'\0'` . Таким образом, его можно использовать только тогда, когда строка *гарантированно* завершена NUL.

Также имейте в виду, что если строка содержит любые символы Unicode, `strlen` не укажет, сколько символов в строке (так как некоторые символы могут иметь несколько байтов). В таких случаях вам нужно подсчитать символы (*например* , единицы кода) самостоятельно. Рассмотрим выход следующего примера:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char asciiString[50] = "Hello world!";
    char utf8String[50] = "Γειά σου Κόσμε!"; /* "Hello World!" in Greek */

    printf("asciiString has %zu bytes in the array\n", sizeof(asciiString));
    printf("utf8String has %zu bytes in the array\n", sizeof(utf8String));
    printf("\"%s\" is %zu bytes\n", asciiString, strlen(asciiString));
    printf("\"%s\" is %zu bytes\n", utf8String, strlen(utf8String));
}
```

Выход:

```
asciiString has 50 bytes in the array
utf8String has 50 bytes in the array
"Hello world!" is 12 bytes
"Γειά σου Κόσμε!" is 27 bytes
```

Копировать и объединить: strcpy (), strcat ()

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  /* Always ensure that your string is large enough to contain the characters
   * and a terminating NUL character ('\0')!
   */
  char mystring[10];

  /* Copy "foo" into `mystring`, until a NUL character is encountered. */
  strcpy(mystring, "foo");
  printf("%s\n", mystring);

  /* At this point, we used 4 chars of `mystring`, the 3 characters of "foo",
   * and the NUL terminating byte.
   */

  /* Append "bar" to `mystring`. */
  strcat(mystring, "bar");
  printf("%s\n", mystring);

  /* We now use 7 characters of `mystring`: "foo" requires 3, "bar" requires 3
```

```
   * and there is a terminating NUL character ('\0') at the end.
   */

  /* Copy "bar" into `mystring`, overwriting the former contents. */
  strcpy(mystring, "bar");
  printf("%s\n", mystring);

  return 0;
}
```

Выходы:

```
foo
foobar
bar
```

Если вы добавляете или копируете из существующей строки, убедитесь, что она завершена NUL!

Строковые литералы (например, `"foo"`) всегда будут завершаться NUL компилятором.

Comparsion: strcmp (), strncmp (), strcasecmp (), strncasecmp ()

`strcase*` -функции не являются стандартными C, а расширением POSIX.

Функция `strcmp` лексикографически сравнивает два массива символов с нулевым символом. Функции возвращают отрицательное значение, если первый аргумент появляется перед вторым в лексикографическом порядке, ноль, если они сравнивают равный или положительный, если первый аргумент появляется после второго в лексикографическом порядке.

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcmp(lhs, rhs); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "BBB");
    compare("BBB", "CCCCC");
    compare("BBB", "AAAAAA");
    return 0;
}
```

Выходы:

```
BBB equals BBB
BBB comes before CCCCC
BBB comes after AAAAAA
```

В `strcmp` `strcasecmp` функция `strcasecmp` также сравнивает лексикографически ее аргументы после перевода каждого символа в его нижний регистр:

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcasecmp(lhs, rhs); // compute case-insensitive comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "bBB");
    compare("BBB", "ccCCC");
    compare("BBB", "aaaaaa");
    return 0;
}
```

Выходы:

```
BBB equals bBB
BBB comes before ccCCC
BBB comes after aaaaaa
```

`strncmp` и `strncasecmp` сравнивают не более n символов:

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs, int n)
{
    int result = strncmp(lhs, rhs, n); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
```

```
{
    compare("BBB", "Bb", 1);
    compare("BBB", "Bb", 2);
    compare("BBB", "Bb", 3);
    return 0;
}
```

Выходы:

```
BBB equals Bb
BBB comes before Bb
BBB comes before Bb
```

Tokenisation: strtok (), strtok_r () и strtok_s ()

Функция `strtok` разбивает строку на меньшие строки или жетоны, используя набор разделителей.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int toknum = 0;
    char src[] = "Hello,, world!";
    const char delimiters[] = ", !";
    char *token = strtok(src, delimiters);
    while (token != NULL)
    {
        printf("%d: [%s]\n", ++toknum, token);
        token = strtok(NULL, delimiters);
    }
    /* source is now "Hello\0, world\0\0" */
}
```

Выход:

```
1: [Hello]
2: [world]
```

Строка разделителей может содержать один или несколько разделителей, и с каждым вызовом `strtok` могут использоваться разные строки разделителя.

Вызовы в `strtok` для продолжения токенизации одной и той же исходной строки не должны снова передавать исходную строку, а вместо этого передавать `NULL` в качестве первого аргумента. Если же исходная строка передается затем первый маркер вместо этого будет повторно лексемы. То есть, учитывая те же разделители, `strtok` просто вернет первый токен снова.

Обратите внимание: поскольку `strtok` не выделяет новую память для токенов, *она изменяет исходную строку* . То есть в приведенном выше примере строка `src` будет обрабатываться

для создания маркеров, на которые ссылается указатель, возвращаемый вызовами `strtok` . Это означает, что исходная строка не может быть `const` (поэтому она не может быть строковым литералом). Это также означает, что личность разделительного байта теряется (т. Е. В примере «,» и «!» Эффективно удаляются из исходной строки, и вы не можете определить, какой символ разделителя совпадают).

Заметим также, что несколько последовательных разделителей в исходной строке рассматриваются как один; в этом примере вторая запятая игнорируется.

`strtok` является ни потокобезопасным, ни повторным, поскольку он использует статический буфер при разборе. Это означает, что если функция вызывает `strtok` , никакая функция, которую она вызывает при использовании `strtok` также может использовать `strtok` , и она не может быть вызвана любой функцией, которая сама использует `strtok` .

Пример, демонстрирующий проблемы, вызванные тем, что `strtok` не является повторным, выглядит следующим образом:

```
char src[] = "1.2,3.5,4.2";
char *first = strtok(src, ",");

do
{
    char *part;
    /* Nested calls to strtok do not work as desired */
    printf("[%s]\n", first);
    part = strtok(first, ".");
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok(NULL, ".");
    }
} while ((first = strtok(NULL, ",")) != NULL);
```

Выход:

```
[1.2]
 [1]
 [2]
```

Ожидаемая операция является то , что наружный `do while` во `"1.2"` `"3.5"` `"4.2"` `strtok` `do while` цикла должно создать три маркера , состоящие из каждой строки десятичных чисел (`"1.2"` , `"3.5"` , `"4.2"`), для каждого из которых `strtok` предусматривает внутренний цикл должен разделить его на отдельные (`"1"` , `"2"` , `"3"` , `"5"` , `"4"` , `"2"`).

Однако, поскольку `strtok` не является повторным, этого не происходит. Вместо этого первый `strtok` правильно создает токен «1,2 \ 0», а внутренний цикл правильно создает токены `"1"` и `"2"` . Но тогда `strtok` во внешнем цикле находится в конце строки, используемой внутренним циклом, и немедленно возвращает NULL. Вторая и третья подстроки массива `src` вообще не анализируются.

C11

Стандартные библиотеки C не содержат потокобезопасную или повторную версию, но некоторые другие, например POSIX ' `strtok_r` . Обратите внимание, что в MSVC эквивалент `strtok` `strtok_s` является потокобезопасным.

C11

C11 имеет необязательную часть, приложение K, которая предлагает поточно-безопасную и повторную версию с именем `strtok_s` . Вы можете протестировать эту функцию с помощью `__STDC_LIB_EXT1__` . Эта дополнительная часть не поддерживается широко.

Функция `strtok_s` отличается от функции POSIX `strtok_r` ее от хранения за пределами токенированной строки и проверяя ограничения времени выполнения. Однако при правильно написанных программах `strtok_s` и `strtok_r` ведут себя одинаково.

Использование `strtok_s` с примером теперь дает правильный ответ, например:

```
/* you have to announce that you want to use Annex K */
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

#ifndef __STDC_LIB_EXT1__
# error "we need strtok_s from Annex K"
#endif

char src[] = "1.2,3.5,4.2";
char *next = NULL;
char *first = strtok_s(src, ",", &next);

do
{
    char *part;
    char *posn;

    printf("[%s]\n", first);
    part = strtok_s(first, ".", &posn);
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok_s(NULL, ".", &posn);
    }
}
while ((first = strtok_s(NULL, ",", &next)) != NULL);
```

И выход будет:

```
[1.2]
 [1]
 [2]
[3.5]
 [3]
 [5]
[4.2]
 [4]
```

```
 [2]
```

Найти первое / последнее вхождение определенного символа: strchr (), strrchr ()

Функции `strchr` и `strrchr` находят символ в строке, то есть в символьном массиве с нулевым символом. `strchr` возвращает указатель на первое вхождение и `strrchr` на последний.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char toSearchFor = 'A';

    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        printf("Argument missing.\n");
        return EXIT_FAILURE;
    }

    {
        char *firstOcc = strchr(argv[1], toSearchFor);
        if (firstOcc != NULL)
        {
            printf("First position of %c in %s is %td.\n",
              toSearchFor, argv[1], firstOcc-argv[1]); /* A pointer difference's result
                                is a signed integer and uses the length modifier 't'. */
        }
        else
        {
            printf("%c is not in %s.\n", toSearchFor, argv[1]);
        }
    }

    {
        char *lastOcc = strrchr(argv[1], toSearchFor);
        if (lastOcc != NULL)
        {
            printf("Last position of %c in %s is %td.\n",
              toSearchFor, argv[1], lastOcc-argv[1]);
        }
    }

    return EXIT_SUCCESS;
}
```

Выходы (после генерации исполняемого файла с именем `pos`):

```
$ ./pos AAAAAAA
First position of A in AAAAAAA is 0.
Last position of A in AAAAAAA is 6.
$ ./pos BAbbbbbAccccAAAAzzz
First position of A in BAbbbbbAccccAAAAzzz is 1.
```

```
Last position of A in BAbbbbbAccccAAAAzzz is 15.
$  ./pos qwerty
A is not in qwerty.
```

Одним из распространенных `strrchr` использования `strrchr` является извлечение имени файла из пути. Например, чтобы извлечь `myfile.txt` из `C:\Users\eak\myfile.txt` :

```
char *getFileName(const char *path)
{
    char *pend;

    if ((pend = strrchr(path, '\')) != NULL)
        return pend + 1;

    return NULL;
}
```

Итерация над символами в строке

Если мы знаем длину строки, мы можем использовать цикл for для итерации по своим символам:

```
char * string = "hello world"; /* This 11 chars long, excluding the 0-terminator. */
size_t i = 0;
for (; i < 11; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

В качестве альтернативы мы можем использовать стандартную функцию `strlen()` чтобы получить длину строки, если мы не знаем, что такое строка:

```
size_t length = strlen(string);
size_t i = 0;
for (; i < length; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

Наконец, мы можем воспользоваться тем фактом, что строки в C гарантируют завершение нулями (что мы уже делали, передавая его в `strlen()` в предыдущем примере ;-)). Мы можем выполнять итерацию по массиву независимо от его размера и останавливать итерацию после достижения нулевого символа:

```
size_t i = 0;
while (string[i] != '\0') {       /* Stop looping when we reach the null-character. */
    printf("%c\n", string[i]);    /* Print each character of the string. */
    i++;
}
```

Основное введение в строки

В C **строка** представляет собой последовательность символов, которая заканчивается нулевым символом ('\ 0').

Мы можем создавать строки с использованием **строковых литералов** , которые представляют собой последовательности символов, окруженных двойными кавычками; например, взять строковый литерал `"hello world"` . Строковые литералы автоматически завершают нуль.

Мы можем создавать строки, используя несколько методов. Например, мы можем объявить `char *` и инициализировать его, чтобы указать на первый символ строки:

```
char * string = "hello world";
```

При инициализации `char *` в строковой константе, как указано выше, сама строка обычно выделяется в данных только для чтения; `string` - это указатель на первый элемент массива, который является символом `'h'` .

Поскольку строковый литерал выделяется в постоянной памяти, он не модифицируется [1] . Любая попытка изменить его приведет к неопределенному поведению , поэтому лучше добавить `const` чтобы получить ошибку времени компиляции

```
char const * string = "hello world";
```

Он имеет аналогичный эффект [2] как

```
char const string_arr[] = "hello world";
```

Чтобы создать изменяемую строку, вы можете объявить массив символов и инициализировать его содержимое с помощью строкового литерала, например:

```
char modifiable_string[] = "hello world";
```

Это эквивалентно следующему:

```
char modifiable_string[] = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
```

Поскольку во второй версии используется инициализатор с закрытой фигурной скобкой, строка не будет автоматически завершаться нулем, если символ `'\0'` явно не включен в массив символов, обычно как последний элемент.

---

1 Немодифицируемый означает, что символы в строковом литерале не могут быть изменены, но помните, что `string` указателя может быть изменена (может указывать в другом месте или может быть увеличена или уменьшена).

2 Обе строки имеют аналогичный эффект в том смысле, что символы обеих строк не могут быть изменены.

Следует отметить, что `string` является указателем на `char` и это модифицируемое значение l, поэтому его можно увеличивать или указывать на какое-либо другое местоположение, в то время как массив `string_arr` является немодифицируемым значением l, его нельзя изменить.

Создание массивов строк

Массив строк может означать пару вещей:

1. Массив, элементы которого `char *` s
2. Массив, элементы которого являются массивами `char`

Мы можем создать массив указателей символов, например:

```
char * string_array[] = {
    "foo",
    "bar",
    "baz"
};
```

Помните: когда мы назначаем строковые литералы `char *` , сами строки выделяются в постоянной памяти. Однако массив `string_array` выделяется в памяти чтения / записи. Это означает, что мы можем изменить указатели в массиве, но мы не можем изменить строки, на которые они указывают.

В C параметр main `argv` (массив аргументов командной строки, переданных при запуске программы) представляет собой массив `char *` : `char * argv[]` .

Мы также можем создавать массивы массивов символов. Поскольку строки являются массивами символов, массив строк - это просто массив, элементами которого являются массивы символов:

```
char modifiable_string_array_literals[][4] = {
    "foo",
    "bar",
    "baz"
};
```

Это эквивалентно:

```
char modifiable_string_array[][4] = {
    {'f', 'o', 'o', '\0'},
    {'b', 'a', 'r', '\0'},
    {'b', 'a', 'z', '\0'}
};
```

Обратите внимание, что мы указываем `4` как размер второго измерения массива; каждая из строк в нашем массиве на самом деле составляет 4 байта, так как мы должны включать нуль-завершающий символ.

strstr

```
/* finds the next instance of needle in haystack
   zbpos: the zero-based position to begin searching from
   haystack: the string to search in
   needle: the string that must be found
   returns the next match of `needle` in `haystack`, or -1 if not found
*/
int findnext(int zbpos, const char *haystack, const char *needle)
{
    char *p;

    if (((p = strstr(haystack + zbpos, needle)) != NULL)
        return p - haystack;

    return -1;
}
```

`strstr` ищет аргумент `haystack` (первый) для строки, на которую указывает `needle` . Если найдено, `strstr` возвращает адрес события. Если он не мог найти `needle` , он возвращает NULL. Мы используем `zbpos` чтобы мы не продолжали находить ту же самую иглу снова и снова. Чтобы пропустить первый экземпляр, добавим смещение `zbpos` . Клон Notepad может вызвать `findnext` как это, чтобы реализовать свой диалог «Найти дальше»:

```
/*
    Called when the user clicks "Find Next"
    doc: The text of the document to search
    findwhat: The string to find
*/
void onfindnext(const char *doc, const char *findwhat)
{
    static int i;

    if ((i = findnext(i, doc, findwhat)) != -1)
        /* select the text starting from i and ending at i + strlen(findwhat) */
    else
        /* display a message box saying "end of search" */
}
```

Строковые литералы

Строковые литералы представляют нулевые завершающие массивы статической продолжительности `char` . Поскольку у них есть статическая продолжительность хранения, строковый литерал или указатель на один и тот же базовый массив можно безопасно использовать несколькими способами, которые не может указывать указатель на автоматический массив. Например, возврат строкового литерала из функции имеет четко определенное поведение:

```
const char *get_hello() {
    return "Hello, World!";  /* safe */
}
```

По историческим причинам элементы массива, соответствующие строковому литералу, формально не `const` . Тем не менее, любая попытка изменить их имеет неопределенное поведение . Как правило, программа, которая пытается изменить массив, соответствующий строковому литералу, выйдет из строя или иным образом неисправна.

```
char *foo = "hello";
foo[0] = 'y';  /* Undefined behavior - BAD! */
```

Если указатель указывает на строковый литерал - или где он иногда может это сделать - желательно объявить референта `const` указателя, чтобы избежать случайного использования такого неопределенного поведения.

```
const char *foo = "hello";
/* GOOD: can't modify the string pointed to by foo */
```

С другой стороны, указатель на или в базовый массив строкового литерала сам по себе не является особенным; его значение может быть свободно изменено, чтобы указать на что-то еще:

```
char *foo = "hello";
foo = "World!"; /* OK - we're just changing what foo points to */
```

Более того, хотя инициализаторы для массивов `char` могут иметь ту же форму, что и строковые литералы, использование такого инициализатора не связывает характеристики строкового литерала с инициализированным массивом. Инициализатор просто обозначает длину и начальное содержимое массива. В частности, элементы могут быть изменены, если явно не объявлены `const` :

```
char foo[] = "hello";
foo[0] = 'y';  /* OK! */
```

Обнуление строки

Вы можете вызвать `memset` чтобы обнулить строку (или любой другой блок памяти).

Где `str` - строка с нулевым значением, а `n` - количество байтов в строке.

```
#include <stdlib.h> /* For EXIT_SUCCESS */
#include <stdio.h>
#include <string.h>


int main(void)
{
  char str[42] = "fortytwo";
  size_t n = sizeof str; /* Take the size not the length. */

  printf("'%s'\n", str);
```

```
  memset(str, '\0', n);

  printf("'%s'\n", str);

  return EXIT_SUCCESS;
}
```

Печать:

```
'fortytwo'
''
```

Другой пример:

```
#include <stdlib.h> /* For EXIT_SUCCESS */
#include <stdio.h>
#include <string.h>


#define FORTY_STR "forty"
#define TWO_STR "two"

int main(void)
{
  char str[42] = FORTY_STR TWO_STR;
  size_t n = sizeof str; /* Take the size not the length. */
  char * point_to_two = strstr(str, TWO_STR);

  printf("'%s'\n", str);

  memset(point_to_two, '\0', n);

  printf("'%s'\n", str);

  memset(str, '\0', n);

  printf("'%s'\n", str);

  return EXIT_SUCCESS;
}
```

Печать:

```
'fortytwo'
'forty'
''
```

strspn и strcspn

Учитывая строку, `strspn` вычисляет длину начальной подстроки (span), состоящей исключительно из определенного списка символов. `strcspn` аналогичен, за исключением того, что вычисляет длину исходной подстроки, состоящей из любых символов, кроме перечисленных:

```
/*
  Provided a string of "tokens" delimited by "separators", print the tokens along
  with the token separators that get skipped.
*/
#include <stdio.h>
#include <string.h>

int main(void)
{
    const char sepchars[] = ",.;!?";
    char foo[] = ";ball call,.fall gall hall!?.,";
    char *s;
    int n;

    for (s = foo; *s != 0; /*empty*/) {
        /* Get the number of token separator characters. */
        n = (int)strspn(s, sepchars);

        if (n > 0)
            printf("skipping separators: << %.*s >> (length=%d)\n", n, s, n);

        /* Actually skip the separators now. */
        s += n;

        /* Get the number of token (non-separator) characters. */
        n = (int)strcspn(s, sepchars);

        if (n > 0)
            printf("token found: << %.*s >> (length=%d)\n", n, s, n);

        /* Skip the token now. */
        s += n;
    }

    printf("== token list exhausted ==\n");

    return 0;
}
```

Аналогичные функции с использованием широкоформатных строк - `wcsspn` и `wcscspn` ; они используются одинаково.

Копирование строк

Назначения указателей не копируют строки

Вы можете использовать оператор `=` для копирования целых чисел, но вы не можете использовать оператор `=` для копирования строк в C. Строки в C представлены в виде массивов символов с завершающим нулевым символом, поэтому использование оператора `=` будет сохранять только адрес (указатель) строки.

```
#include <stdio.h>

int main(void) {
    int a = 10, b;
    char c[] = "abc", *d;

    b = a; /* Integer is copied */
    a = 20; /* Modifying a leaves b unchanged - b is a 'deep copy' of a */
    printf("%d %d\n", a, b); /* "20 10" will be printed */

    d = c;
    /* Only copies the address of the string -
    there is still only one string stored in memory */

    c[1] = 'x';
    /* Modifies the original string - d[1] = 'x' will do exactly the same thing */

    printf("%s %s\n", c, d); /* "axc axc" will be printed */

    return 0;
}
```

Вышеприведенный пример скомпилирован, потому что мы использовали `char *d` а не `char d[3]` . Использование последнего приведет к ошибке компилятора. Вы не можете назначать массивы в C.

