

# Оглавление

<b>Об авторе</b> .....	15
<b>О научных редакторах</b> .....	16
<b>Введение</b> .....	17
Для кого эта книга .....	18
Структура издания .....	19
Условия, при соблюдении которых книга будет максимально полезной .....	21
Скачивание файлов с примерами кода .....	22
Условные обозначения .....	22
От издательства .....	24
<b>Глава 1. Основные возможности языка</b> .....	25
Директивы препроцессора .....	27
Макросы .....	28
Условная компиляция .....	41
Указатели на переменные .....	44
Синтаксис .....	45
Арифметические операции с указателями на переменные .....	47
Обобщенные указатели .....	50
Размер указателей .....	53
Висячие указатели .....	53
Общая информация о функциях .....	56
Анатомия функции .....	56
Роль функций в архитектуре приложений .....	57
Управление стеком .....	57
Передача по значению и передача по ссылке .....	58
Указатели на функции .....	60
Структуры .....	63
Зачем нужны структуры .....	63
Зачем нужны пользовательские типы .....	64
Принцип работы структур .....	65

Размещение структур в памяти .....	66
Вложенные структуры .....	70
Указатели на структуры .....	71
Резюме .....	72
<b>Глава 2. Компиляция и компоновка .....</b>	<b>74</b>
Процесс компиляции .....	75
Сборка проекта на языке C.....	77
Этап 1: предобработка .....	83
Этап 2: компиляция в ассемблерный код .....	85
Этап 3: компиляция в машинные инструкции .....	88
Этап