```
#include <stdio.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    b = a; /* compile error */
    printf("%s\n", b);

    return 0;
}
```

Копирование строк с использованием стандартных функций

strcpy()

Чтобы фактически скопировать строки, функция `strcpy()` доступна в `string.h` . Перед копированием достаточно места для места назначения.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "abc";
    char b[8];
```

```
    strcpy(b, a); /* think "b special equals a" */
    printf("%s\n", b); /* "abc" will be printed */

    return 0;
}
```

C99

snprintf()

Чтобы избежать переполнения буфера, можно использовать `snprintf()` . Это не лучшее решение по производительности, так как оно должно анализировать строку шаблона, но это единственная безопасная для буфера функция для копирования строк, доступных в стандартной библиотеке, которые можно использовать без каких-либо дополнительных шагов.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "012345678901234567890";
    char b[8];

#if 0
    strcpy(b, a); /* causes buffer overrun (undefined behavior), so do not execute this here!
*/
#endif

    snprintf(b, sizeof(b), "%s", a); /* does not cause buffer overrun */
    printf("%s\n", b); /* "0123456" will be printed */

    return 0;
}
```

strncat()

Второй вариант с лучшей производительностью - использовать `strncat()` (версия проверки переполнения буфера `strcat()`) - он принимает третий аргумент, который сообщает ему максимальное количество копируемых байтов:

```
char dest[32];

dest[0] = '\0';
strncat(dest, source, sizeof(dest) - 1);
    /* copies up to the first (sizeof(dest) - 1) elements of source into dest,
    then puts a \0 on the end of dest */
```

Обратите внимание, что эта формулировка использует `sizeof(dest) - 1` ; это важно, потому что `strncat()` всегда добавляет нулевой байт (хороший), но не считает, что размер строки (причина замешательства и перезаписывания буфера).

Также обратите внимание, что альтернатива - конкатенация после непустой строки - еще

более чревата. Рассматривать:

```
char dst[24] = "Clownfish: ";
char src[] = "Marvin and Nemo";
size_t len = strlen(dst);

strncat(dst, src, sizeof(dst) - len - 1);
printf("%zu: [%s]\n", strlen(dst), dst);
```

Выход:

```
23: [Clownfish: Marvin and N]
```

Обратите внимание, однако, что размер, указанный в качестве длины, *не* был размером целевого массива, а объемом оставшегося в нем места, не считая байт нулевого терминала. Это может вызвать большие проблемы с перезаписи. Это также немного расточительно; для правильного указания аргумента длины вы знаете длину данных в месте назначения, поэтому вместо этого вы можете указать адрес нулевого байта в конце существующего содержимого, сохраняя `strncat()` при его повторном сканировании:

```
    strcpy(dst, "Clownfish: ");
    assert(len < sizeof(dst) - 1);
    strncat(dst + len, src, sizeof(dst) - len - 1);
    printf("%zu: [%s]\n", strlen(dst), dst);
```

Это дает тот же результат, что и раньше, но `strncat()` не должен проверять существующее содержимое `dst` прежде чем он начнет копирование.

strncpy()

Последним вариантом является функция `strncpy()` . Хотя вы можете подумать, что это должно быть на первом месте, это довольно обманчивая функция, которая имеет две основные ошибки:

1. Если копирование через `strncpy()` попадает в предел буфера, завершающий нулевой символ не будет записан.
2. `strncpy()` всегда полностью заполняет пункт назначения, при необходимости принимает нулевые байты.

(Такая причудливая реализация является исторической и первоначально была предназначена для обработки имен файлов UNIX)

Единственный правильный способ его использования - вручную обеспечить нулевое завершение:

```
strncpy(b, a, sizeof(b)); /* the third parameter is destination buffer size */
b[sizeof(b)/sizeof(*b) - 1] = '\0'; /* terminate the string */
printf("%s\n", b); /* "0123456" will be printed */
```

Даже тогда, если у вас есть большой буфер, становится очень неэффективно использовать `strncpy()` из-за дополнительного заполнения нулями.

Преобразуйте строки в число: atoi (), atof () (опасно, не используйте их)

Предупреждение. Функции `atoi` , `atol` , `atoll` и `atof` по своей сути являются небезопасными, потому что: *если значение результата не может быть представлено, поведение не определено.* (7.20.1p1)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    int val;
    if (argc < 2)
    {
        printf("Usage: %s <integer>\n", argv[0]);
        return 0;
    }

    val = atoi(argv[1]);

    printf("String value = %s, Int value = %d\n", argv[1], val);

    return 0;
}
```

Когда строка, подлежащая преобразованию, является допустимым десятичным целым, находящимся в диапазоне, функция работает:

```
$ ./atoi 100
String value = 100, Int value = 100
$ ./atoi 200
String value = 200, Int value = 200
```

Для строк, начинающихся с числа, а затем чего-то другого, обрабатывается только начальное число:

```
$ ./atoi 0x200
0
$ ./atoi 0123x300
123
```

Во всех остальных случаях поведение не определено:

```
$ ./atoi hello
Formatting the hard disk...
```

Из-за двусмысленностей выше и этого неопределенного поведения семейство функций `atoi` никогда не должно использоваться.

- Чтобы преобразовать в `long int` , используйте `strtol()` вместо `atol()` .
- Чтобы преобразовать в `double` , используйте `strtod()` вместо `atof()` .

C99

- Чтобы преобразовать в `long long int` , используйте `strtoll()` вместо `atoll()` .

чтение / запись в формате форматированных строк

Запись форматированных данных в строку

```
int sprintf ( char * str, const char * format, ... );
```

используйте функцию `sprintf` для записи данных float в строку.

```
#include <stdio.h>
int main ()
{
  char buffer [50];
  double PI = 3.1415926;
  sprintf (buffer, "PI = %.7f", PI);
  printf ("%s\n",buffer);
  return 0;
}
```

Чтение форматированных данных из строки

```
int sscanf ( const char * s, const char * format, ...);
```

используйте функцию `sscanf` для анализа форматированных данных.

```
#include <stdio.h>
int main ()
{
  char sentence []="date : 06-06-2012";
  char str [50];
  int year;
  int month;
  int day;
  sscanf (sentence,"%s : %2d-%2d-%4d", str, &day, &month, &year);
  printf ("%s -> %02d-%02d-%4d\n",str, day, month, year);
  return 0;
}
```

Безопасное преобразование строк в число: функции strtoX

C99

С C99 библиотека C имеет набор безопасных функций преобразования, которые интерпретируют строку как число. Их имена имеют форму `strtoX` , где `X` является одним из `l` , `ul` , `d` и т. Д., Чтобы определить целевой тип преобразования

```
double strtod(char const* p, char** endptr);
long double strtold(char const* p, char** endptr);
```

Они обеспечивают проверку того, что конверсия имела избыточный или недостаточный поток:

```
double ret = strtod(argv[1], 0); /* attempt conversion */

/* check the conversion result. */
if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)
    return;  /* numeric overflow in in string */
else if (ret == HUGE_VAL && errno == ERANGE)
    return; /* numeric underflow in in string */

/* At this point we know that everything went fine so ret may be used */
```

Если строка фактически не содержит номера, это использование `strtod` возвращает `0.0` .

Если это не является удовлетворительным, можно использовать дополнительный параметр `endptr` . Это указатель на указатель, который будет указывать на конец обнаруженного числа в строке. Если он установлен в `0` , как указано выше, или `NULL` , он просто игнорируется.

Этот параметр `endptr` указывает, было ли успешное преобразование, и если да, то где число закончилось:

```
char *check = 0;
double ret = strtod(argv[1], &check); /* attempt conversion */

/* check the conversion result. */
if (argv[1] == check)
    return; /* No number was detected in string */
else if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)
    return; /* numeric overflow in in string */
else if (ret == HUGE_VAL && errno == ERANGE)
    return; /* numeric underflow in in string */

/* At this point we know that everything went fine so ret may be used */
```

Существуют аналогичные функции для преобразования в более широкие целые типы:

```
long strtol(char const* p, char** endptr, int nbase);
long long strtoll(char const* p, char** endptr, int nbase);
unsigned long strtoul(char const* p, char** endptr, int nbase);
unsigned long long strtoull(char const* p, char** endptr, int nbase);
```

Эти функции имеют третий параметр `nbase` который содержит базу чисел, в которой записано число.

```
long a = strtol("101",   0, 2 ); /* a = 5L */
long b = strtol("101",   0, 8 ); /* b = 65L */
long c = strtol("101",   0, 10); /* c = 101L */
```

```
long d = strtol("101",   0, 16); /* d = 257L */
long e = strtol("101",   0, 0 ); /* e = 101L */
long f = strtol("0101",  0, 0 ); /* f = 65L */
long g = strtol("0x101", 0, 0 ); /* g = 257L */
```

Специальное значение `0` для `nbase` означает, что строка интерпретируется так же, как литералы чисел интерпретируются в программе на C: префикс `0x` соответствует шестнадцатеричному представлению, в противном случае ведущее `0` является восьмеричным, а все остальные числа рассматриваются как десятичные.

Таким образом, наиболее практичным способом интерпретации аргумента командной строки как числа будет

```
int main(int argc, char* argv[] {
    if (argc < 1)
        return EXIT_FAILURE; /* No number given. */

    /* use strtoull because size_t may be wide */
    size_t mySize = strtoull(argv[1], 0, 0);

    /* then check conversion results. */

     ...

    return EXIT_SUCCESS;
}
```

Это означает, что программа может вызываться с параметром в восьмеричном, десятичном или шестнадцатеричном.

Прочитайте Струны онлайн: https://riptutorial.com/ru/c/topic/1990/струны

глава 54: Темы (родной)

Синтаксис

- `#ifndef __STDC_NO_THREADS__`
- `# include <threads.h>`
- `#endif`
- `void call_once(once_flag *flag, void (*func)(void));`
- `int cnd_broadcast(cnd_t *cond);`
- `void cnd_destroy(cnd_t *cond);`
- `int cnd_init(cnd_t *cond);`
- `int cnd_signal(cnd_t *cond);`
- `int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx, const struct timespec *restrict ts);`
- `int cnd_wait(cnd_t *cond, mtx_t *mtx);`
- `void mtx_destroy(mtx_t *mtx);`
- `int mtx_init(mtx_t *mtx, int type);`
- `int mtx_lock(mtx_t *mtx);`
- `int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);`
- `int mtx_trylock(mtx_t *mtx);`
- `int mtx_unlock(mtx_t *mtx);`
- `int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);`
- `thrd_t thrd_current(void);`
- `int thrd_detach(thrd_t thr);`
- `int thrd_equal(thrd_t thr0, thrd_t thr1);`
- `_Noreturn void thrd_exit(int res);`
- `int thrd_join(thrd_t thr, int *res);`
- `int thrd_sleep(const struct timespec *duration, struct timespec* remaining);`
- `void thrd_yield(void);`
- `int tss_create(tss_t *key, tss_dtor_t dtor);`
- `void tss_delete(tss_t key);`
- `void *tss_get(tss_t key);`
- `int tss_set(tss_t key, void *val);`

замечания

Потоки C11 являются дополнительной функцией. Их отсутствие можно протестировать с помощью `__STDC__NO_THREAD__` . В настоящее время (июль 2016) эта функция еще не реализована всеми библиотеками C, которые в противном случае поддерживают C11.

C-библиотеки, которые, как известно, поддерживают потоки C11:

- MUSL

C, которые не поддерживают потоки C11, пока:

- gnu libc

Examples

Начать несколько потоков

```
#include <stdio.h>
#include <threads.h>
#include <stdlib.h>

struct my_thread_data {
   double factor;
};

int my_thread_func(void* a) {
   struct my_thread_data* d = a;
   // do something with d
   printf("we found %g\n", d->factor);
   // return an success or error code
   return d->factor > 1.0;
}


int main(int argc, char* argv[argc+1]) {
    unsigned n = 4;
    if (argc > 1) n = strtoull(argv[1], 0, 0);
    // reserve space for the arguments for the threads
    struct my_thread_data D[n];     // can't be initialized
    for (unsigned i = 0; i < n; ++i) {
         D[i] = (struct my_thread_data){ .factor = 0.5*i, };
    }
    // reserve space for the ID's of the threads
    thrd_t id[4];
    // launch the threads
    for (unsigned i = 0; i < n; ++i) {
         thrd_create(&id[i], my_thread_func, &D[i]);
    }
    // Wait that all threads have finished, but throw away their
    // return values
    for (unsigned i = 0; i < n; ++i) {
         thrd_join(id[i], 0);
    }
    return EXIT_SUCCESS;
}
```

Инициализация одним потоком

В большинстве случаев все данные, к которым обращаются несколько потоков, должны быть инициализированы до создания потоков. Это гарантирует, что все потоки начнутся с четкого состояния, и никаких *условий гонки* не произойдет.

Если это невозможно, можно использовать один `once_flag` и `call_once`

```
#include <threads.h>
#include <stdlib.h>
```

```
// the user data for this example
double const* Big = 0;

// the flag to protect big, must be global and/or static
static once_flag onceBig = ONCE_INIT;

void destroyBig(void) {
   free((void*)Big);
}

void initBig(void) {
    // assign to temporary with no const qualification
    double* b = malloc(largeNum);
    if (!b) {
       perror("allocation failed for Big");
       exit(EXIT_FAILURE);
    }
    // now initialize and store Big
    initializeBigWithSophisticatedValues(largeNum, b);
    Big = b;
    // ensure that the space is freed on exit or quick_exit
    atexit(destroyBig);
    at_quick_exit(destroyBig);
}

// the user thread function that relies on Big
int myThreadFunc(void* a) {
   call_once(&onceBig, initBig);
   // only use Big from here on
   ...
   return 0;
}
```

Параметр `once_flag` используется для координации различных потоков, которые могут захотеть инициализировать одни и те же данные `Big` . Вызов `call_once` гарантирует, что

- `initBig` вызывается ровно один раз
- `call_once` до тех пор, пока такой вызов `initBig` не будет выполнен ни тем же, ни другим потоком.

Помимо выделения, типичной задачей в такой однократно называемой функции является динамическая инициализация структур данных управления потоком, таких как `mtx_t` или `cnd_t` которые не могут быть инициализированы статически, используя `mtx_init` или `cnd_init` соответственно.

Прочитайте Темы (родной) онлайн: https://riptutorial.com/ru/c/topic/4432/темы--родной-

глава 55: Тип Квалификаторы

замечания

Квалификаторы типов - это ключевые слова, которые описывают дополнительную семантику о типе. Они являются неотъемлемой частью сигнатур типа. Они могут отображаться как на самом верхнем уровне объявления (непосредственно влияющем на идентификатор), так и на под-уровнях (что касается только указателей, влияющих на значения, отмеченные):

Ключевое слово	замечания
`const`	Предотвращает мутацию объявленного объекта (если он отображается на самом верхнем уровне) или предотвращает мутацию указательного значения (путем появления рядом с подтипом указателя).
`volatile`	Сообщает компилятору, что объявленный объект (на самом верхнем уровне) или указательное значение (в подтипах указателя) может изменить свое значение в результате внешних условий, а не только в результате потока управления программой.
`restrict`	Рекомендации по оптимизации, относящиеся только к указателям. Объявляет намерение, что для жизни указателя никакие другие указатели не будут использоваться для доступа к одному и тому же объекту с указателем.

Порядок выполнения спецификаторов типов в отношении спецификаторов класса хранения (`static` , `extern` , `auto` , `register`), модификаторы типов (`signed` , `unsigned` , `short` , `long`) и спецификаторы типов (`int` , `char` , `double` и т. Д.) Не применяются, но хорошая практика заключается в том, чтобы привести их в вышеупомянутый порядок:

```
static const volatile unsigned long int a = 5; /* good practice */
unsigned volatile long static int const b = 5; /* bad practice */
```

Квалификация высшего уровня

```
/* "a" cannot be mutated by the program but can change as a result of external conditions */
const volatile int a = 5;

/* the const applies to array elements, i.e. "a[0]" cannot be mutated */
const int arr[] = { 1, 2, 3 };
```

```
/* for the lifetime of "ptr", no other pointer could point to the same "int" object */
int *restrict ptr;
```

Требования к подтипу указателя

```
/* "s1" can be mutated, but "*s1" cannot */
const char *s1 = "Hello";

/* neither "s2" (because of top-level const) nor "*s2" can be mutated */
const char *const s2 = "World";

/* "*p" may change its value as a result of external conditions, "**p" and "p" cannot */
char *volatile *p;

/* "q", "*q" and "**q" may change their values as a result of external conditions */
volatile char *volatile *volatile q;
```

Examples

Немодифицируемые (const) переменные

```
const int a = 0; /* This variable is "unmodifiable", the compiler
                    should throw an error when this variable is changed */
int b = 0; /* This variable is modifiable */

b += 10; /* Changes the value of 'b' */
a += 10; /* Throws a compiler error */
```

`const` означает только то, что мы не имеем права изменять данные. Это не означает, что ценность не может измениться за нашей спиной.

```
_Bool doIt(double const* a) {
   double rememberA = *a;
   // do something long and complicated that calls other functions

   return rememberA == *a;
}
```

Во время выполнения других вызовов `*a` может измениться, и поэтому эта функция может возвращать либо `false` либо `true` .

Предупреждение

Переменные с `const` квалификацией могут быть изменены с помощью указателей:

```
const int a = 0;
```

```
int *a_ptr = (int*)&a; /* This conversion must be explicitly done with a cast */
*a_ptr += 10;          /* This has undefined behavior */

printf("a = %d\n", a); /* May print: "a = 10" */
```

Но это ошибка, которая приводит к неопределенному поведению. Трудность здесь состоит в том, что это может вести себя так, как ожидалось, в простых примерах, как это, но затем идти не так, когда код растет.

Неустойчивые переменные

Ключевое слово `volatile` сообщает компилятору, что значение переменной может измениться в любое время в результате внешних условий, а не только в результате потока управления программой.

Компилятор не будет оптимизировать что-либо, что связано с изменчивой переменной.

```
volatile int foo; /* Different ways to declare a volatile variable */
int volatile foo;

volatile uint8_t * pReg; /* Pointers to volatile variable */
uint8_t volatile * pReg;
```

Существует две основные причины использования изменчивых переменных:

- Для взаимодействия с оборудованием, имеющим регистры ввода-вывода с отображением памяти.
- При использовании переменных, которые изменяются вне потока управления программой (например, в процедуре обслуживания прерываний)

Давайте посмотрим на этот пример:

```
int quit = false;

void main()
{
    ...
    while (!quit) {
      // Do something that does not modify the quit variable
    }
    ...
}

void interrupt_handler(void)
{
  quit = true;
}
```

Компилятору разрешено заметить, что цикл while не изменяет переменную `quit` и преобразовывает цикл в бесконечный цикл `while (true)` . Даже если переменная `quit`

задана в обработчике сигналов для `SIGINT` и `SIGTERM` , компилятор этого не знает.

Объявление `quit` as `volatile` подскажет компилятору не оптимизировать цикл, и проблема будет решена.

Такая же проблема возникает при доступе к аппаратным средствам, как мы видим в этом примере:

```
uint8_t * pReg = (uint8_t *) 0x1717;

// Wait for register to become non-zero
while (*pReg == 0) { } // Do something else
```

Поведение оптимизатора состоит в том, чтобы прочитать значение переменной один раз, нет необходимости перечитывать его, так как значение всегда будет одинаковым. Таким образом, мы заканчиваем бесконечным циклом. Чтобы заставить компилятор делать то, что мы хотим, мы модифицируем объявление:

```
uint8_t volatile * pReg = (uint8_t volatile *) 0x1717;
```

Прочитайте Тип Квалификаторы онлайн: https://riptutorial.com/ru/c/topic/2588/тип-квалификаторы

глава 56: Типы данных

замечания

- Хотя `char` должен быть 1 байт, 1 байт **не** должен быть 8 бит (часто также называемый *октетом*), хотя большинство современных компьютерных платформ определяют его как 8 бит. Число бит реализации для каждого `char` обеспечивается макросом `CHAR_BIT` , определенным в `<limits.h>` . POSIX требует, чтобы 1 байт был 8 бит.
- Целые типы с фиксированной шириной должны использоваться редко, встроенные типы C предназначены для того, чтобы быть естественными для каждой архитектуры, типы фиксированной ширины должны использоваться только в том случае, если вам явно требуется целое число определенного размера (например, для сетей).

Examples

Целочисленные типы и константы

Подписанные целые числа могут быть такого типа (`int` после `short` или `long` необязательно):

```
signed char c = 127; /* required to be 1 byte, see remarks for further information. */
signed short int si = 32767; /* required to be at least 16 bits. */
signed int i = 32767; /* required to be at least 16 bits */
signed long int li = 2147483647; /* required to be at least 32 bits. */
```

C99

```
signed long long int li = 2147483647; /* required to be at least 64 bits */
```

Каждый из этих подписанных целочисленных типов имеет неподписанную версию.

```
unsigned int i = 65535;
unsigned short = 2767;
unsigned char = 255;
```

Для всех типов, кроме `char signed` версия предполагается, если пропущена `signed` или `unsigned` часть. Тип `char` представляет собой третий тип символа, отличный от `signed char` и `unsigned char` а подпись (или нет) зависит от платформы.

Различные типы целых констант (называемые *литералами* на языке C) могут быть записаны в разных базах и различной ширине на основе их префикса или суффикса.

```
/* the following variables are initialized to the same value: */
int d = 42;   /* decimal constant (base10) */
int o = 052;  /* octal constant (base8) */
```

```
int x = 0xaf; /* hexadecimal constants (base16) */
int X = 0XAf; /* (letters 'a' through 'f' (case insensitive) represent 10 through 15) */
```

Десятичные константы всегда `signed` . Шестнадцатеричные константы начинаются с `0x` или `0X` а восьмеричные константы начинаются с `0` . Последние два являются `signed` или `unsigned` зависимости от того, соответствует ли значение подписанному типу или нет.

```
/* suffixes to describe width and signedness : */
long int i = 0x32; /* no suffix represent int, or long int */
unsigned int ui = 65535u; /* u or U represent unsigned int, or long int */
long int li = 655361; /* l or L represent long int */
```

Без суффикса константа имеет первый тип, который соответствует его значению, то есть десятичная константа, которая больше `INT_MAX` имеет тип `long` если это возможно, или `long long` противном случае.

Заголовочный файл `<limits.h>` описывает пределы целых чисел следующим образом. Их значения, определяемые реализацией, должны быть равны или больше по величине (по абсолютной величине) тем, которые показаны ниже, с тем же знаком.

макрос	Тип	Значение
`CHAR_BIT`	наименьший объект, который не является битовым полем (байтом)	8
`SCHAR_MIN`	`signed char`	-127 / - (2^7 - 1)
`SCHAR_MAX`	`signed char`	+127 / 2^7 - 1
`UCHAR_MAX`	`unsigned char`	255/2^8 - 1
`CHAR_MIN`	`char`	увидеть ниже
`CHAR_MAX`	`char`	увидеть ниже
`SHRT_MIN`	`short int`	-32767 / - (2^{15} - 1)
`SHRT_MAX`	`short int`	+32767 / 2^{15} - 1
`USHRT_MAX`	`unsigned short int`	65535/2^{16} - 1
`INT_MIN`	`int`	-32767 / - (2^{15} - 1)
`INT_MAX`	`int`	+32767 / 2^{15} - 1
`UINT_MAX`	`unsigned int`	65535/2^{16} - 1
`LONG_MIN`	`long int`	-2147483647 / - (2^{31} - 1)

макрос	Тип	Значение
LONG_MAX	long int	+2147483647 / 2^{31} - 1
ULONG_MAX	unsigned long int	4294967295/2^{32} - 1

C99

макрос	Тип	Значение
LLONG_MIN	long long int	-9223372036854775807 / - (2^{63} - 1)
LLONG_MAX	long long int	+9223372036854775807 / 2^{63} - 1
ULLONG_MAX	unsigned long long int	18446744073709551615/2^{64} - 1

Если значение объекта типа `char` sign-extends при использовании в выражении, значение `CHAR_MIN` должно быть таким же, как значение `SCHAR_MIN` а значение `CHAR_MAX` должно быть таким же, как значение `SCHAR_MAX` . Если значение объекта типа `char` не подписывается, если оно используется в выражении, значение `CHAR_MIN` должно быть 0, а значение `CHAR_MAX` должно быть таким же, как значение `UCHAR_MAX` .

C99

В стандарте C99 добавлен новый заголовок `<stdint.h>` , который содержит определения для целых чисел фиксированной ширины. См. Пример целочисленной фиксированной ширины для более подробного объяснения.

Строковые литералы

Строковый литерал в C - это последовательность символов, завершаемая буквальным нулем.

```
char* str = "hello, world"; /* string literal */

/* string literals can be used to initialize arrays */
char a1[] = "abc"; /* a1 is char[4] holding {'a','b','c','\0'} */
char a2[4] = "abc"; /* same as a1 */
char a3[3] = "abc"; /* a1 is char[3] holding {'a','b','c'}, missing the '\0' */
```

Строковые литералы **не изменяются** (и фактически могут быть помещены в постоянное запоминающее устройство, такое как .rodata). Попытка изменить их значения приводит к неопределенному поведению.

```
char* s = "foobar";
s[0] = 'F'; /* undefined behaviour */

/* it's good practice to denote string literals as such, by using `const` */
char const* s1 = "foobar";
```

```
s1[0] = 'F'; /* compiler error! */
```

Несколько строковых литералов объединяются во время компиляции, что означает, что вы можете написать такую конструкцию.

C99

```
/* only two narrow or two wide string literals may be concatenated */
char* s = "Hello, " "World";
```

C99

```
/* since C99, more than two can be concatenated */
/* concatenation is implementation defined */
char* s1 = "Hello" ", " "World";

/* common usages are concatenations of format strings */
char* fmt = "%" PRId16; /* PRId16 macro since C99 */
```

Строковые литералы, такие же, как символьные константы, поддерживают разные наборы символов.

```
/* normal string literal, of type char[] */
char* s1 = "abc";

/* wide character string literal, of type wchar_t[] */
wchar_t* s2 = L"abc";
```

C11

```
/* UTF-8 string literal, of type char[] */
char* s3 = u8"abc";

/* 16-bit wide string literal, of type char16_t[] */
char16_t* s4 = u"abc";

/* 32-bit wide string literal, of type char32_t[] */
char32_t* s5 = U"abc";
```

Типы целых чисел с фиксированной шириной (с C99)

C99

Заголовок `<stdint.h>` содержит несколько определений целочисленного типа фиксированной ширины. Эти типы являются *необязательными* и предоставляются только в том случае, если платформа имеет целочисленный тип соответствующей ширины, и если соответствующий подписанный тип имеет два дополнительных представления отрицательных значений.

См. Раздел примечаний для подсказок использования типов фиксированной ширины.

```
/* commonly used types include */
uint32_t u32 = 32; /* exactly 32-bits wide */

uint8_t u8 = 255;  /* exactly 8-bits wide */

int64_t i64 = -65  /* exactly 64 bit in two's complement representation */
```

Константы с плавающей запятой

Язык C имеет три обязательных типа с плавающей запятой, `float` , `double` и `long double` .

```
float f = 0.314f;        /* suffix f or F denotes type float */
double d = 0.314;        /* no suffix denotes double */
long double ld = 0.314l; /* suffix l or L denotes long double */

/* the different parts of a floating point definition are optional */
double x = 1.; /* valid, fractional part is optional */
double y = .1; /* valid, whole-number part is optional */

/* they can also defined in scientific notation */
double sd = 1.2e3; /* decimal fraction 1.2 is scaled by 10^3, that is 1200.0 */
```

Заголовок `<float.h>` определяет различные ограничения для операций с плавающей запятой.

Определенная реализация арифметики с плавающей точкой. Однако большинство современных платформ (arm, x86, x86_64, MIPS) используют операции с плавающей запятой IEEE 754 .

C также имеет три необязательных типа с плавающей точкой, которые получены из вышеизложенного.

Интерпретация деклараций

Отличительной синтаксической особенностью C является то, что объявления отражают использование объявленного объекта, как это было бы в нормальном выражении.

Следующий набор операторов с одинаковым приоритетом и ассоциативностью повторно используется в деклараторах, а именно:

- унарный `*` «разыменовать» оператор , который обозначает указатель;
- бинарный `[]` «подписчик на массив», который обозначает массив;
- оператор (1 + n) -ary `()` «вызов функции», который обозначает функцию;
- `()` скобки группировки, которые переопределяют приоритет и ассоциативность остальных перечисленных операторов.

Вышеуказанные три оператора имеют следующий приоритет и ассоциативность:

оператор	Относительный приоритет	Ассоциативность
`[]` (подписка на массив)	1	Слева направо
`()` (вызов функции)	1	Слева направо
`*` (разыменование)	2	Справа налево

При интерпретации объявлений нужно начинать с идентификатора наружу и применять соседние операторы в правильном порядке в соответствии с приведенной выше таблицей. Каждое приложение оператора может быть заменено следующими английскими словами:

выражение	интерпретация
`thing[X]`	массив размера `X` ...
`thing(t1, t2, t3)`	функция, принимающая `t1` , `t2` , `t3` и возвращающая ...
`*thing`	указатель на ...

Из этого следует, что начало английской интерпретации всегда начинается с идентификатора и заканчивается типом, который стоит в левой части объявления.

Примеры

```
char *names[20];
```

`[]` имеет приоритет над `*` , поэтому интерпретация: `names` - это массив размером 20 указателя на `char` .

```
char (*place)[10];
```

В случае использования скобок для переопределения приоритета сначала применяется `*` : `place` - это указатель на массив размером 10 `char` .

```
int fn(long, short);
```

Здесь нет приоритета: `fn` - это функция, выполняющая `long` , `short` и возвращающие `int` .

```
int *fn(void);
```

Сначала применяется функция `()` : `fn` - это функция, принимающая `void` и возвращающая указатель на `int` .

```
int (*fp)(void);
```

Переопределение приоритета `()` : `fp` является указателем на функцию, принимающую `void` и возвращающую `int` .

```
int arr[5][8];
```

Многомерные массивы не являются исключением из правила; операторы `[]` применяются в порядке слева направо в соответствии с ассоциативностью в таблице: `arr` представляет собой массив размером 5 массива размером 8 из `int` .

```
int **ptr;
```

Два оператора разыменования имеют одинаковый приоритет, поэтому ассоциативность вступает в силу. Операторы применяются в порядке справа налево: `ptr` является указателем на указатель на `int` .

Несколько объявлений

Запятая может использоваться как разделитель (* not *, действующий как оператор запятой), чтобы разграничить несколько деклараций внутри одного оператора. Следующий оператор содержит пять объявлений:

```
int fn(void), *ptr, (*fp)(int), arr[10][20], num;
```

Объявленные объекты в приведенном выше примере:

- `fn` : функция, принимающая `void` и возвращающая `int` ;
- `ptr` : указатель на `int` ;
- `fp` : указатель на функцию, принимающую `int` и возвращающую `int` ;
- `arr` : массив размера 10 массива размера 20 из `int` ;
- `num` : `int` .

Альтернативная интерпретация

Поскольку использование зеркал отражения используется, декларация также может быть интерпретирована в терминах операторов, которые могут быть применены к объекту, и конечного результирующего типа этого выражения. Тип, стоящий с левой стороны, является конечным результатом, который получается после применения всех операторов.

```
/*
 * Subscripting "arr" and dereferencing it yields a "char" result.
 * Particularly: *arr[5] is of type "char".
```

```
 */
char *arr[20];

/*
 * Calling "fn" yields an "int" result.
 * Particularly: fn('b') is of type "int".
 */
int fn(char);

/*
 * Dereferencing "fp" and then calling it yields an "int" result.
 * Particularly: (*fp)() is of type "int".
 */
int (*fp)(void);

/*
 * Subscripting "strings" twice and dereferencing it yields a "char" result.
 * Particularly: *strings[5][15] is of type "char"
 */
char *strings[10][20];
```

Прочитайте Типы данных онлайн: https://riptutorial.com/ru/c/topic/309/типы-данных

глава 57: Точки последовательности

замечания

Международный стандарт ISO / IEC 9899: 201x Языки программирования - C

> Доступ к изменчивому объекту, изменение объекта, изменение файла или вызов функции, выполняющей любую из этих операций, являются всеми *побочными эффектами* , которые являются изменениями состояния среды выполнения.
>
> Наличие *точки последовательности* между оценкой выражений A и B означает, что каждое вычисление значения и побочный эффект, связанные с A, секвенируются перед вычислением каждого значения и побочным эффектом, связанным с B.

Ниже приведен полный список точек последовательности из Приложения C онлайн-проекта предварительной публикации стандарта языка C на 2011 год :

> **Точки последовательности**
>
> 1 Ниже приведены точки последовательности, описанные в 5.1.2.3:
>
> - Между оценками имени функции и фактическими аргументами в вызове функции и фактическим вызовом. (6.5.2.2).
> - Между оценками первого и второго операндов следующих операторов: логическое AND `&&` (6.5.13); логический ИЛИ `||` (6.5.14); запятая `,` (6.5.17).
> - Между оценками первого операнда условного `? :` оператор и в зависимости от второго и третьего операндов (6.5.15).
> - Конец полного декларатора: деклараторы (6.7.6);
> - Между оценкой полного выражения и следующим полным выражением, которое должно быть оценено. Ниже приведены полные выражения: инициализатор, не являющийся частью составного литерала (6.7.9); выражение в выражении выражения (6.8.3); управляющее выражение оператора выбора (`if` или `switch`) (6.8.4); управляющее выражение `while` или `do` (6.8.5); каждое из (необязательных) выражений оператора `for` (6.8.5.3); (необязательное) выражение в операторе `return` (6.8.6.4).
> - Непосредственно перед возвратом функции библиотеки (7.1.4).
> - После действий, связанных с каждым форматированным спецификатором преобразования функции ввода / вывода (7.21.6, 7.29.2).
> - Непосредственно перед и сразу после каждого вызова функции сравнения, а также между любым вызовом функции сравнения и любым

перемещением объектов, переданных в качестве аргументов для этого вызова (7.22.5).

Examples

Последовательные выражения

Следующие выражения *секвенированы* :

```
a && b
a || b
a , b
a ? b : c
for ( a ; b ; c ) { ... }
```

Во всех случаях выражение `a` полностью оценивается и *все побочные эффекты применяются* до того, как оцениваются `b` или `c` . В четвертом случае будет оцениваться только один из `b` или `c` . В последнем случае `b` полностью оценивается и все побочные эффекты применяются до оценки `c` .

Во всех случаях оценка выражения `a` *секвенируется до* оценок `b` или `c` (поочередно оценки `b` и `c` *секвенируются после* оценки `a`).

Таким образом, выражения типа

```
x++ && x++
x++ ? x++ : y++
(x = f()) && x != 0
for ( x = 0; x < 10; x++ ) { ... }
y = (x++, x++);
```

имеют четко определенное поведение.

Непоследовательные выражения

C11

Следующие выражения не подвержены *влиянию* :

```
a + b;
a - b;
a * b;
a / b;
a % b;
a & b;
a | b;
```

В приведенных выше примерах выражение `a` может быть оценено до или после выражения `b` , `b` может быть оценено до `a` , или они могут даже смешиваться, если они соответствуют

нескольким инструкциям.

Аналогичное правило выполняется для вызовов функций:

```
f(a, b);
```

Здесь не только и `a` `b` являются unsequenced (то есть `,` оператор в вызове функции *не* создает точку последовательности) , но и `f` , выражение , которое определяет функцию , которая должна быть вызвана.

Побочные эффекты могут применяться сразу после оценки или отсрочки до более поздней точки.

Выражения вроде

```
x++ & x++;
f(x++, x++); /* the ',' in a function call is *not* the same as the comma operator */
x++ * x++;
a[i] = i++;
```

или же

```
x++ & x;
f(x++, x);
x++ * x;
a[i++] = i;
```

приведет к *неопределенному поведению,* потому что

- модификация объекта и любой другой доступ к нему должны быть упорядочены
- порядок оценки и порядок применения *побочных эффектов* [1] не указаны.

---

1 Любые изменения состояния среды выполнения.

Неопределенно упорядоченные выражения

Функциональные вызовы как `f(a)` всегда подразумевают точку последовательности между оценкой аргументов и обозначением (здесь `f` и `a`) и фактическим вызовом. Если два таких вызова не имеют последовательности, два вызова функции неопределенно секвенированы, то есть один выполняется перед другим, а порядок не указан.

```
unsigned counter = 0;

unsingned account(void) {
   return counter++;
}

int main(void) {
   printf("the order is %u %u\n", account(), account());
```

```
}
```

Эта неявная двукратная модификация `counter` при оценке аргументов `printf` действительна, мы просто не знаем, какой из вызовов на первом месте. Поскольку заказ не указан, он может меняться и не может зависеть. Таким образом, распечатка может быть:

    порядок равен 0 1

или же

    заказ 1 0

Аналогичное утверждение выше, без вызова промежуточной функции

```
   printf("the order is %u %u\n", counter++, counter++); // undefined behavior
```

имеет неопределенное поведение, поскольку между двумя модификациями `counter` нет точки последовательности.

Прочитайте Точки последовательности онлайн: https://riptutorial.com/ru/c/topic/1275/точки-последовательности

глава 58: указатели

Вступление

Указатель - это тип переменной, который может хранить адрес другого объекта или функции.

Синтаксис

- <Тип данных> * <Имя переменной>;
- int * ptrToInt;
- void * ptrToVoid; / * C89 + * /
- struct someStruct * ptrToStruct;
- int ** ptrToPtrToInt;
- int arr [длина]; int * ptrToFirstElem = arr; / * Для <C99 'length' должна быть константа времени компиляции, для> = C11 она может быть одной. * /
- int * arrayOfPtrsToInt [длина]; / * Для <C99 'length' должна быть константа времени компиляции, для> = C11 она может быть одной. * /

замечания

Позиция звездочки не влияет на смысл определения:

```
/* The * operator binds to right and therefore these are all equivalent. */
int *i;
int * i;
int* i;
```

Однако при определении нескольких указателей сразу требуется каждая из них:

```
int *i, *j; /* i and j are both pointers */
int* i, j;  /* i is a pointer, but j is an int not a pointer variable */
```

Также возможен массив указателей, где перед именем переменной массива задается звездочка:

```
int *foo[2]; /* foo is a array of pointers, can be accessed as *foo[0] and *foo[1] */
```

Examples

Общие ошибки

Неправильное использование указателей часто является источником ошибок, которые могут включать ошибки безопасности или сбои программ, чаще всего из-за сбоев сегментации.

Не проверять наличие сбоев

Выделение памяти не гарантируется, и вместо этого может возвращать указатель `NULL` . Используя возвращаемое значение, не проверяя, выполнено ли выделение, вызывает неопределенное поведение . Это обычно приводит к сбою, но нет никакой гарантии, что произойдет сбой, поэтому полагаться на это также может привести к проблемам.

Например, небезопасный способ:

```
struct SomeStruct *s = malloc(sizeof *s);
s->someValue = 0; /* UNSAFE, because s might be a null pointer */
```

Безопасный способ:

```
struct SomeStruct *s = malloc(sizeof *s);
if (s)
{
    s->someValue = 0; /* This is safe, we have checked that s is valid */
}
```

Использование литеральных чисел вместо sizeof при запросе памяти

Для конкретной конфигурации компилятора / машины типы имеют известный размер; однако нет никакого стандарта, который определяет, что размер данного типа (кроме `char`) будет одинаковым для всех конфигураций компилятора / машины. Если код использует 4 вместо `sizeof(int)` для распределения памяти, он может работать на исходном компьютере, но код не обязательно переносится на другие машины или компиляторы. Фиксированные размеры для типов должны быть заменены `sizeof(that_type)` или `sizeof(*var_ptr_to_that_type)` .

Не переносное распределение:

```
int *intPtr = malloc(4*1000);    /* allocating storage for 1000 int */
long *longPtr = malloc(8*1000);  /* allocating storage for 1000 long */
```

Переносное распределение:

```
int *intPtr = malloc(sizeof(int)*1000);     /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(long)*1000);  /* allocating storage for 1000 long */
```

Или, еще лучше:

```
int *intPtr = malloc(sizeof(*intPtr)*1000);     /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(*longPtr)*1000);  /* allocating storage for 1000 long */
```

Утечка памяти

Невыполнение выделенной памяти с помощью `free` ведет к созданию памяти без повторного использования, которая больше не используется программой; это называется утечкой памяти . Память утечки ресурсов памяти отходов и может привести к сбоям распределения.

Логические ошибки

Все распределения должны соответствовать одному и тому же шаблону:

1. Выделение с использованием `malloc` (или `calloc`)
2. Использование для хранения данных
3. Де-распределение с использованием `free`

Несоблюдение этого шаблона, например, использование памяти после вызова `free` (висячего указателя) или перед вызовом `malloc` (дикий указатель), вызов `free` дважды (« double free») и т. Д. Обычно вызывает ошибку сегментации и приводит к краху программы.

Эти ошибки могут быть временными и трудно отлаживать - например, освобожденная память обычно не сразу восстанавливается ОС, и поэтому оборванные указатели могут сохраняться некоторое время и, похоже, работают.

В системах, где это работает, Valgrind - бесценный инструмент для определения того, какая память просочилась и где она была изначально выделена.

Создание указателей на стек переменных

Создание указателя не продлевает срок действия указанной переменной. Например:

```
int* myFunction()
{
    int x = 10;
    return &x;
}
```

Здесь `x` имеет время *автоматического хранения* (обычно называемое распределением *стека*). Поскольку он выделяется в стеке, его время жизни остается до тех пор, пока выполняется `myFunction` ; после выхода `myFunction` переменная `x` будет уничтожена. Эта функция получает адрес `x` (с использованием `&x`) и возвращает его вызывающему,

оставляя вызывающего абонента указателем на несуществующую переменную. Попытка доступа к этой переменной затем вызовет неопределенное поведение .

Большинство компиляторов фактически не очищают кадр стека после выхода функции, поэтому разыменование возвращаемого указателя часто дает вам ожидаемые данные. Однако при вызове другой функции указываемая память может быть перезаписана, и, похоже, поврежденные данные были повреждены.

Чтобы устранить это, либо `malloc` хранилище для возвращаемой переменной, либо верните указатель на вновь созданное хранилище, либо потребуйте, чтобы действительный указатель был передан функции вместо того, чтобы возвращать ее, например:

```
#include <stdlib.h>
#include <stdio.h>

int *solution1(void)
{
    int *x = malloc(sizeof *x);
    if (x == NULL)
    {
        /* Something went wrong */
        return NULL;
    }

    *x = 10;

    return x;
}

void solution2(int *x)
{
    /* NB: calling this function with an invalid or null pointer
       causes undefined behaviour. */

    *x = 10;
}

int main(void)
{
    {
        /* Use solution1() */

        int *foo = solution1();
        if (foo == NULL)
        {
            /* Something went wrong */
            return 1;
        }

        printf("The value set by solution1() is %i\n", *foo);
        /* Will output: "The value set by solution1() is 10" */

        free(foo);    /* Tidy up */
    }

    {
        /* Use solution2() */
```

```
        int bar;
        solution2(&bar);

        printf("The value set by solution2() is %i\n", bar);
        /* Will output: "The value set by solution2() is 10" */
    }

    return 0;
}
```

Приращение / декремент и разыменование

Если вы пишете `*p++` чтобы увеличить то, что указано `p` , вы ошибаетесь.

Приращение / декрементирование сообщения выполняется до разыменования. Поэтому это выражение будет увеличивать указатель `p` и возвращать то, что было указано `p` до того, как он увеличился, не меняя его.

Вы должны написать `(*p)++` для увеличения того, на что указывает `p` .

Это правило также применяется к пост декрементированию: `*p--` будет декрементировать указатель `p` , а не то, что указано `p` .

Выделение указателя

```
int a = 1;
int *a_pointer = &a;
```

Чтобы разыменовать `a_pointer` и изменить значение a, мы используем следующую операцию

```
*a_pointer = 2;
```

Это можно проверить, используя следующие операторы печати.

```
printf("%d\n", a); /* Prints 2 */
printf("%d\n", *a_pointer); /* Also prints 2 */
```

Однако можно было бы ошибаться, чтобы разыменовать `NULL` или иначе недействительный указатель. это

```
int *p1, *p2;

p1 = (int *) 0xbad;
p2 = NULL;

*p1 = 42;
*p2 = *p1 + 1;
```

обычно является неопределенным поведением . `p1` не может быть разыменован, поскольку он указывает на адрес `0xbad` который не может быть действительным адресом. Кто знает, что там? Это может быть операционная память системы или память другой программы. Единственный временный код, подобный этому, используется во встроенной разработке, которая хранит определенную информацию по жестко закодированным адресам. `p2` не может быть разыменован, поскольку он имеет `NULL` , что является недопустимым.

Выделение указателя на структуру

Допустим, у нас есть следующая структура:

```
struct MY_STRUCT
{
    int my_int;
    float my_float;
};
```

Мы можем определить `MY_STRUCT` чтобы опустить ключевое слово `struct` поэтому нам не нужно набирать `struct MY_STRUCT` каждый раз, когда мы его используем. Это, однако, не является обязательным.

```
typedef struct MY_STRUCT MY_STRUCT;
```

Если у нас есть указатель на экземпляр этой структуры

```
MY_STRUCT *instance;
```

Если этот оператор появляется в области файлов, `instance` будет инициализирован нулевым указателем при запуске программы. Если этот оператор появляется внутри функции, его значение не определено. Переменная должна быть инициализирована, чтобы указывать на действительную переменную `MY_STRUCT` или на динамически распределенное пространство, прежде чем она может быть разыменована. Например:

```
MY_STRUCT info = { 1, 3.141593F };
MY_STRUCT *instance = &info;
```

Когда указатель действителен, мы можем разыменовать его для доступа к его членам с использованием одного из двух разных обозначений:

```
int a = (*instance).my_int;
float b = instance->my_float;
```

Хотя оба эти метода работают, лучше использовать оператор arrow `->` а не комбинацию круглых скобок, оператора разыменования `*` и точки `.` потому что его легче читать и понимать, особенно с вложенными приложениями.

Другое важное различие показано ниже:

```
MY_STRUCT copy = *instance;
copy.my_int    = 2;
```

В этом случае `copy` содержит копию содержимого `instance` . Изменение `my_int copy` не изменит его в `instance` .

```
MY_STRUCT *ref = instance;
ref->my_int    = 2;
```

В этом случае `ref` является ссылкой на `instance` . Изменение `my_int` с использованием ссылки изменит его в `instance` .

Общепринятой практикой является использование указателей для структур как параметров в функциях, а не самих структур. Использование структур в качестве параметров функции может привести к переполнению стека, если структура велика. Использование указателя на структуру использует только достаточное пространство стека для указателя, но может вызвать побочные эффекты, если функция изменяет структуру, которая передается в функцию.

Указатели функций

Указатели также могут использоваться для указания функций.

Возьмем основную функцию:

```
int my_function(int a, int b)
{
    return 2 * a + 3 * b;
}
```

Теперь давайте определим указатель на тип этой функции:

```
int (*my_pointer)(int, int);
```

Чтобы создать его, просто используйте этот шаблон:

```
return_type_of_func (*my_func_pointer)(type_arg1, type_arg2, ...)
```

Затем мы должны назначить этот указатель на функцию:

```
my_pointer = &my_function;
```

Этот указатель теперь можно использовать для вызова функции:

```
/* Calling the pointed function */
int result = (*my_pointer)(4, 2);

...

/* Using the function pointer as an argument to another function */
void another_function(int (*another_pointer)(int, int))
{
    int a = 4;
    int b = 2;
    int result = (*another_pointer)(a, b);

    printf("%d\n", result);
}
```

Хотя этот синтаксис кажется более естественным и согласованным с базовыми типами, указатели на функции присвоения и разыменования не требуют использования операторов `&` и `*` . Таким образом, следующий снипп одинаково справедлив:

```
/* Attribution without the & operator */
my_pointer = my_function;

/* Dereferencing without the * operator */
int result = my_pointer(4, 2);
```

Чтобы повысить читаемость указателей на функции, можно использовать typedefs.

```
typedef void (*Callback)(int a);

void some_function(Callback callback)
{
    int a = 4;
    callback(a);
}
```

Еще одна удобочитаемость заключается в том, что стандарт C позволяет упростить указатель функции в аргументах, подобных приведенным выше (но не в объявлении переменных), к тому, что выглядит как прототип функции; таким образом, для определений функций и деклараций можно эквивалентно использовать следующее:

```
void some_function(void callback(int))
{
    int a = 4;
    callback(a);
}
```

Смотрите также

Указатели функций

Инициализация указателей

Инициализация указателя - хороший способ избежать диких указателей. Инициализация проста и ничем не отличается от инициализации переменной.

```
#include <stddef.h>

int main()
{
    int *p1 = NULL;
    char *p2 = NULL;
    float *p3 = NULL;

         /* NULL is a macro defined in stddef.h, stdio.h, stdlib.h, and string.h */

    ...
}
```

В большинстве операционных систем, непреднамеренно используя указатель, который был инициализирован `NULL` , часто приводит к сбою программы немедленно, что позволяет легко определить причину проблемы. Использование неинициализированного указателя часто может вызывать затруднительные диагностические ошибки.

Внимание:

Результат разыменования указателя `NULL` не определен, поэтому он *не обязательно приведет к сбою,* даже если это естественное поведение операционной системы, в которой работает программа. Оптимизация компилятора может привести к сбою в сбое, вызвав возникновение сбоя до или после точки в исходном коде, в котором произошла ошибка разыменования нулевого указателя, или привести к неожиданному удалению частей кода, которые содержат разыменование нулевого указателя, из программы. Отладочные сборки обычно не будут демонстрировать это поведение, но это не гарантируется языковым стандартом. Также допускается другое неожиданное и / или нежелательное поведение.

Поскольку `NULL` никогда не указывает на переменную, на выделенную память или на функцию, ее безопасно использовать в качестве защитного значения.

Внимание:

Обычно `NULL` определяется как `(void *)0` . Но это не означает, что назначенный адрес памяти равен `0x0` . Для получения дополнительной информации обратитесь к C-faq для NULL-указателей

Обратите внимание, что вы также можете инициализировать указатели, чтобы они содержали значения, отличные от NULL.

```
int i1;

int main()
```

```
{
   int *p1 = &i1;
   const char *p2 = "A constant string to point to";
   float *p3 = malloc(10 * sizeof(float));
}
```

Адрес-оператора (&)

Для любого объекта (т. Е. Переменной, массива, объединения, структуры, указателя или функции) оператор унарного адреса может использоваться для доступа к адресу этого объекта.

Предположим, что

```
int i = 1;
int *p = NULL;
```

Итак, утверждение `p = &i;` , копирует адрес переменной `i` в указатель `p` .

Это выражается как `p` ***points to*** `i` .

`printf("%d\n", *p);` prints 1, значение `i` .

Арифметика указателей

См. Здесь: Арифметика указателей

void * указатели в качестве аргументов и возвращаемые значения для стандартных функций

K & P

`void*` - это тип catch для типов указателей на типы объектов. Примером этого является использование функции `malloc` , которая объявляется как

```
void* malloc(size_t);
```

Тип возвращаемого указателя в void означает, что можно присвоить возвращаемое значение из `malloc` указателю на любой другой тип объекта:

```
int* vector = malloc(10 * sizeof *vector);
```

Обычно считается хорошей практикой, чтобы явным образом *не* вводить значения в указатели void и из них. В конкретном случае `malloc()` это связано с тем, что при явном приведении компилятор может в противном случае предположить, но не предупреждать о некорректном возвращаемом типе для `malloc()` , если вы забыли включить `stdlib.h` . Это

также случай использования правильного поведения указателей пустот, чтобы лучше соответствовать принципу DRY (не повторяйте себя); сравните это со следующим, в котором следующий код содержит несколько ненужных дополнительных мест, где опечатка может вызвать проблемы:

```
int* vector = (int*)malloc(10 * sizeof int*);
```

Аналогично, такие функции, как

```
void* memcpy(void *restrict target, void const *restrict source, size_t size);
```

имеют свои аргументы, указанные как `void *` потому что адрес любого объекта, независимо от типа, может быть передан. Здесь также вызов не должен использовать листинг

```
unsigned char buffer[sizeof(int)];
int b = 67;
memcpy(buffer, &b, sizeof buffer);
```

Концентраторы

Одиночные указатели

- Указатель на `int`

  Указатель может указывать на разные целые числа, а `int` может быть изменен с помощью указателя. Этот образец кода присваивает b для указания на `int b` затем изменяет значение `b` на `100` .

  ```
  int b;
  int* p;
  p = &b;    /* OK */
  *p = 100;  /* OK */
  ```

- Указатель на `const int`

  Указатель может указывать на разные целые числа, но значение `int` не может быть изменено с помощью указателя.

  ```
  int b;
  const int* p;
  p = &b;    /* OK */
  *p = 100;  /* Compiler Error */
  ```

- `const` указатель на `int`

  Указатель может указывать только на один `int` но значение `int` может быть изменено с помощью указателя.

```
int a, b;
int* const p = &b; /* OK as initialisation, no assignment */
*p = 100;  /* OK */
p = &a;    /* Compiler Error */
```

- `const` указатель на `const int`

  Указатель может указывать только на один `int` и `int` не может быть изменен с помощью указателя.

```
int a, b;
const int* const p = &b; /* OK as initialisation, no assignment */
p = &a;   /* Compiler Error */
*p = 100; /* Compiler Error */
```

Указатель на указатель

- Указатель на указатель на `int`

  Этот код присваивает адрес `p1` двойному указателю `p` (который затем указывает на `int* p1` (который указывает на `int`)).

  Затем `p1` на точку `int a` . Затем изменяется значение a равным 100.

```
void f1(void)
{
  int a, b;
  int *p1;
  int **p;
  p1 = &b; /* OK */
  p = &p1; /* OK */
  *p = &a; /* OK */
  **p = 100; /* OK */
}
```

- Указатель на указатель на `const int`

```
 void f2(void)
{
  int b;
  const int *p1;
  const int **p;
  p = &p1; /* OK */
  *p = &b; /* OK */
  **p = 100; /* error: assignment of read-only location ‘**p’ */
}
```

- Указатель на указатель `const` на `int`

```
void f3(void)
{
```

```
  int b;
  int *p1;
  int * const *p;
  p = &p1; /* OK */
  *p = &b; /* error: assignment of read-only location ‘*p’ */
  **p = 100; /* OK */
}
```

- `const` указатель на указатель на `int`

```
void f4(void)
{
  int b;
  int *p1;
  int ** const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable ‘p’ */
  *p = &b; /* OK */
  **p = 100; /* OK */
}
```

- Указатель на `const` указатель на `const int`

```
void f5(void)
{
  int b;
  const int *p1;
  const int * const *p;
  p = &p1; /* OK */
  *p = &b; /* error: assignment of read-only location ‘*p’ */
  **p = 100; /* error: assignment of read-only location ‘**p’ */
}
```

- `const` указатель на указатель на `const int`

```
void f6(void)
{
  int b;
  const int *p1;
  const int ** const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable ‘p’ */
  *p = &b; /* OK */
  **p = 100; /* error: assignment of read-only location ‘**p’ */
}
```

- `const` указатель на `const` указатель на `int`

```
void f7(void)
{
  int b;
  int *p1;
  int * const * const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable ‘p’  */
  *p = &b; /* error: assignment of read-only location ‘*p’ */
  **p = 100; /* OK */
}
```

Одинаковая звездочка, разные значения

посылка

Самая запутанная вещь, связанная с синтаксисом указателя на C и C ++, состоит в том, что на самом деле существуют два разных значения, которые применяются, когда символ указателя, звездочка (`*`) используется с переменной.

пример

Во-первых, вы используете `*` для **объявления** переменной указателя.

```
int i = 5;
/* 'p' is a pointer to an integer, initialized as NULL */
int *p = NULL;
/* '&i' evaluates into address of 'i', which then assigned to 'p' */
p = &i;
/* 'p' is now holding the address of 'i' */
```

Когда вы не объявляете (или не умножаете), `*` используется для **разыменования** переменной указателя:

```
*p = 123;
/* 'p' was pointing to 'i', so this changes value of 'i' to 123 */
```

Если вы хотите, чтобы существующая переменная указателя удерживала адрес другой переменной, вы **не** используете `*` , но делаете это следующим образом:

```
p = &another_variable;
```

Обычная путаница среди новичков C-программирования возникает, когда они объявляют и инициализируют переменную указателя одновременно.

```
int *p = &i;
```

Поскольку `int i = 5;` и `int i; i = 5;` дают тот же результат, некоторые из них могли бы считать `int *p = &i;` и `int *p; *p = &i;` дайте тот же результат тоже. Дело в том, что нет, `int *p; *p = &i;` будет пытаться уважать **неинициализированный** указатель, который приведет к UB. Никогда не используйте `*` если вы не декларируете и не разыскиваете указатель.

Заключение

Звездочка (`*`) имеет два различных значения внутри C относительно указателей, в зависимости от того, где они используются. При использовании в **объявлении переменной** значение в правой части равенства равно ***значению указателя*** на ***адрес*** в памяти. При использовании с уже **объявленной переменной** , звездочка будет ***разыменовывать*** значение указателя, следуя за ним в указанное место в памяти и позволяя присвоить или получить значение, которое будет там сохранено.

навынос

Важно помнить о своих P и Q, так сказать, при работе с указателями. Помните, когда вы используете звездочку, и что это означает, когда вы ее используете. Осмотр этой крошечной детали может привести к ошибкам и / или неопределенному поведению, с которыми вы действительно не хотите иметь дело.

Указатель на указатель

В C указатель может ссылаться на другой указатель.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &pA;
  int*** pppA = &ppA;

  printf("%d", ***pppA); /* prints 42 */

  return EXIT_SUCCESS;
}
```

Но ссылки и ссылки напрямую не разрешены.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &&A; /* Compilation error here! */
  int*** pppA = &&&A;  /* Compilation error here! */

  ...
```

Вступление

Указатель объявляется так же, как любая другая переменная, за исключением того, что звездочка (`*`) помещается между типом и именем переменной, чтобы обозначить это указатель.

```
int *pointer; /* inside a function, pointer is uninitialized and doesn't point to any valid
object yet */
```

Чтобы объявить две переменные указателя того же типа, в том же объявлении, используйте символ звездочки перед каждым идентификатором. Например,

```
int *iptr1, *iptr2;
int *iptr3,  iptr4;  /* iptr3 is a pointer variable, whereas iptr4 is misnamed and is an int
*/
```

Оператор адреса или ссылки, обозначенный амперсандом (`&`), дает адрес данной переменной, который может быть помещен в указатель соответствующего типа.

```
int value = 1;
pointer = &value;
```

Оператор косвенности или разыменования, обозначенный звездочкой (`*`), получает содержимое объекта, на который указывает указатель.

```
printf("Value of pointed to integer: %d\n", *pointer);
/* Value of pointed to integer: 1 */
```

Если указатель указывает на структуру или тип объединения, вы можете разыменовать его и получить доступ к его членам напрямую с помощью оператора `->` :

```
SomeStruct *s = &someObject;
s->someMember = 5; /* Equivalent to (*s).someMember = 5 */
```

В C указатель представляет собой отдельный тип значения, который может быть переназначен и в противном случае рассматривается как переменная в своем собственном праве. Например, следующий пример печатает значение самого указателя (переменной).

```
printf("Value of the pointer itself: %p\n", (void *)pointer);
/* Value of the pointer itself: 0x7ffcd41b06e4 */
/* This address will be different each time the program is executed */
```

Поскольку указатель является изменяемой переменной, он может не указывать на действительный объект, либо путем установки значения null

```
pointer = 0;     /* or alternatively */
pointer = NULL;
```

или просто путем размещения произвольного битового шаблона, который не является допустимым адресом. Последнее очень плохое, потому что он не может быть протестирован до того, как указатель будет разыменован, есть только тест для случая, когда указатель имеет значение null:

```
if (!pointer) exit(EXIT_FAILURE);
```

Указатель может быть разыменован только если он указывает на *действительный* объект, иначе поведение не определено. Многие современные реализации могут помочь вам, подняв некоторую ошибку, такую как ошибка сегментации и прекратить выполнение, но другие могут просто оставить вашу программу в недопустимом состоянии.

Значение, возвращаемое оператором разыменования, является изменчивым псевдонимом исходной переменной, поэтому его можно изменить, изменив исходную переменную.

```
*pointer += 1;
printf("Value of pointed to variable after change: %d\n", *pointer);
/* Value of pointed to variable after change: 2 */
```

Указатели также переустанавливаются. Это означает, что указатель, указывающий на объект, может впоследствии использоваться для указания на другой объект того же типа.

```
int value2 = 10;
pointer = &value2;
printf("Value from pointer: %d\n", *pointer);
/* Value from pointer: 10 */
```

Как и любая другая переменная, указатели имеют определенный тип. Например, вы не можете назначить адрес `short int` указателю на `long int` . Такое поведение упоминается как тип punning и запрещено в C, хотя есть несколько исключений.

Хотя указатель должен иметь определенный тип, память, выделенная для каждого типа указателя, равна памяти, используемой средой для хранения адресов, а не размеру указанного типа.

```
#include <stdio.h>

int main(void) {
    printf("Size of int pointer: %zu\n", sizeof (int*));      /* size 4 bytes */
    printf("Size of int variable: %zu\n", sizeof (int));      /* size 4 bytes */
    printf("Size of char pointer: %zu\n", sizeof (char*));    /* size 4 bytes */
    printf("Size of char variable: %zu\n", sizeof (char));    /* size 1 bytes */
    printf("Size of short pointer: %zu\n", sizeof (short*));  /* size 4 bytes */
    printf("Size of short variable: %zu\n", sizeof (short));  /* size 2 bytes */
    return 0;
}
```

(NB: если вы используете Microsoft Visual Studio, которая не поддерживает стандарты C99 или C11, вы должны использовать `%Iu` [1] *вместо* `%zu` *в приведенном выше примере.)*

Обратите внимание, что приведенные выше результаты могут варьироваться в зависимости от окружения и окружающей среды, но все среды отображают одинаковые размеры для разных типов указателей.

Экстракт на основе информации из Кардиффского университета C Указатели Введение

Указатели и массивы

Указатели и массивы тесно связаны в C. Массивы в C всегда хранятся в смежных местах в памяти. Арифметика указателя всегда масштабируется по размеру указанного пункта. Поэтому, если у нас есть массив из трех двойников и указатель на базу, `*ptr` относится к первому двойнику `*(ptr + 1)` ко второму, `*(ptr + 2)` к третьему. Более удобная нотация - использовать нотацию массива `[]` .

```
double point[3] = {0.0, 1.0, 2.0};
double *ptr = point;

/* prints x 0.0, y 1.0 z 2.0 */
printf("x %f y %f z %f\n", ptr[0], ptr[1], ptr[2]);
```

Таким образом, ptr и имя массива являются взаимозаменяемыми. Это правило также означает, что массив переходит в указатель при передаче в подпрограмму.

```
double point[3] = {0.0, 1.0, 2.0};

printf("length of point is %s\n", length(point));

/* get the distance of a 3D point from the origin */
double length(double *pt)
{
   return sqrt(pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2])
}
```

Указатель может указывать на любой элемент в массиве или на элемент за последним элементом. Однако ошибка заключается в установке указателя на любое другое значение, включая элемент перед массивом. (Причина в том, что на сегментированных архитектурах адрес перед первым элементом может пересекать границу сегмента, компилятор гарантирует, что это не произойдет для последнего элемента плюс один).

Сноска 1: информация о формате Microsoft может быть найдена с помощью `printf()` и синтаксиса спецификации формата .

Полиморфное поведение с указателями void

Стандартная библиотечная функция `qsort()` является хорошим примером того, как можно использовать указатели void, чтобы одна функция работала на множестве разных типов.

```
void qsort (
    void *base,                                 /* Array to be sorted */
    size_t num,                                 /* Number of elements in array */
    size_t size,                                /* Size in bytes of each element */
```

```
    int (*compar)(const void *, const void *)); /* Comparison function for two elements */
```

Массив, который нужно отсортировать, передается как указатель на void, поэтому можно использовать массив любого типа элемента. Следующие два аргумента `qsort()` сколько элементов он должен ожидать в массиве, и насколько велики в байтах каждый элемент.

Последний аргумент - это указатель на функцию сравнения, которая сама принимает два указателя void. Предоставляя вызывающей стороне эту функцию, `qsort()` может эффективно сортировать элементы любого типа.

Вот пример такой функции сравнения, для сравнения поплавков. Обратите внимание, что любая функция сравнения, переданная в `qsort()` должна иметь подпись этого типа. Способ, которым он сделан полиморфным, заключается в том, чтобы отличить аргументы указателя void от указателей типа элемента, который мы хотим сравнить.

```
int compare_floats(const void *a, const void *b)
{
    float fa = *((float *)a);
    float fb = *((float *)b);
    if (fa < fb)
        return -1;
    if (fa > fb)
        return 1;
    return 0;
}
```

Поскольку мы знаем, что qsort будет использовать эту функцию для сравнения float, мы передаем аргументы указателя void обратно в указатели float перед их разыменованием.

Теперь использование полиморфной функции qsort в массиве «array» с длиной «len» очень просто:

```
qsort(array, len, sizeof(array[0]), compare_floats);
```

Прочитайте указатели онлайн: https://riptutorial.com/ru/c/topic/1108/указатели

глава 59: Указатели функций

Вступление

Указатели функций - это указатели, указывающие на функции вместо типов данных. Они могут использоваться для обеспечения изменчивости функции, которая должна быть вызвана во время выполнения.

Синтаксис

- returnType (* name) (параметры)
- typedef returnType (* name) (параметры)
- typedef returnType Name (параметры);
  Имя * имя;
- typedef returnType Name (параметры);
  typedef Имя * NamePtr;

Examples

Назначение указателя функции

```
#include <stdio.h>

/* increment: take number, increment it by one, and return it */
int increment(int i)
{
    printf("increment %d by 1\n", i);
    return i + 1;
}

/* decrement: take number, decrement it by one, and return it */
int decrement(int i)
{
    printf("decrement %d by 1\n", i);
    return i - 1;
}

int main(void)
{
    int num = 0;          /* declare number to increment */
    int (*fp)(int);       /* declare a function pointer */

    fp = &increment;      /* set function pointer to increment function */
    num = (*fp)(num);     /* increment num */
    num = (*fp)(num);     /* increment num a second time */
```

```
    fp = &decrement;      /* set function pointer to decrement function */
    num = (*fp)(num);     /* decrement num */
    printf("num is now: %d\n", num);
    return 0;
}
```

Возвращаемые указатели функций из функции

```
#include <stdio.h>

enum Op
{
  ADD = '+',
  SUB = '-',
};


/* add: add a and b, return result */
int add(int a, int b)
{
    return a + b;
}

/* sub: subtract b from a, return result */
int sub(int a, int b)
{
    return a - b;
}

/* getmath: return the appropriate math function */
int (*getmath(enum Op op))(int,int)
{
    switch (op)
    {
        case ADD:
            return &add;
        case SUB:
            return &sub;
        default:
            return NULL;
    }
}

int main(void)
{
    int a, b, c;
    int (*fp)(int,int);

    fp = getmath(ADD);

    a = 1, b = 2;
    c = (*fp)(a, b);
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

Лучшие практики

Использование typedef

Было бы удобно использовать `typedef` вместо объявления указателя функции каждый раз вручную.

Синтаксис объявления `typedef` для указателя функции:

```
typedef returnType (*name)(parameters);
```

Пример:

Положим, что у нас есть функция, `sort` , которая ожидает, что указатель функции на функцию `compare` , что:

> compare - Функция сравнения для двух элементов, которая должна быть предоставлена функции сортировки.
>
> «сравнение», как ожидается, возвращает 0, если два элемента считаются равными, положительное значение, если первый переданный элемент является «большим» в некотором смысле, чем последний элемент, и в противном случае функция возвращает отрицательное значение (это означает, что первый элемент равен «меньше», чем последний).

Без `typedef` мы `typedef` бы указатель функции как аргумент функции следующим образом:

```
void sort(int (*compare)(const void *elem1, const void *elem2)) {
    /* inside of this block, the function is named "compare" */
}
```

С `typedef` мы напишем:

```
typedef int (*compare_func)(const void *, const void *);
```

и тогда мы могли бы изменить подпись функции `sort` :

```
void sort(compare_func func) {
    /* In this block the function is named "func" */
}
```

оба определения `sort` будут принимать любую функцию вида

```
int compare(const void *arg1, const void *arg2) {
    /* Note that the variable names do not have to be "elem1" and "elem2" */
}
```

Указатели функций - это единственное место, где вы должны указать свойство указателя типа, например, не пытайтесь определить такие типы, как `typedef struct something_struct *something_type` . Это относится даже для структуры с членами , которые не предполагается доступ непосредственно API абонентов, например, stdio.h `FILE` типа (который , как вы уже заметите это не указатель).

Использование контекстных указателей.

Указатель функции должен почти всегда принимать предоставленный пользователем void * в качестве указателя контекста.

пример

```
/* function minimiser, details unimportant */
double findminimum( double (*fptr)(double x, double y, void *ctx), void *ctx)
{
    ...
    /* repeatedly make calls like this */
    temp = (*fptr)(testx, testy, ctx);
}

/* the function we are minimising, sums two cubics */
double *cubics(double x, double y, void *ctx)
{
    double *coeffsx = ctx;
    double *coeffsy = coeffx + 4;

    return coeffsx[0] * x * x * x + coeffsx[1] * x * x + coeffsx[2] * x + coeffsx[3] +
           coeffsy[0] * y * y * y + coeffsy[1] * y * y + coeffsy[2] * y + coeffsy[3];

}

void caller()
{
    /* context, the coefficients of the cubics */
    double coeffs[8] = {1, 2, 3, 4, 5, 6, 7, 8};
    double min;

    min = findminimum(cubics, coeffs);
}
```

Использование указателя контекста означает, что дополнительные параметры не должны быть жестко закодированы в функции, на которую указывает или требуют использования глобальных переменных.

Функция библиотеки `qsort()` не соответствует этому правилу, и часто можно избегать контекста для тривиальных функций сравнения. Но для чего-то более сложного, указатель контекста становится существенным.

Смотрите также

Указатели функций

Вступление

Точно так же, как `char` и `int` , функция является фундаментальной особенностью C. Как таковая, вы можете объявить указатель на один: это означает, что вы можете передать *какую функцию вызывать* другой функции, чтобы помочь ей выполнить свою работу. Например, если у вас есть функция `graph()` которая отображает график, вы можете передать *какую функцию графику* в `graph()` .

```
// A couple of external definitions to make the example clearer
extern unsigned int screenWidth;
extern void plotXY(double x, double y);

// The graph() function.
// Pass in the bounds: the minimum and maximum X and Y that should be plotted.
// Also pass in the actual function to plot.
void graph(double minX, double minY,
           double maxX, double maxY,
           ???? *fn) {            // See below for syntax

    double stepX = (maxX - minX) / screenWidth;
    for (double x=minX; x<maxX; x+=stepX) {

        double y = fn(x);         // Get y for this x by calling passed-in fn()

        if (minY<=y && y<maxY) {
            plotXY(x, y);         // Plot calculated point
        } // if
    } for
} // graph(minX, minY, maxX, maxY, fn)
```

использование

Таким образом, вышеприведенный код будет отображать любую функцию, которую вы передали в нее, - пока эта функция удовлетворяет определенным критериям: а именно, что вы передаете `double` код и получаете `double` . Существует много таких функций: `sin()` , `cos()` , `tan()` , `exp()` и т. Д., Но их много, например, `graph()` !

Синтаксис

Итак, как вы определяете, какие функции вы можете передать в `graph()` а какие из них вы не можете? Обычным способом является использование синтаксиса, который может быть нелегко прочитать или понять:

```
double (*fn)(double); // fn is a pointer-to-function that takes a double and returns one
```

Вышеупомянутая проблема состоит в том, что одновременно можно определить две вещи: структуру функции и тот факт, что это указатель. Итак, разделите два определения! Но, используя `typedef` , может быть достигнут лучший синтаксис (легче читать и понимать).

Мнемоника для написания указателей функций

Все функции C находятся в указателях актуальности на месте в памяти программы, где существует некоторый код. Основное использование указателя функции заключается в предоставлении «обратного вызова» для других функций (или для имитации классов и объектов).

Синтаксис функции, определенный ниже на этой странице:

returnType (* name) (параметры)

Мнемоника для написания определения указателя функции - это следующая процедура:

1. Начните с написания объявления нормальной функции: `returnType name(parameters)`
2. Оберните имя функции с помощью синтаксиса указателя: `returnType (*name)(parameters)`

основы

Так же, как вы можете иметь указатель на **int** , **char** , **float** , **array / string** , **struct** и т. Д. - вы можете иметь указатель на функцию.

Объявление указателя принимает *возвращаемое значение функции* , *имя функции* и *тип аргументов / параметров, которые она получает* .

Предположим, что вы объявили и инициализировали следующую функцию:

```
int addInt(int n, int m){
    return n+m;
}
```

Вы можете объявить и инициализировать указатель на эту функцию:

```
int (*functionPtrAdd)(int, int) = addInt; // or &addInt - the & is optional
```

Если у вас есть функция void, она может выглядеть так:

```
void Print(void){
    printf("look ma' - no hands, only pointers!\n");
}
```

Тогда объявление указателя на него будет:

```
void (*functionPtrPrint)(void) = Print;
```

Доступ к самой функции потребует разыменования указателя:

```
sum = (*functionPtrAdd)(2, 3); //will assign 5 to sum
(*functionPtrPrint)(); //will print the text in Print function
```

Как видно из более сложных примеров в этом документе, объявление указателя на функцию может стать беспорядочным, если функции передано более нескольких параметров. Если у вас есть несколько указателей на функции, которые имеют идентичную структуру (тот же тип возвращаемого значения и параметры одного и того же типа), лучше всего использовать команду **typedef,** чтобы сэкономить вам некоторое написание и сделать код более понятным:

```
typedef int (*ptrInt)(int, int);

int Add(int i, int j){
    return i+j;
}

int Multiply(int i, int j){
    return i*j;
}

int main()
{
    ptrInt ptr1 = Add;
    ptrInt ptr2 = Multiply;

    printf("%d\n", (*ptr1)(2,3)); //will print 5
    printf("%d\n", (*ptr2)(2,3)); //will print 6
    return 0;
}
```

Вы также можете создать **массив указателей функций** . Если все указатели имеют одну и ту же «структуру»:

```
int (*array[2]) (int x, int y); // can hold 2 function pointers
array[0] = Add;
array[1] = Multiply;
```

Вы можете узнать больше здесь и здесь .

Также возможно определить массив указателей функций разных типов, хотя для этого требуется кастинг, когда вы хотите получить доступ к определенной функции. Вы можете узнать больше здесь .

Прочитайте Указатели функций онлайн: https://riptutorial.com/ru/c/topic/250/указатели-функций

глава 60: Управление памятью

Вступление

Для управления динамически распределенной памятью стандартная библиотека C предоставляет функции `malloc()` , `calloc()` , `realloc()` и `free()` . В C99 и более поздних версиях также есть `aligned_alloc()` . Некоторые системы также предоставляют `alloca()` .

Синтаксис

- void * aligned_alloc (size_t alignment, size_t size); / * Только с C11 * /
- void * calloc (size_t nelements, size_t size);
- void free (void * ptr);
- void * malloc (size_t size);
- void * realloc (void * ptr, size_t size);
- void * alloca (size_t size); / * от alloca.h, не стандартный, не портативный, опасный. * /

параметры

название	описание
размер (`malloc` , `realloc` и `aligned_alloc`)	общий размер памяти в байтах. Для `aligned_alloc` размер должен быть целым кратным выравниванию.
размер (`calloc`)	размер каждого элемента
nelements	количество элементов
PTR	указатель на выделенную память, ранее возвращенную `malloc` , `calloc` , `realloc` или `aligned_alloc`
выравнивание	выравнивание выделенной памяти

замечания

C11

Обратите внимание: `aligned_alloc()` определен только для C11 или более поздних `aligned_alloc()` .

Системы, такие как основанные на POSIX, предоставляют другие способы выделения выровненной памяти (например, `posix_memalign()`), а также другие параметры управления

памятью (например, `mmap()`).

Examples

Освобождение памяти

Можно освободить динамически выделенную память, вызвав free () .

```
int *p = malloc(10 * sizeof *p); /* allocation of memory */
if (p == NULL)
{
    perror("malloc failed");
    return -1;
}

free(p); /* release of memory */
/* note that after free(p), even using the *value* of the pointer p
   has undefined behavior, until a new value is stored into it. */

/* reusing/re-purposing the pointer itself */
int i = 42;
p = &i; /* This is valid, has defined behaviour */
```

Память, на которую указывает `p` , восстанавливается (либо посредством реализации libc, либо базовой ОС) после вызова `free()` , поэтому доступ к этому свободному блоку памяти через `p` приведет к неопределенному поведению . Указатели, которые ссылаются на освобожденные элементы памяти, обычно называются оборванными указателями и представляют угрозу безопасности. Кроме того, стандарт C утверждает, что даже доступ к значению висячего указателя имеет неопределенное поведение. Обратите внимание, что сам указатель `p` может быть переназначен, как показано выше.

Обратите внимание, что вы можете звонить только `free()` по указателям, которые были возвращены непосредственно из функций `malloc()` , `calloc()` , `realloc()` и `aligned_alloc()` или где документация сообщает вам, что память была выделена таким образом (функции такие как `strdup ()` являются заметными примерами). Освобождая указатель, который есть,

- полученные с помощью оператора `&` на переменной, или
- в середине выделенного блока,

запрещен. Такая ошибка обычно не будет диагностирована вашим компилятором, но приведет к выполнению программы в неопределенном состоянии.

Существуют две распространенные стратегии предотвращения таких случаев неопределенного поведения.

Первое и предпочтительное - просто: `p` перестает существовать, когда он больше не нужен, например:

```
if (something_is_needed())
{

    int *p = malloc(10 * sizeof *p);
    if (p == NULL)
    {
        perror("malloc failed");
        return -1;
    }

    /* do whatever is needed with p */

    free(p);
}
```

Вызывая `free()` непосредственно перед концом содержащего блока (то есть `}`), `p` сама перестает существовать. Компилятор даст ошибку компиляции при любой попытке использовать `p` после этого.

Второй подход заключается в том, чтобы также сделать недействительным сам указатель после освобождения памяти, на которую он указывает:

```
free(p);
p = NULL;     // you may also use 0 instead of NULL
```

Аргументы для этого подхода:

- На многих платформах попытка разыменования нулевого указателя вызовет мгновенный сбой: ошибка сегментации. Здесь мы получаем как минимум трассировку стека, указывающую на переменную, которая была использована после освобождения.

  Не устанавливая указатель на `NULL` мы имеем свисающий указатель. Вероятно, программа все равно будет разбиваться, но позже, потому что память, на которую указывает указатель, будет бесшумно повреждена. Такие ошибки трудно отследить, поскольку они могут привести к стеку вызовов, полностью не связанному с исходной проблемой.

  Этот подход, следовательно, следует за неудачной концепцией .

- Безопасно освобождать нулевой указатель. В стандарте C указано, что `free(NULL)` не действует:

  > Свободная функция заставляет пространство, на которое указывает ptr, освобождается, то есть становится доступным для дальнейшего выделения. Если ptr является нулевым указателем, никаких действий не происходит. В противном случае, если аргумент не соответствует указателю, ранее возвращенному функцией `calloc` , `malloc` или `realloc` , или если пространство было освобождено вызовом `free` или `realloc` , поведение

не определено.

- Иногда первый подход не может быть использован (например, память выделяется одной функцией и значительно позже освобождается от совершенно другой функции)

Выделение памяти

Стандартное распределение

Функции распределения динамической памяти C определены в заголовке `<stdlib.h>` . Если вы хотите динамически выделять пространство памяти для объекта, можно использовать следующий код:

```
int *p = malloc(10 * sizeof *p);
if (p == NULL)
{
    perror("malloc() failed");
    return -1;
}
```

Это вычисляет количество байтов, которое занимает десять `int` s в памяти, затем запрашивает, что много байтов из `malloc` и присваивает результат (то есть начальный адрес блока памяти, который только что был создан с помощью `malloc`), указателю с именем `p` .

Рекомендуется использовать `sizeof` для вычисления объема запрашиваемой памяти, так как результат `sizeof` определяется реализацией (за исключением *типов символов* , которые являются `char` , `signed char` и `unsigned char` , для которых `sizeof` определен всегда, чтобы дать `1`).

Поскольку `malloc` не может обслуживать запрос, он может вернуть нулевой указатель. Для этого важно проверить это, чтобы предотвратить последующие попытки разыменовать нулевой указатель.

Память, динамически распределенная с помощью `malloc()` может быть изменена с использованием `realloc()` или, если она больше не понадобится, освобождается с помощью `free()` .

Альтернативно, объявляя `int array[10];` будет выделять один и тот же объем памяти. Однако, если он объявлен внутри функции без ключевого слова `static` , он будет использоваться только внутри функции, в которой он объявлен, и функций, которые он вызывает (поскольку массив будет выделен в стеке, и пространство будет выпущено для повторного использования, когда функция возвращает). В качестве альтернативы, если она определена со `static` внутри функции или если она определена вне любой функции, то ее время жизни является временем жизни программы. Указатели также могут быть

возвращены из функции, однако функция в C не может вернуть массив.

Нулевая память

Память, возвращаемая `malloc` не может быть инициализирована до разумного значения, и следует позаботиться об обнулении памяти с помощью `memset` или немедленно скопировать в нее подходящее значение. В качестве альтернативы `calloc` возвращает блок требуемого размера, где все биты инициализируются до `0` . Это не должно быть таким же, как представление нулевой точки с плавающей точкой или константы нулевого указателя.

```
int *p = calloc(10, sizeof *p);
if (p == NULL)
{
    perror("calloc() failed");
    return -1;
}
```

Замечание о `calloc` : Большинство (обычно используемых) реализаций оптимизируют `calloc()` для производительности, поэтому он будет быстрее, чем вызов `malloc()` , затем `memset()` , хотя сетевой эффект идентичен.

Выровненная память

C11

C11 представила новую функцию `aligned_alloc()` которая выделяет пространство с заданным выравниванием. Он может использоваться, если выделенная память необходима для выравнивания на определенных границах, которые не могут быть удовлетворены `malloc()` или `calloc()` . Функции `malloc()` и `calloc()` выделяют память, которая соответствующим образом выровнена для *любого* типа объекта (т.е. выравнивание равно `alignof(max_align_t)`). Но с `aligned_alloc()` можно запросить большее выравнивание.

```
/* Allocates 1024 bytes with 256 bytes alignment. */
char *ptr = aligned_alloc(256, 1024);
if (ptr) {
    perror("aligned_alloc()");
    return -1;
}
free(ptr);
```

Стандарт C11 накладывает два ограничения: 1) запрошенный *размер* (второй аргумент) должен быть целым кратным *выравниванию* (первый аргумент) и 2) значение *выравнивания* должно быть действительным выравниванием, поддерживаемым реализацией. Несоблюдение любого из них приводит к неопределенному поведению .

Перераспределение памяти

Возможно, вам придется расширять или сокращать пространство для хранения указателя после того, как вы выделили ему память. Функция `void *realloc(void *ptr, size_t size)` освобождает старый объект, на который указывает `ptr` и возвращает указатель на объект, размер которого задан по `size` . `ptr` - это указатель на блок памяти, ранее выделенный с помощью `malloc` , `calloc` или `realloc` (или нулевой указатель), который должен быть перераспределен. Сохраняется максимально возможное содержимое исходной памяти. Если новый размер больше, любая дополнительная память за пределами старого размера не инициализируется. Если новый размер короче, содержимое усаженной части теряется. Если `ptr` равно NULL, выделяется новый блок, и указатель на него возвращается функцией.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p = malloc(10 * sizeof *p);
    if (NULL == p)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }

    p[0] = 42;
    p[9] = 15;

    /* Reallocate array to a larger size, storing the result into a
     * temporary pointer in case realloc() fails. */
    {
        int *temporary = realloc(p, 1000000 * sizeof *temporary);

        /* realloc() failed, the original allocation was not free'd yet. */
        if (NULL == temporary)
        {
            perror("realloc() failed");
            free(p); /* Clean up. */
            return EXIT_FAILURE;
        }

        p = temporary;
    }

    /* From here on, array can be used with the new size it was
     * realloc'ed to, until it is free'd. */

    /* The values of p[0] to p[9] are preserved, so this will print:
       42 15
    */
    printf("%d %d\n", p[0], p[9]);

    free(p);

    return EXIT_SUCCESS;
}
```

Перераспределенный объект может иметь или не иметь тот же адрес, что и `*p` . Поэтому

важно зафиксировать возвращаемое значение из `realloc` которое содержит новый адрес, если вызов будет успешным.

Убедитесь, что возвращаемое значение `realloc` присваивается `temporary` а не оригиналу `p` . `realloc` вернет null в случае любого сбоя, который перезапишет указатель. Это потеряет ваши данные и создаст утечку памяти.

Многомерные массивы переменной величины

C99

Поскольку C99, C имеет массивы переменной длины, VLA, эта модель массивов с границами, которые известны только во время инициализации. Хотя вы должны быть осторожны, чтобы не выделять слишком большие VLA (они могут разбивать ваш стек), использование *указателей в VLA* и использование их в выражениях `sizeof` в порядке.

```
double sumAll(size_t n, size_t m, double A[n][m]) {
    double ret = 0.0;
    for (size_t i = 0; i < n; ++i)
       for (size_t j = 0; j < m; ++j)
          ret += A[i][j]
    return ret;
}

int main(int argc, char *argv[argc+1]) {
   size_t n = argc*10;
   size_t m = argc*8;
   double (*matrix)[m] = malloc(sizeof(double[n][m]));
   // initialize matrix somehow
   double res = sumAll(n, m, matrix);
   printf("result is %g\n", res);
   free(matrix);
}
```

Здесь `matrix` является указателем на элементы типа `double[m]` , а выражение `sizeof` с `double[n][m]` гарантирует, что оно содержит пространство для `n` таких элементов.

Все это пространство распределено смежно и, таким образом, может быть освобождено одним звонком на `free` .

Наличие VLA на языке также влияет на возможные объявления массивов и указателей в заголовках функций. Теперь внутри `[]` параметров массива допускается общее целочисленное выражение. Для обеих функций выражения в `[]` используют параметры, которые были объявлены ранее в списке параметров. Для `sumAll` это длины, которые пользовательский код ожидает для матрицы. Что касается всех параметров функции массива в C, то самое внутреннее измерение переписывается на тип указателя, поэтому это эквивалентно объявлению

```
  double sumAll(size_t n, size_t m, double (*A)[m]);
```

То есть `n` не является частью функционального интерфейса, но информация может быть полезна для документации, и она также может использоваться ограничителями, проверяющими компиляторы, чтобы предупреждать об отсутствии доступа к границам.

По существу, для `main` выражение `argc+1` является минимальной длиной, которую стандарт C предписывает для аргумента `argv` .

Обратите внимание, что официальная поддержка VLA является необязательной в C11, но мы не знаем компилятора, который реализует C11, и у которого их нет. Если нужно, вы можете протестировать макрос `__STDC_NO_VLA__` .

realloc (ptr, 0) не эквивалентен свободному (ptr)

`realloc` *концептуально эквивалентен* `malloc + memcpy + free` на другом указателе.

Если размер запрашиваемого пространства равен нулю, поведение `realloc` определяется реализацией. Это похоже на все функции выделения памяти, которые получают параметр `size 0` . Такие функции могут фактически возвращать ненулевой указатель, но это никогда не должно быть разыменовано.

Таким образом, `realloc(ptr,0)` не эквивалентен `free(ptr)` . Это может

- быть «ленивой» реализацией и просто вернуть `ptr`
- `free(ptr)` , выделите фиктивный элемент и верните его
- `free(ptr)` и возврат `0`
- просто верните `0` для отказа и ничего не делайте.

Поэтому, в частности, последние два случая неразличимы по коду приложения.

Это означает, что `realloc(ptr,0)` может не освобождать / освобождать память и, следовательно, никогда не следует использовать в качестве замены `free` .

Пользовательское управление памятью

`malloc()` часто вызывает базовые функции операционной системы для получения страниц памяти. Но нет ничего особенного в функции, и она может быть реализована в прямом C, объявив большой статический массив и выделяя из него (есть небольшая трудность в обеспечении правильного выравнивания, на практике выравнивание до 8 байтов почти всегда адекватно).

Для реализации простой схемы блок управления хранится в области памяти непосредственно перед указателем, который должен быть возвращен из вызова. Это означает, что `free()` может быть реализовано путем вычитания из возвращаемого указателя и считывания управляющей информации, которая обычно является размером блока плюс некоторая информация, которая позволяет вернуть ее в свободный список - связанный список нераспределенных блоков.

Когда пользователь запрашивает выделение, выполняется поиск свободного списка до тех пор, пока не будет найден блок с одинаковым или большим размером запрашиваемой суммы, а затем при необходимости он будет разделен. Это может привести к фрагментации памяти, если пользователь постоянно делает много распределений и освобождает непредсказуемый размер, и с непредсказуемыми интервалами (не все реальные программы ведут себя так, простая схема часто подходит для небольших программ).

```
/* typical control block */
struct block
{
   size_t size;         /* size of block */
   struct block *next;  /* next block in free list */
   struct block *prev;  /* back pointer to previous block in memory */
   void *padding;       /* need 16 bytes to make multiple of 8 */
}

static struct block arena[10000]; /* allocate from here */
static struct block *firstfree;
```

Многим программам требуется большое количество распределений небольших объектов одинакового размера. Это очень легко реализовать. Просто используйте блок со следующим указателем. Поэтому, если требуется блок из 32 байтов:

```
union block
{
   union block * next;
   unsigned char payload[32];
}

static union block arena[100];
static union block * head;
void init(void)
{
    int i;
    for (i = 0; i < 100 - 1; i++)
        arena[i].next = &arena[i + 1];
    arena[i].next = 0; /* last one, null */
    head = &block[0];
}

void *block_alloc()
{
    void *answer = head;
    if (answer)
        head = head->next;
    return answer;
}

void block_free(void *ptr)
{
    union block *block = ptr;
    block->next = head;
    head - block;
}
```

Эта схема чрезвычайно быстрая и эффективная, и ее можно сделать родовым с определенной потерей ясности.

alloca: выделить память в стеке

Предостережение: `alloca` упоминается здесь только для полноты. Он полностью не переносится (не подпадает под действие каких-либо общих стандартов) и имеет ряд потенциально опасных функций, которые делают его небезопасным для не подозревающих. Современный C-код должен заменить его *массивами переменной длины* (VLA).

Ручная страница

```
#include <alloca.h>
// glibc version of stdlib.h include alloca.h by default

void foo(int size) {
    char *data = alloca(size);
    /*
      function body;
    */
    // data is automatically freed
}
```

Выделяйте память в стеке стека вызывающего, пространство, на которое ссылается возвращенный указатель, автоматически освобождается 'd, когда функция вызывающего абонента заканчивается.

Хотя эта функция удобна для автоматического управления памятью, имейте в виду, что запрос на большое выделение может привести к переполнению стека и что вы не можете использовать `free` с памятью, выделенной с помощью `alloca` (что может вызвать дополнительную проблему при переполнении стека).

По этой причине не рекомендуется использовать `alloca` внутри цикла или рекурсивную функцию.

И поскольку память `free` при возврате функции, вы не можете вернуть указатель как результат функции (поведение будет неопределенным).

Резюме

- вызов идентичен `malloc`
- автоматически free'd при возврате функции
- несовместимые со `free` функциями `realloc` (неопределенное поведение)
- указатель не может быть возвращен как результат функции (неопределенное поведение)
- размер ограниченного пространства стека, который (на большинстве машин) намного

меньше места кучи, доступного для использования `malloc()`
- избегайте использования `alloca()` и VLA (массивы с переменной длиной) в одной функции
- `alloca()` не так переносима, как `malloc()` et al.

Рекомендация

- Не используйте `alloca()` в новом коде

C99

Современная альтернатива.

```
void foo(int size) {
    char data[size];
    /*
      function body;
    */
    // data is automatically freed
}
```

Это работает там, где `alloca()` делает и работает в местах, где `alloca()` не работает (например, внутри циклов). Он предполагает либо реализацию C99, либо реализацию C11, которая не определяет `__STDC_NO_VLA__` .

Прочитайте Управление памятью онлайн: https://riptutorial.com/ru/c/topic/4726/управление-памятью

глава 61: Утверждение

Вступление

Утверждение является предикатом, что представленное условие должно быть истинным в момент, когда утверждение встречается программным обеспечением. Наиболее распространенными являются ***простые утверждения*** , которые проверяются во время выполнения. Тем не менее, ***статические утверждения*** проверяются во время компиляции.

Синтаксис

- утверждает (выражение)
- static_assert (выражение, сообщение)
- _Static_assert (выражение, сообщение)

параметры

параметр	подробности
выражение	выражение скалярного типа.
сообщение	строковый литерал, который будет включен в диагностическое сообщение.

замечания

Оба `assert` и `static_assert` - это макросы, определенные в `assert.h` .

Определение `assert` зависит от макроса `NDEBUG` который не определен стандартной библиотекой. Если `NDEBUG` определен, `assert` - это no-op:

```
#ifdef NDEBUG
#  define assert(condition) ((void) 0)
#else
#  define assert(condition) /* implementation defined */
#endif
```

Мнение зависит от того, следует ли использовать `NDEBUG` для компиляции продукции.

- Про-лагерей утверждает, что сообщения `assert abort` и assertion не полезны для конечных пользователей, поэтому результат не помогает пользователю. Если у вас есть фатальные условия для проверки производственного кода, вы должны

использовать обычные условия `if/else` и `exit` или `quick_exit` для завершения программы. В отличие от `abort` , они позволяют программе выполнять некоторую очистку (через функции, зарегистрированные с помощью `atexit` или `at_quick_exit`).
- Кон-лагерь утверждает, что вызовы `assert` никогда не должны запускаться в производственном коде, но если они это сделают, проверенное условие означает, что что-то резко ошибочно, и программа будет плохо себя вести, если исполнение продолжается. Поэтому лучше иметь утверждения, действующие в производственном кодексе, потому что, если они стреляют, ад уже сломался.
- Другой вариант - использовать систему утверждений, которая всегда выполняет проверку, но обрабатывает ошибки по-разному между разработкой (когда это `abort` подходит) и производством (где «неожиданная внутренняя ошибка - обратитесь в службу технической поддержки» может быть более уместным).

`static_assert` расширяется до `_Static_assert` который является ключевым словом. Условие проверяется во время компиляции, поэтому `condition` должно быть постоянным выражением. Нет необходимости в том, чтобы это было по-разному обрабатываться между развитием и производством.

Examples

Предварительное условие и постусловие

Одним из предпосылок для утверждения является предварительное условие и постусловие. Это может быть очень полезно для поддержания инварианта и дизайна по контракту . Например, длина всегда равна нулю или положительна, поэтому эта функция должна возвращать нулевое или положительное значение.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int length2 (int *a, int count)
{
    int i, result = 0;

    /* Precondition: */
    /* NULL is an invalid vector */
    assert (a != NULL);
    /* Number of dimensions can not be negative.*/
    assert (count >= 0);

    /* Calculation */
    for (i = 0; i < count; ++i)
    {
        result = result + (a[i] * a[i]);
    }

    /* Postcondition: */
    /* Resulting length can not be negative. */
```

```
    assert (result >= 0);
    return result;
}

#define COUNT 3

int main (void)
{
    int a[COUNT] = {1, 2, 3};
    int *b = NULL;
    int r;
    r = length2 (a, COUNT);
    printf ("r = %i\n", r);
    r = length2 (b, COUNT);
    printf ("r = %i\n", r);
    return 0;
}
```

Простое утверждение

Утверждение - это утверждение, используемое для утверждения, что факт должен быть правдой, когда эта строка кода будет достигнута. Утверждения полезны для обеспечения выполнения ожидаемых условий. Когда условие, переданное утверждению, истинно, действия нет. Поведение в ложных условиях зависит от флагов компилятора. Когда утверждения разрешены, ложный ввод вызывает немедленную остановку программы. Когда они отключены, никаких действий не предпринимается. Общепринятой практикой является включение утверждений во внутренние и отладочные сборки и их отключение в сборках релизов, хотя утверждения часто включаются в выпуске. (Оправдывание прекращается или хуже, чем ошибки зависят от программы.) Утверждения должны использоваться только для того, чтобы ловить внутренние ошибки программирования, что обычно означает, что передаются плохие параметры.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int main(void)
{
    int x = -1;
    assert(x >= 0);

    printf("x = %d\n", x);
    return 0;
}
```

Возможный выход с неопределенным `NDEBUG` :

```
a.out: main.c:9: main: Assertion `x >= 0' failed.
```

Возможный вывод с помощью `NDEBUG` :

```
x = -1
```

Хорошая практика - определить `NDEBUG` глобально, чтобы вы могли легко скомпилировать свой код со всеми утверждениями как `NDEBUG` , так и выключить. Легкий способ сделать это - определить `NDEBUG` в качестве опции для компилятора или определить его в общем заголовке конфигурации (например, `config.h`).

Статическое утверждение

C11

Статические утверждения используются для проверки правильности условия при компиляции кода. Если это не так, компилятор должен выпустить сообщение об ошибке и остановить процесс компиляции.

Статическое утверждение - это такое, которое проверяется во время компиляции, а не время выполнения. Условие должно быть константным выражением, и если false приведет к ошибке компилятора. Первый аргумент, проверяемое условие, должен быть константным выражением, а второй - строковым литералом.

В отличие от assert, `_Static_assert` - это ключевое слово. Удобный макрос `static_assert` определяется в `<assert.h>` .

```
#include <assert.h>

enum {N = 5};
_Static_assert(N == 5, "N does not equal 5");
static_assert(N > 10, "N is not greater than 10");  /* compiler error */
```

C99

До C11 не было прямой поддержки статических утверждений. Однако в C99 статические утверждения могут быть эмулированы с помощью макросов, которые могут вызвать сбой компиляции, если условие времени компиляции было ложным. В отличие от `_Static_assert` , второй параметр должен быть правильным именем маркера, чтобы с ним можно было создать имя переменной. Если утверждение не выполняется, имя переменной отображается в ошибке компилятора, так как эта переменная использовалась в синтаксически неправильном объявлении массива.

```
#define STATIC_MSG(msg, l) STATIC_MSG2(msg, l)
#define STATIC_MSG2(msg,l) on_line_##l##__##msg
#define STATIC_ASSERT(x, msg) extern char STATIC_MSG(msg, __LINE__) [(x)?1:-1]

enum { N = 5 };
STATIC_ASSERT(N == 5, N_must_equal_5);
STATIC_ASSERT(N > 5, N_must_be_greater_than_5); /* compile error */
```

До C99 вы не могли объявлять переменные в произвольных местоположениях в блоке,

поэтому вам пришлось бы очень осторожно относиться к использованию этого макроса, гарантируя, что он появляется только там, где объявление переменной будет действительным.

Утверждение недостижимого кода

Во время разработки, когда определенные пути кода должны быть недоступны для потока управления, вы можете использовать `assert(0)` чтобы указать, что такое условие является ошибочным:

```
switch (color) {
    case COLOR_RED:
    case COLOR_GREEN:
    case COLOR_BLUE:
        break;

    default:
        assert(0);
}
```

Всякий раз, когда аргумент макроса `assert()` вычисляет false, макрос будет записывать диагностическую информацию в стандартный поток ошибок, а затем прервать программу. Эта информация включает в себя номер файла и строки оператора `assert()` и может быть очень полезной при отладке. Зарезервировать можно, `NDEBUG` макрос `NDEBUG` .

Другой способ прервать программу при возникновении ошибки являются стандартной библиотеки функций `exit` , `quick_exit` или `abort` . `exit` и `quick_exit` принимают аргумент, который можно передать обратно в вашу среду. `abort()` (и, таким образом, `assert`) может быть действительно серьезным завершением вашей программы, и некоторые очистки, которые в противном случае были бы выполнены в конце выполнения, могут не выполняться.

Основным преимуществом `assert()` является то, что он автоматически печатает отладочную информацию. Вызов `abort()` имеет то преимущество, что он не может быть отключен как assert, но может не вызывать отображение какой-либо информации отладки. В некоторых ситуациях использование обеих конструкций вместе может быть полезным:

```
if (color == COLOR_RED || color == COLOR_GREEN) {
   ...
} else if (color == COLOR_BLUE) {
   ...
} else {
   assert(0), abort();
}
```

Когда `assert()` *включены* , вызов `assert()` будет печатать отладочную информацию и завершать программу. Выполнение никогда не достигает вызова `abort()` . Когда `assert()` *отключены* , вызов `assert()` ничего не делает и вызывается `abort()` . Это гарантирует, что

программа *всегда* заканчивается для этого условия ошибки; включение и отключение утверждений влияет только на то, напечатан ли вывод отладки.

Вы никогда не должны оставлять такое `assert` в производственном коде, потому что отладочная информация не полезна для конечных пользователей, и поскольку `abort` обычно является слишком строгим завершением, которое блокирует обработчики очистки, которые установлены для `exit` или `quick_exit` для запуска.

Сообщения об ошибках подтверждения

Существует трюк, который может отображать сообщение об ошибке вместе с утверждением. Обычно вы пишете такой код

```
void f(void *p)
{
    assert(p != NULL);
    /* more code */
}
```

Если утверждение не выполнено, сообщение об ошибке будет напоминать

> Утверждение не выполнено: p! = NULL, файл main.c, строка 5

Однако вы можете использовать логическое И (`&&`), чтобы дать сообщение об ошибке

```
void f(void *p)
{
    assert(p != NULL && "function f: p cannot be NULL");
    /* more code */
}
```

Теперь, если утверждение не выполняется, сообщение об ошибке будет читать что-то вроде этого

> Утверждение не выполнено: p! = NULL && "функция f: p не может быть NULL", файл main.c, строка 5

Причина в том, почему это работает, заключается в том, что строковый литерал всегда вычисляет ненулевое значение (true). Добавление `&& 1` к булевому выражению не влияет. Таким образом, добавление `&& "error message"` имеет никакого эффекта, за исключением того, что компилятор отобразит полное выражение, которое не удалось.

Прочитайте Утверждение онлайн: https://riptutorial.com/ru/c/topic/555/утверждение

глава 62: Файлы и потоки ввода-вывода

Синтаксис

- #include <stdio.h> / * Включите это, чтобы использовать любой из следующих разделов * /
- FILE * fopen (const char * path, const char * mode); / * Открыть поток по файлу по *пути* с указанным *режимом* * /
- FILE * freopen (const char * path, const char * mode, FILE * stream); / * Повторно открыть существующий поток в файле по *пути* с указанным *режимом* * /
- int fclose (поток FILE *); / * Закрыть открытый поток * /
- size_t fread (void * ptr, size_t size, size_t nmemb, поток FILE *); / * Прочитайте максимум *nmemb* элементов байтов *размера* каждый из *потока* и напишите их в *ptr* . Возвращает количество элементов чтения. * /
- size_t fwrite (const void * ptr, size_t size, size_t nmemb, поток FILE *); / * Напишите *nmemb* элементы байтов *размера* каждый из *ptr* в *поток* . Возвращает количество написанных элементов. * /
- int fseek (поток FILE *, длинное смещение, int whence); / * Установите курсор потока для *смещения* относительно *смещения* , указанного командой *whence* , и вернет 0, если это удалось. * /
- long ftell (поток FILE *); / * Возвращает смещение текущей позиции курсора от начала потока. * /
- void rewind (поток FILE *); / * Установите позицию курсора в начало файла. * /
- int fprintf (FILE * fout, const char * fmt, ...); / * Записывает строку формата printf в fout * /
- FILE * stdin; / * Стандартный поток ввода * /
- FILE * stdout; / * Стандартный выходной поток * /
- FILE * stderr; / * Стандартный поток ошибок * /

параметры

параметр	подробности
const char * mode	Строка, описывающая режим открытия потока с файловой поддержкой. См. Замечания для возможных значений.
int откуда	Может быть `SEEK_SET` для установки с начала файла `SEEK_END` для установки с его конца или `SEEK_CUR` для установки относительно текущего значения курсора. Примечание. `SEEK_END` не переносится.

замечания

Строки режима:

Модификации режима в `fopen()` и `freopen()` могут быть одним из следующих значений:

- `"r"` : открыть файл в режиме только для чтения, при этом курсор установлен в начало файла.
- `"r+"` : открыть файл в режиме чтения-записи, при этом курсор установлен в начало файла.
- `"w"` : открыть или создать файл в режиме только для записи, при этом его содержимое обрезается до 0 байтов. Курсор установлен в начало файла.
- `"w+"` : открыть или создать файл в режиме чтения-записи, при этом его содержимое обрезается до 0 байтов. Курсор установлен в начало файла.
- `"a"` : открыть или создать файл в режиме только записи, при этом курсор установлен в конец файла.
- `"a+"` : открыть или создать файл в режиме чтения-записи, при этом указатель чтения установлен в начало файла. Выход, однако, *всегда* будет добавлен в конец файла.

Каждый из этих файловых режимов может иметь `b` добавленный после начальной буквы (например, `"rb"` или `"a+b"` или `"ab+"`). `b` означает, что файл следует рассматривать как двоичный файл вместо текстового файла в тех системах, где есть разница. Это не влияет на Unix-подобные системы; это важно для систем Windows. (Кроме того, Windows `fopen` позволяет явному `t` вместо `b` указывать «текстовый файл» и множество других параметров для конкретной платформы.)

C11

- `"wx"` : создать текстовый файл в режиме только для записи. *Файл может не существовать* .
- `"wbx"` : создать двоичный файл в режиме только для записи. *Файл может не существовать* .

`x` , если присутствует, должен быть последним символом в строке режима.

Examples

Открыть и записать в файл

```
#include <stdio.h>   /* for perror(), fopen(), fputs() and fclose() */
#include <stdlib.h>  /* for the EXIT_* macros */

int main(int argc, char **argv)
{
    int e = EXIT_SUCCESS;

    /* Get path from argument to main else default to output.txt */
    char *path = (argc > 1) ? argv[1] : "output.txt";
```

```
    /* Open file for writing and obtain file pointer */
    FILE *file = fopen(path, "w");

    /* Print error message and exit if fopen() failed */
    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Writes text to file. Unlike puts(), fputs() does not add a new-line. */
    if (fputs("Output in file.\n", file) == EOF)
    {
        perror(path);
        e = EXIT_FAILURE;
    }

    /* Close file */
    if (fclose(file))
    {
        perror(path);
        return EXIT_FAILURE;
    }
    return e;
}
```

Эта программа открывает файл с именем, указанным в аргументе main, по `output.txt` для `output.txt` если аргумент не указан. Если файл с тем же именем уже существует, его содержимое отбрасывается, и файл рассматривается как новый пустой файл. Если файлы еще не существуют, создается вызов `fopen()` .

Если по какой-либо причине вызов `fopen()` завершился с ошибкой, он возвращает значение `NULL` и устанавливает значение глобальной переменной `errno` . Это означает, что программа может проверить возвращаемое значение после вызова `fopen()` и использовать `perror()` если `fopen()` терпит неудачу.

Если вызов `fopen()` завершается успешно, он возвращает действительный указатель `FILE` . Этот указатель затем может использоваться для ссылки на этот файл до тех пор, пока на нем не будет `fclose()` .

Функция `fputs()` записывает данный текст в открытый файл, заменяя любое предыдущее содержимое файла. Аналогично функции `fopen()` функция `fputs()` также устанавливает значение `errno` если она терпит неудачу, хотя в этом случае функция возвращает `EOF` для указания отказа (иначе он возвращает неотрицательное значение).

Функция `fclose()` сбрасывает любые буферы, закрывает файл и освобождает память, на которую указывает `FILE *` . Возвращаемое значение указывает на завершение так же, как и `fputs()` (хотя при успешном завершении возвращает «0»), снова также устанавливая значение `errno` в случае сбоя.

fprintf

Вы можете использовать `fprintf` в файле так же, как на консоли с `printf` . Например, чтобы отслеживать выигрыши игр, потери и связи, которые вы могли бы написать

```
/* saves wins, losses and, ties */
void savewlt(FILE *fout, int wins, int losses, int ties)
{
    fprintf(fout, "Wins: %d\nTies: %d\nLosses: %d\n", wins, ties, losses);
}
```

Замечание: некоторые системы (печально, Windows) не используют то, что большинство программистов назвали бы «нормальными» окончаниями строк. Хотя UNIX-подобные системы используют \ n для завершения строк, Windows использует пару символов: \ r (возврат каретки) и \ n (строка). Эта последовательность обычно называется CRLF. Однако при использовании C вам не нужно беспокоиться об этих деталях, зависящих от платформы. AC-компилятор необходим для преобразования каждого экземпляра \ n в правильную конечную строку платформы. Поэтому компилятор Windows будет конвертировать \ n в \ r \ n, но компилятор UNIX сохранит его как есть.

Выполнить процесс

```
#include <stdio.h>

void print_all(FILE *stream)
{
    int c;
    while ((c = getc(stream)) != EOF)
        putchar(c);
}
int main(void)
{
    FILE *stream;

    /* call netstat command. netstat is available for Windows and Linux */
    if ((stream = popen("netstat", "r")) == NULL)
        return 1;

    print_all(stream);
    pclose(stream);
    return 0;
}
```

Эта программа запускает процесс (`netstat`) через `popen()` и считывает весь стандартный вывод процесса и выводит его на стандартный вывод.

Примечание: `popen()` не существует в стандартной библиотеке C , но это скорее часть POSIX C)

Получить строки из файла с помощью getline ()

Библиотека POSIX C определяет функцию `getline()` . Эта функция выделяет буфер для хранения содержимого строки и возвращает новую строку, количество символов в строке и

размер буфера.

Пример программы, которая получает каждую строку из `example.txt` :

```
#include <stdlib.h>
#include <stdio.h>


#define FILENAME "example.txt"

int main(void)
{
  /* Open the file for reading */
  char *line_buf = NULL;
  size_t line_buf_size = 0;
  int line_count = 0;
  ssize_t line_size;
  FILE *fp = fopen(FILENAME, "r");
  if (!fp)
  {
    fprintf(stderr, "Error opening file '%s'\n", FILENAME);
    return EXIT_FAILURE;
  }

  /* Get the first line of the file. */
  line_size = getline(&line_buf, &line_buf_size, fp);

  /* Loop through until we are done with the file. */
  while (line_size >= 0)
  {
    /* Increment our line count */
    line_count++;

    /* Show the line details */
    printf("line[%06d]: chars=%06zd, buf size=%06zu, contents: %s", line_count,
        line_size, line_buf_size, line_buf);

    /* Get the next line */
    line_size = getline(&line_buf, &line_buf_size, fp);
  }

  /* Free the allocated line buffer */
  free(line_buf);
  line_buf = NULL;

  /* Close the file now that we are done with it */
  fclose(fp);

  return EXIT_SUCCESS;
}
```

Входной файл `example.txt`

```
This is a file
  which has
multiple lines
    with various indentation,
blank lines
```

```
a really long line to show that getline() will reallocate the line buffer if the length of a
line is too long to fit in the buffer it has been given,
  and punctuation at the end of the lines.
```

Выход

```
line[000001]: chars=000015, buf size=000016, contents: This is a file
line[000002]: chars=000012, buf size=000016, contents:   which has
line[000003]: chars=000015, buf size=000016, contents: multiple lines
line[000004]: chars=000030, buf size=000032, contents:     with various indentation,
line[000005]: chars=000012, buf size=000032, contents: blank lines
line[000006]: chars=000001, buf size=000032, contents:
line[000007]: chars=000001, buf size=000032, contents:
line[000008]: chars=000001, buf size=000032, contents:
line[000009]: chars=000150, buf size=000160, contents: a really long line to show that
getline() will reallocate the line buffer if the length of a line is too long to fit in the
buffer it has been given,
line[000010]: chars=000042, buf size=000160, contents:  and punctuation at the end of the
lines.
line[000011]: chars=000001, buf size=000160, contents:
```

В этом примере `getline()` изначально вызывается без выделенного буфера. Во время этого первого вызова `getline()` выделяет буфер, считывает первую строку и помещает содержимое строки в новый буфер. При последующих вызовах `getline()` обновляет один и тот же буфер и перераспределяет буфер только тогда, когда он больше не достаточно большой, чтобы соответствовать всей строке. Затем временный буфер освобождается, когда мы закончили с файлом.

Другой вариант - `getdelim()` . Это то же самое, что и `getline()` за исключением указания символа окончания строки. Это необходимо, только если последний символ строки для вашего типа файла не является \ n. `getline()` работает даже с текстовыми файлами Windows, потому что с завершением многобайтовой строки (`"\r\n"`) '\ n'` по-прежнему остается последним символом в строке.

Пример реализации `getline()`

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <stdint.h>


#if !(defined _POSIX_C_SOURCE)
typedef long int ssize_t;
#endif

/* Only include our version of getline() if the POSIX version isn't available. */

#if !(defined _POSIX_C_SOURCE) || _POSIX_C_SOURCE < 200809L
```

```
#if !(defined SSIZE_MAX)
#define SSIZE_MAX (SIZE_MAX >> 1)
#endif

ssize_t getline(char **pline_buf, size_t *pn, FILE *fin)
{
  const size_t INITALLOC = 16;
  const size_t ALLOCSTEP = 16;
  size_t num_read = 0;

  /* First check that none of our input pointers are NULL. */
  if ((NULL == pline_buf) || (NULL == pn) || (NULL == fin))
  {
    errno = EINVAL;
    return -1;
  }

  /* If output buffer is NULL, then allocate a buffer. */
  if (NULL == *pline_buf)
  {
    *pline_buf = malloc(INITALLOC);
    if (NULL == *pline_buf)
    {
      /* Can't allocate memory. */
      return -1;
    }
    else
    {
      /* Note how big the buffer is at this time. */
      *pn = INITALLOC;
    }
  }

  /* Step through the file, pulling characters until either a newline or EOF. */

  {
    int c;
    while (EOF != (c = getc(fin)))
    {
      /* Note we read a character. */
      num_read++;

      /* Reallocate the buffer if we need more room */
      if (num_read >= *pn)
      {
        size_t n_realloc = *pn + ALLOCSTEP;
        char * tmp = realloc(*pline_buf, n_realloc + 1); /* +1 for the trailing NUL. */
        if (NULL != tmp)
        {
          /* Use the new buffer and note the new buffer size. */
          *pline_buf = tmp;
          *pn = n_realloc;
        }
        else
        {
          /* Exit with error and let the caller free the buffer. */
          return -1;
        }

        /* Test for overflow. */
```

```
        if (SSIZE_MAX < *pn)
        {
          errno = ERANGE;
          return -1;
        }
      }

      /* Add the character to the buffer. */
      (*pline_buf)[num_read - 1] = (char) c;

      /* Break from the loop if we hit the ending character. */
      if (c == '\n')
      {
        break;
      }
    }

    /* Note if we hit EOF. */
    if (EOF == c)
    {
      errno = 0;
      return -1;
    }
  }

  /* Terminate the string by suffixing NUL. */
  (*pline_buf)[num_read] = '\0';

  return (ssize_t) num_read;
}

#endif
```

Открыть и записать в двоичный файл

```
#include <stdlib.h>
#include <stdio.h>


int main(void)
{
   result = EXIT_SUCCESS;

   char file_name[] = "outbut.bin";
   char str[] = "This is a binary file example";
   FILE * fp = fopen(file_name, "wb");

   if (fp == NULL)  /* If an error occurs during the file creation */
   {
     result = EXIT_FAILURE;
     fprintf(stderr, "fopen() failed for '%s'\n", file_name);
   }
   else
   {
     size_t element_size = sizeof *str;
     size_t elements_to_write = sizeof str;

     /* Writes str (_including_ the NUL-terminator) to the binary file. */
     size_t elements_written = fwrite(str, element_size, elements_to_write, fp);
```

```
     if (elements_written != elements_to_write)
     {
       result = EXIT_FAILURE;
       /* This works for >=c99 only, else the z length modifier is unknown. */
       fprintf(stderr, "fwrite() failed: wrote only %zu out of %zu elements.\n",
         elements_written, elements_to_write);
       /* Use this for <c99: *
       fprintf(stderr, "fwrite() failed: wrote only %lu out of %lu elements.\n",
         (unsigned long) elements_written, (unsigned long) elements_to_write);
        */
     }

     fclose(fp);
   }

   return result;
}
```

Эта программа создает и записывает текст в двоичной форме через функцию `fwrite` в файл `output.bin` .

Если файл с тем же именем уже существует, его содержимое отбрасывается, и файл рассматривается как новый пустой файл.

Бинарный поток представляет собой упорядоченную последовательность символов, которая может прозрачно записывать внутренние данные. В этом режиме байты записываются между программой и файлом без какой-либо интерпретации.

Чтобы записывать целые числа переносимым образом, необходимо знать, ожидает ли формат файла их в формате большой или малой длины, а также размер (обычно 16, 32 или 64 бита). Бит-сдвиг и маскирование могут затем использоваться для записи байтов в правильном порядке. Целые числа в C не гарантируют наличие двух дополняющих представлений (хотя почти все реализации выполняются). К счастью, преобразование в беззнаковый гарантированно использовать двойки комплемента. Поэтому код для записи знакового целого в двоичный файл немного удивителен.

```
/* write a 16-bit little endian integer */
int fput16le(int x, FILE *fp)
{
    unsigned int rep = x;
    int e1, e2;

    e1 = fputc(rep & 0xFF, fp);
    e2 = fputc((rep >> 8) & 0xFF, fp);

    if(e1 == EOF || e2 == EOF)
        return EOF;
    return 0;
}
```

Другие функции следуют одному и тому же шаблону с незначительными изменениями для размера и порядка байтов.

fscanf ()

Предположим, у нас есть текстовый файл, и мы хотим прочитать все слова в этом файле, чтобы выполнить некоторые требования.

file.txt :

```
This is just
a test file
to be used by fscanf()
```

Это основная функция:

```
#include <stdlib.h>
#include <stdio.h>

void printAllWords(FILE *);

int main(void)
{
  FILE *fp;

  if ((fp = fopen("file.txt", "r")) == NULL) {
      perror("Error opening file");
      exit(EXIT_FAILURE);
  }

  printAllWords(fp);

  fclose(fp);

  return EXIT_SUCCESS;
}

void printAllWords(FILE * fp)
{
    char tmp[20];
    int i = 1;

    while (fscanf(fp, "%19s", tmp) != EOF) {
        printf("Word %d: %s\n", i, tmp);
        i++;
    }
}
```

Выход будет:

```
Word 1: This
Word 2: is
Word 3: just
Word 4: a
Word 5: test
Word 6: file
Word 7: to
Word 8: be
Word 9: used
```

```
Word 10: by
Word 11: fscanf()
```

Чтение строк из файла

Заголовок `stdio.h` определяет функцию `fgets()` . Эта функция считывает строку из потока и сохраняет ее в указанной строке. Функция прекращает чтение текста из потока, когда считывается `n - 1` символ, читается символ новой строки (`'\n'`) или заканчивается конец файла (EOF).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE_LENGTH 80

int main(int argc, char **argv)
{
    char *path;
    char line[MAX_LINE_LENGTH] = {0};
    unsigned int line_count = 0;

    if (argc < 1)
        return EXIT_FAILURE;
    path = argv[1];

    /* Open file */
    FILE *file = fopen(path, "r");

    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Get each line until there are none left */
    while (fgets(line, MAX_LINE_LENGTH, file))
    {
        /* Print each line */
        printf("line[%06d]: %s", ++line_count, line);

        /* Add a trailing newline to lines that don't already have one */
        if (line[strlen(line) - 1] != '\n')
            printf("\n");
    }

    /* Close file */
    if (fclose(file))
    {
        return EXIT_FAILURE;
        perror(path);
    }
}
```

Вызов программы с аргументом, который представляет собой путь к файлу, содержащему следующий текст:

```
This is a file
  which has
multiple lines
    with various indentation,
blank lines



a really long line to show that the line will be counted as two lines if the length of a line
is too long to fit in the buffer it has been given,
  and punctuation at the end of the lines.
```

Результатом будет следующий вывод:

```
line[000001]: This is a file
line[000002]:   which has
line[000003]: multiple lines
line[000004]:     with various indentation,
line[000005]: blank lines
line[000006]:
line[000007]:
line[000008]:
line[000009]: a really long line to show that the line will be counted as two lines if the le
line[000010]: ngth of a line is too long to fit in the buffer it has been given,
line[000011]:  and punctuation at the end of the lines.
line[000012]:
```

Этот очень простой пример позволяет фиксированную максимальную длину строки, так что более длинные строки будут эффективно считаться двумя строками. Функция `fgets()` требует, чтобы вызывающий код предоставлял память, которая будет использоваться в качестве адресата для прочитанной строки.

POSIX делает доступной функцию `getline()` которая вместо этого внутренне выделяет память, чтобы увеличить буфер, если необходимо, для линии любой длины (при условии, что имеется достаточная память).

Прочитайте Файлы и потоки ввода-вывода онлайн: https://riptutorial.com/ru/c/topic/507/файлы-и-потоки-ввода-вывода

глава 63: Форматированный вход / выход

Examples

Печать значения указателя на объект

Чтобы напечатать значение указателя на объект (в отличие от указателя функции), используйте спецификатор преобразования `p` . Он определяется только для печати `void` -pointers, поэтому для распечатки значения не- `void` -pointer он должен быть явно преобразован («casted *») в `void*` .

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h>  /* for printf() */

int main(void)
{
  int i;
  int * p = &i;

  printf("The address of i is %p.\n", (void*) p);

  return EXIT_SUCCESS;
}
```

C99

Использование `<inttypes.h>` и `uintptr_t`

Другой способ печати указателей на C99 или более поздней версии использует тип `uintptr_t` и макросы из `<inttypes.h>` :

```
#include <inttypes.h> /* for uintptr_t and PRIXPTR */
#include <stdio.h>    /* for printf() */

int main(void)
{
  int  i;
  int *p = &i;

  printf("The address of i is 0x%" PRIXPTR ".\n", (uintptr_t)p);

  return 0;
}
```

Теоретически не может быть целочисленного типа, который может содержать любой указатель, преобразованный в целое число (поэтому тип `uintptr_t` может не существовать). На практике это действительно существует. Указатели на функции не должны быть конвертируемыми в тип `uintptr_t` хотя они чаще всего являются конвертируемыми.

Если тип `uintptr_t` существует, то `intptr_t` тип `intptr_t` . Непонятно, почему вы когда-либо хотели обрабатывать адреса как целые числа.

K & R C89

Предварительная стандартная история:

До C89 во время K & R-C не было никакого типа `void*` (ни заголовок `<stdlib.h>` , ни прототипы, и, следовательно, нет `int main(void)` notation), поэтому указатель был отброшен до `long unsigned int` и напечатан с использованием модификатор длины `lx` / спецификатор преобразования.

Пример, приведенный ниже, предназначен только для информационных целей. В настоящее время это неверный код, который очень хорошо может спровоцировать печально известное Undefined Behavior .

```
#include <stdio.h> /* optional in pre-standard C - for printf() */

int main()
{
  int  i;
  int *p = &i;

  printf("The address of i is 0x%lx.\n", (long unsigned) p);

  return 0;
}
```

Печать разницы значений двух указателей на объект

Вычитание значений двух указателей на объект приводит к значению целого числа [*1] . Таким образом, он будет напечатан с использованием, *по крайней мере,* спецификатора преобразования `d` .

Чтобы убедиться, что существует такой тип, достаточно широкий, чтобы провести такую «разницу между указателями», так как C99 `<stddef.h>` определяет тип `ptrdiff_t` . Чтобы напечатать `ptrdiff_t` используйте модификатор `t` длины.

C99

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */
#include <stddef.h> /* for ptrdiff_t */


int main(void)
{
  int a[2];
  int * p1 = &a[0], * p2 = &a[1];
  ptrdiff_t pd = p2 - p1;
```

```
  printf("p1 = %p\n", (void*) p1);
  printf("p2 = %p\n", (void*) p2);
  printf("p2 - p1 = %td\n", pd);

  return EXIT_SUCCESS;
}
```

Результат может выглядеть так:

```
p1 = 0x7fff6679f430
p2 = 0x7fff6679f434
p2 - p1 = 1
```

Обратите внимание, что результирующее значение разности масштабируется по размеру типа, на который указывает вычитаемая точка, `int` здесь. Размер `int` для этого примера равен 4.

---

* 1 Если два указателя, которые нужно вычесть, не указывают на один и тот же объект, поведение не определено.

Спецификации преобразования для печати

Спецификация конверсии	Тип аргумента	Описание
`i` , `d`	ИНТ	печатает десятичные числа
`u`	unsigned int	печатает десятичные числа
`o`	unsigned int	отпечатки восьмеричные
`x`	unsigned int	печатает шестнадцатеричный, строчный
`X`	unsigned int	печатает шестнадцатеричный, верхний регистр
`f`	двойной	печатает float с точностью по умолчанию 6, если не задана точность (нижний регистр используется для специальных чисел `nan` и `inf` или `infinity`)
`F`	двойной	prints float с точностью по умолчанию 6, если точность не задана (верхний регистр используется для специальных номеров `NAN` и `INF` или `INFINITY`)

Спецификация конверсии	Тип аргумента	Описание
`e`	двойной	печатает float с точностью по умолчанию 6, если точность не задана, используя научную нотацию с использованием мантиссы / экспоненты; индекс нижнего регистра и специальные номера
`E`	двойной	печатает float с точностью по умолчанию 6, если точность не задана, используя научную нотацию с использованием мантиссы / экспоненты; индекс верхнего регистра и специальные числа
`g`	двойной	использует либо `f` либо `e` *[см. ниже]*
`G`	двойной	использует `F` или `E` *[см. ниже]*
`a`	двойной	печатает шестнадцатеричный, строчный
`A`	двойной	печатает шестнадцатеричный, верхний регистр
`c`	голец	печатает один символ
`s`	символ *	печатает строку символов до терминатора `NUL` или усекает до длины, заданной точностью, если указано
`p`	недействительным *	печатает значение `void` -pointer; не `void` - указатель должен быть явно преобразован («слепок») к `void*` ; указатель на объект, а **не** указатель функции
`%`	н /	печатает символ `%`
`n`	int *	***напишите*** количество байт, напечатанных до сих пор в `int` указывает.

Обратите внимание, что модификаторы длины могут быть применены к `%n` (например, `%hhn` указывает, что *следующий указатель преобразования `n` применяется к указателю на знак `signed char`* соответствии с ISO / IEC 9899: 2011 §7.21.6.1 ¶7).

Обратите внимание, что преобразования с плавающей запятой применяются к типам `float` и `double` из-за правил рассылки по умолчанию - §6.5.2.2. Вызовы функций, ¶7 *. Обозначение*

многоточия в декларатор прототипа функции приводит к тому, что преобразование типа аргумента останавливается после последнего объявленного параметра. Продвижение аргументов по умолчанию выполняется по завершающим аргументам.) Таким образом, такие функции, как `printf()` передаются только `double` , даже если указанная переменная имеет тип `float` .

В форматах `g` и `G` выбор между обозначениями `e` и `f` (или `E` и `F`) документируется в стандарте C и в спецификации POSIX для `printf()` :

> Двойной аргумент, представляющий число с плавающей запятой, должен быть преобразован в стиле `f` или `e` (или в стиле `F` или `E` в случае спецификатора преобразования `G`), в зависимости от преобразованного значения и точности. Пусть `P` равно точности, если отличная от нуля, 6, если точность опущена, или 1, если точность равна нулю. Тогда, если преобразование со стилем `E` будет иметь показатель `X` :
>
> - Если P> X> = -4, преобразование должно быть со стилем `f` (или `F`) и точностью `P - (X+1)` .
> - В противном случае преобразование должно быть со стилем `e` (или `E`) и точностью `P - 1` .
>
> Наконец, если не используется флаг «#», любые конечные нули должны быть удалены из дробной части результата, а символ десятичной точки должен быть удален, если не осталось дробной части.

Функция printf ()

Доступ через функцию `<stdio.h>` , функция `printf()` является основным инструментом, используемым для печати текста на консоли в C.

```
printf("Hello world!");
// Hello world!
```

Обычные, неформатированные массивы символов могут быть напечатаны самим собой, помещая их непосредственно между круглыми скобками.

```
printf("%d is the answer to life, the universe, and everything.", 42);
// 42 is the answer to life, the universe, and everything.

int x = 3;
char y = 'Z';
char* z = "Example";
printf("Int: %d, Char: %c, String: %s", x, y, z);
// Int: 3, Char: Z, String: Example
```

Альтернативно, целые числа, числа с плавающей запятой, символы и т. Д. Могут быть напечатаны с использованием escape-символа `%` , за которым следует символ или

последовательность символов, обозначающих формат, известный как *спецификатор формата* .

Все дополнительные аргументы функции `printf()` разделяются запятыми, и эти аргументы должны быть в том же порядке, что и спецификаторы формата. Дополнительные аргументы игнорируются, а неверно введенные аргументы или отсутствие аргументов приводят к ошибкам или неопределенному поведению. Каждый аргумент может быть либо буквальным значением, либо переменной.

После успешного выполнения количество напечатанных символов возвращается с типом `int` . В противном случае отказ возвращает отрицательное значение.

Модификаторы длины

Стандарты C99 и C11 определяют следующие модификаторы длины для `printf()` ; их значения:

Модификатор	Модифицирует	Относится к
чч	d, i, o, u, x или X	`char` , `signed char` или `unsigned char`
час	d, i, o, u, x или X	`short int` или `unsigned short int`
L	d, i, o, u, x или X	`long int` или `unsigned long int`
L	a, A, e, E, f, F, g или G	`double` (для совместимости с `scanf()` ; undefined в C90)
Л.Л.	d, i, o, u, x или X	`long long int` или `unsigned long long int`
J	d, i, o, u, x или X	`intmax_t` или `uintmax_t`
Z	d, i, o, u, x или X	`size_t` или соответствующий тип `ssize_t` (`ssize_t` в POSIX)
T	d, i, o, u, x или X	`ptrdiff_t` или соответствующий целочисленный тип без знака
L	a, A, e, E, f, F, g или G	`long double`

Если модификатор длины появляется с любым спецификатором преобразования, отличным от указанного выше, поведение не определено.

Microsoft указывает некоторые модификаторы длины и явно не поддерживает `hh` , `j` , `z` или `t` .

Модификатор	Модифицирует	Относится к
I32	d, i, o, x или X	`__int32`
I32	o, u, x или X	`unsigned __int32`
I64	d, i, o, x или X	`__int64`
I64	o, u, x или X	`unsigned __int64`
я	d, i, o, x или X	`ptrdiff_t` (то есть `__int32` на 32-битных платформах, `__int64` на 64-битных платформах)
я	o, u, x или X	`size_t` (то есть `unsigned __int32` на 32-битных платформах, `unsigned __int64` на 64-битных платформах)
л или L	a, A, e, E, f, g или G	`long double` (В Visual C ++, хотя `long double` является отдельным типом, он имеет то же внутреннее представление, что и `double`).
l или w	c или C	Широкий характер с функциями `printf` и `wprintf` . (`lc` , `lC` , `wc` или `wC` является синонимом функций `C` в `printf` и `c` функциями `wprintf` .)
l или w	s, S или Z	Широкосимвольная строка с функциями `printf` и `wprintf` . (`wS` типа `ls` , `lS` , `ws` или `wS` является синонимом `S` в функциях `printf` и с `s` в функциях `wprintf` .)

Обратите внимание, что спецификаторы преобразования `C` , `S` и `Z` и модификаторы `I` , `I32` , `I64` и `w` length являются расширениями Microsoft. Обработка `l` в качестве модификатора для `long double` а не `double` , отличается от стандарта, хотя вам будет трудно определить разницу, если `long double` имеет другого представления из `double` .

Флаги формата печати

Стандарт C (C11 и C99 тоже) определяет следующие флаги для `printf()` :

Флаг	Конверсии	Имея в виду
`-`	все	Результат преобразования должен быть оставлен по полю в поле. Преобразование является правильным, если этот флаг не указан.
`+`	числовое	Результат подписанного преобразования всегда начинается со

Флаг	Конверсии	Имея в виду
	число	знака ('+' или '-'). Преобразование начинается со знака только тогда, когда отрицательное значение преобразуется, если этот флаг не указан.
`<space>`	числовое число	Если первый символ подписанного преобразования не является признаком или если подписанное преобразование не приводит к символам, в результат должен быть префикс `<space>` . Это означает, что при появлении флагов `<space>` и ' `+` ' флаг `<space>` игнорируется.
`#`	все	Указывает, что значение должно быть преобразовано в альтернативную форму. Для преобразования `o` , оно должно увеличивать точность, если и только если необходимо, чтобы заставить первую цифру результата быть нулевой (если значение и точность равны 0, выводится один 0). Для спецификаторов преобразования `x` или `X` ненулевой результат должен иметь префикс `0x` (или `0X`). Для спецификаторов преобразования `a` , `A` , `e` , `E` , `f` , `F` , `g` и `G` результат всегда должен содержать знак радиуса, даже если никакие цифры не следуют за символом radix. Без этого флага в результате этих преобразований появляется символ радиуса, только если после него следует цифра. Для спецификаторов преобразования `g` и `G` конечные нули не должны удаляться из результата, как обычно. Для других спецификаторов преобразования поведение не определено.
`0`	числовой	Для d, i, o, u, x, X, a, A, e, E, f, F, g и G спецификаторов преобразования, ведущие нули (после любого указания знака или основания) используются для заполнения поля ширину, а не выполнение пробела, за исключением того, что при преобразовании бесконечности или NaN. Если появляются флаги «0» и «-», флаг «0» игнорируется. Для спецификаторов преобразования d, i, o, u, x и X, если задана точность, флаг «0» игнорируется. Если появляются флаги '0' и `<apostrophe>` , символы группировки вставляются перед нулевым заполнением. Для других преобразований поведение не определено.

Эти флаги также поддерживаются Microsoft с тем же значением.

Спецификация POSIX для `printf()` добавляет:

Флаг	Конверсии	Имея в виду
'	i, d, u, f, F, g, G	Целочисленная часть результата десятичного преобразования должна быть отформатирована тысячами символов группировки. Для других преобразований поведение не определено. Используется символ неденежной группировки.

Прочитайте Форматированный вход / выход онлайн: https://riptutorial.com/ru/c/topic/3750/форматированный-вход---выход

кредиты

S. No	Главы	Contributors
1	Начало работы с языком C	4444, Abhineet, Alejandro Caro, alk, Ankush, ArturFH, Bahm, bevenson, bfd, Blacksilver, blatinox, bta, chqrlie, Community, Dair, Dan Fairaizl, Daniel Jour, Daniel Margosian, David G., David Grayson, Donald Duck, Dov Benyomin Sohacheski, Ed Cottrell, employee of the month, EOF, EsmaeelE, Frosty The DopeMan, Iskar Jarak, Jens Gustedt, John Slegers, JonasCz, Jonathan Leffler, Juan T, juleslasne, Kusalananda, Leandros, LiHRaM, Lundin, Malick, Mark Yisri, MC93, MoultoB, msohng, Myst, Narox Nox, Neal, Nemanja Boric, Nicolas Verlet, OiciTrap , P.P., PSN, Rakitić, RamenChef, Roland Illig, Ryan Hilbert, Shoe, Shog9, skrtbhtngr, sohnryang, stackptr, syb0rg, techydesigner, tlhIngan, Toby, vasili111, Vin, Vraj Pandya
2	- классификация символов и конверсия	Alejandro Caro, Jonathan Leffler, Roland Illig, Toby
3	Interprocess Communication (IPC)	cLdseed, EsmaeelE, Jonathan Leffler, Toby
4	Typedef	Buser, Chandrahas Aroori, GoodDeeds, Jonathan Leffler, mame98, PhotometricStereo, Stephen Leppik, Toby
5	Valgrind	abacles, alk, Ankush, Chandrahas Aroori, Devansh Tandon, drov, Firas Moalla, J F, Jonathan Leffler, vasili111
6	X-макросы	Cimbali, Jens Gustedt, John Bollinger, Leandros, MD XF, mpromonet, poolie, RamenChef, technosaurus, templatetypedef, Toby
7	Аргументы командной строки	4386427, A B, alk, drov, dvhh, Jonathan Leffler, Malcolm McLean, Shog9, syb0rg, Toby, Woodrow Barlow, Yotam Salmon
8	атомная энергетика	Jens Gustedt
9	Битовые поля	alk, EvilTeach, Fantastic Mr Fox, haccks, Ishay Peled, Jens Gustedt, John Odom, Jonathan Leffler, Lundin, madD7, Paul Hutchinson, RamenChef, Rishikesh Raje, Toby, vkgade
10	Встраивание	Alex, EsmaeelE, Jens Gustedt, Jonathan Leffler, Toby

11	Встроенная сборка	bevenson, EsmaeelE, Jonathan Leffler
12	Выборочные заявления	alk, bevenson, Blagovest Buyukliev, Faisal Mudhir, GoodDeeds, gsamaras, jxh, L.V.Rao, lordjohncena, MikeCAT, NeoR, noamgot, OznOg, P.P., Toby, tofro
13	Генерация случайных чисел	dylanweber, ganchito55, haccks, hexwab, Jonathan Leffler, Leandros, Malcolm McLean, MikeCAT, Toby
14	Декларация против определения	Ashish Ahuja, foxtrot9, Kerrek SB, Toby
15	инициализация	Jonathan Leffler, Liju Thomas, P.P.
16	Итерационные выражения / Циклы: для, пока, делать-пока	alk, GoodDeeds, Jens Gustedt, jxh, L.V.Rao, Malcolm McLean, Nagaraj, RamenChef, reshad, Toby
17	Классы хранения	alk, Blagovest Buyukliev, Chrono Kitsune, greatwolf, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler, L.V.Rao, madD7, Neui, Nitinkumar Ambekar, P.P., Toby, tversteeg, vuko_zrno
18	Комментарии	Ankush, Chandrahas Aroori, Jonathan Leffler, Toby
19	компиляция	alk, Amani Kilumanga, bevenson, Blacksilver, Firas Moalla, haccks, Ishay Peled, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler, Jossi, jxh, MC93, MikeCAT, nathanielng, P.P., Qrchack, R. Joiny, syb0rg, Toby, tofro, Turtle, Vraj Pandya, Алексей Неудачин
20	Литералы для чисел, символов и строк	Jens Gustedt, Jonathan Leffler, Klas Lindbäck, Neui, Paul92, Toby
21	логический	alk, Bob__, Braden Best, Chrono Kitsune, dhein, Insane, Jens Gustedt, Magisch, Mateusz Piotrowski, Peter, Toby
22	Массивы	2501, alk, AnArrayOfFunctions, AShelly, cdrini, cSmout, Dariusz, Elazar, Eli Sadoff, Firas Moalla, Guy, Iskar Jarak, Jasmin Solanki, Jens Gustedt, John Bollinger, Jonathan Leffler, L.V.Rao, Leandros, Liju Thomas, lordjohncena, Magisch, mhk, OznOg, Ray, Ryan Haining, Ryan Hilbert, stackptr, Toby, Waqas Bukhary
23	Многопоточность	Parham Alvani, Toby

24	Многосимвольная последовательность символов	Jonathan Leffler, PassionInfinite, Toby
25	Неопределенное поведение	2501, Abhineet, Aleksi Torhamo, alk, Antti Haapala, Armali, Ben Steffan, blatinox, bta, BurnsBA, caf, Christoph, Cody Gray, Community, cshu, DaBler, Daniel Jour, DarkDust, FedeWar, Firas Moalla, Giorgi Moniava, gsamaras, haccks, hmijail, honk, Jacob H, Jean-Baptiste Yunès, Jens Gustedt, John, John Bollinger, Jonathan Leffler, Kamiccolo, Leandros, Lundin, Magisch, Mark Yisri, Martin, MikeCAT, Nemanja Boric, P.P., Peter, Roland Illig, TimF, Toby, tversteeg, user45891, Vasfed, void
26	Неявные и явные конверсии	alk, Firas Moalla, Jens Gustedt, Jeremy Thien, kdopen, Lundin, Toby
27	Область идентификатора	embedded_guy, Firas Moalla, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler
28	Обработка ошибок	Jens Gustedt, stackptr
29	Обработка сигналов	3442, alk, Dariusz, Jens Gustedt, Leandros, mirabilos
30	Общие идиомы программирования C и методы разработчика	Chandrahas Aroori, Jonathan Leffler, Nityesh Agarwal, Shubham Agrawal
31	Общий выбор	2501, Jens Gustedt, Sun Qingyao
32	Объявления	alk, AnArrayOfFunctions, Blacksilver, Firas Moalla, J Wu, Jens Gustedt, Jonathan Leffler, Jonathon Reinhart
33	Обычные подводные камни	abacles, Accepted Answer, alk, bevenson, Bjorn A., Chrono Kitsune, clearlight, Community, Dmitry Grigoryev, Dreamer, Dunno, FedeWar, Fred Barclay, Gavin Higham, Giorgi Moniava , hlovdal, Ishay Peled, Jeremy, John Hascall, Jonathan Leffler, Ken Y-N, Leandros, Lord Farquaad, MikeCAT, P.P., Roland Illig, rxantos, Sourav Ghosh, stackptr, Tamarous, techEmbedded, Toby, Waqas Bukhary
34	Ограничения	Armali, Toby, Vality
35	операторы	202_accepted, 3442, alk, Amani Kilumanga, Andrea Corbelli, Bakhtiar Hasan, BenG, blatinox, cpplearner, Damien, Dariusz, EsmaeelE, Faisal Mudhir, Fantastic Mr Fox, Firas Moalla, gsamaras, hrs, Iwillnotexist Idonotexist, Jens Gustedt,

		Jonathan Leffler, kdopen, Ken Y-N, L.V.Rao, Leandros, LostAvatar, Magisch, MikeCAT, noamgot, P.P., Paul92, Peter, stackptr, Toby, Will, Wolf, Yu Hao
36	Параметры функции	2501, Alejandro Caro, alk, Chrono Kitsune, ganesh kumar, George Stocker, Jens Gustedt, Jonathan Leffler, Leandros, MikeCAT, Minar Ashiq Tishan, P.P., RamenChef, Richard Chambers, someoneigna, syb0rg, Toby
37	Передача 2D-массивов в функции	deamentiaemundi, Malcolm McLean, Shrinivas Patgar, Toby
38	Перейти к началу страницы	alk, Jens Gustedt, Jonathan Leffler, lordjohncena, Malcolm McLean, Sourav Ghosh, syb0rg, Toby
39	Переменные аргументы	2501, Blacksilver, eush77, Jean-Baptiste Yunès, Jonathan Leffler, Leandros, mirabilos, syb0rg, Toby
40	Перечисления	Alejandro Caro, alk, jasoninnn, Jens Gustedt, Jonathan Leffler, OznOg, Toby
41	Побочные эффекты	EsmaeelE, Jonathan Leffler, L.V.Rao, madD7, RamenChef, Sirsireesh Kodali, Toby
42	Поведение, определяемое реализацией	Jens Gustedt, John Bollinger, P.P.
43	Препроцессор и макросы	Alex Garcia, alk, bevenson, bwoebi, Dariusz, DrPrItay, Erlend Graff, EsmaeelE, EvilTeach, fastlearner, Firas Moalla, gman, hashdefine, hlovdal, javac, Jens Gustedt, Jonathan Leffler, Justin, Leandros, luser droog, Madhusoodan P, Maniero, mnoronha, Nitinkumar Ambekar, P.P., Paul J. Lucas, Peter, Richard Chambers, Robert Baldyga, stackptr, Toby, v7d8dpo4
44	Прокладка и упаковка структуры	EsmaeelE, Jarrod Dixon, Jedi, Jesferman, Jonathan Leffler, Liju Thomas, MayeulC, tilz0R
45	Псевдонимы и эффективный тип	2501, 4386427, Jens Gustedt
46	Связанные списки	4386427, alk, Andrea Biondo, bevenson, iRove, Jonathan Leffler, Jossi, Leandros, Mateusz Piotrowski, Ryan, Toby
47	Создание и включение файлов заголовков	4444, Jonathan Leffler, patrick96, Sirsireesh Kodali

48	Составные литералы	alk, haccks, Jens Gustedt, Kerrek SB
49	Союзы	Jossi, RamenChef, Toby, Vality
50	Стандартная математика	Alejandro Caro, alk, Blagovest Buyukliev, immerhart, Jonathan Leffler, manav m-n, Toby
51	Структуры	alk, Chrono Kitsune, Damien, Elazar, EsmaeelE, Faisal Mudhir, Firas Moalla, gmug, jasoninnn, Jens Gustedt, Jonathan Leffler, Jossi, kamoroso94, Madhusoodan P, OznOg, Paul Kramme, PhotometricStereo, RamenChef, Toby, Vality
52	Структуры тестирования	Community, EsmaeelE, Jonathan Leffler, lordjohncena, Toby, user2314737, vuko_zrno
53	Струны	4386427, alk, Amani Kilumanga, Andrey Markeev, bevenson, catalogue_number, Chris Sprague, Chrono Kitsune, Cody Gray , Damien, Daniel, depperm, dylanweber, FedeWar, Firas Moalla, haccks, Ishay Peled, jasoninnn, Jean-Baptiste Yunès, Jens Gustedt, John Bollinger, Jonathan Leffler, Leandros, Malcolm McLean, mantal, MikeCAT, P.P., Purag, Roland Illig, stackptr, still_learning, syb0rg, Toby, vasili111, Waqas Bukhary , Wolf, Wyzard, Алексей Неудачин
54	Темы (родной)	alk, Jens Gustedt, P.P.
55	Тип Квалификаторы	alk, Blagovest Buyukliev, Jens Gustedt, Jesferman, madD7, tversteeg
56	Типы данных	2501, alk, Blagovest Buyukliev, Firas Moalla, Jens Gustedt, Keith Thompson, Ken Y-N, Leandros, P.P., Peter, WMios
57	Точки последовательности	2501, Armali, bta, Community, haccks, Jens Gustedt, John Bode, Toby
58	указатели	0xEDD1E, alk, Altece, Amani Kilumanga, Andrey Markeev, Ankush, Antti Haapala, Ashish Ahuja, Bjorn A., bruno, bta, chqrlie, Courtney Pattison, Dair, Daniel Porteous, David G., dhein, dkrmr, Don't You Worry Child, e.jahandar, elslooo, EOF, erebos, Faisal Mudhir, Fantastic Mr Fox, FedeWar, Firas Moalla, fluter, foxtrot9, Gavin Higham, gdc, Giorgi Moniava, gsamaras, haccks, haltode, Harry Johnston, Hemant Kumar, honk, Jens Gustedt, Jonathan Leffler, Jonnathan Soares, Josh de Kock, jpX, L.V.Rao, LaneL, Leandros, Luiz Berti, Malcolm McLean, Matthieu, Michael Fitzpatrick, MikeCAT, Neui, Nitinkumar Ambekar, OiciTrap, P.P., Pbd, Peter, RamenChef, raymai97, Rohan, Sergey, Shahbaz, signal, slugonamission,

		solomonope, someoneigna, Spidey, Srikar, stackptr, syb0rg, tbodt, the sudhakar, thndrwrks, Toby, Vality, vijay kant sharma, Vivek S, Wyzard, xhienne, Алексей Неудачин
59	Указатели функций	Alejandro Caro, alk, David Refaeli, Filip Allberg, hlovdal, John Burger, Leandros, Malcolm McLean, P.P., Srikar, stackptr, Toby
60	Управление памятью	4386427, alk, Anderson Giacomolli, Andrey Markeev, Ankush, Antti Haapala, Cullub, Daksh Gupta, dhein, dkrmr, doppelheathen, dvhh, elslooo, EOF, EsmaeelE, Firas Moalla, fluter, gdc, greatwolf, honk, Jens Gustedt, Jonathan Leffler, juleslasne, Luiz Berti, madD7, Malcolm McLean, Mark Yisri, Matthieu, Neui, P.P., Paul Campbell, Paul V, reflective_mind, Seth, Srikar, stackptr, syb0rg, Tamarous, tbodt, the sudhakar, Toby, tofro, Vivek S, vuko_zrno, Wyzard
61	Утверждение	2501, AShelly, Blagovest Buyukliev, bta, eush77, greatwolf, J Wu, Jens Gustedt, Jonathan Leffler, Jossi, jxh, Leandros, Malcolm McLean, Ryan Haining, stackptr, syb0rg, Tim Post, Toby
62	Файлы и потоки ввода-вывода	alk, bevenson, EWoodward, haccks, iRove, Jean Vitor, Jens Gustedt, Jonathan Leffler, Jossi, Leandros, Malcolm McLean, Pedro Henrique A. Oliveira, RamenChef, reshad, Snaipe, stackptr, syb0rg, tkk, Toby, tversteeg, William Pursell
63	Форматированный вход / выход	alk, fluter, Jonathan Leffler, Jossi, lardenn, MikeCAT, polarysekt, StardustGogeta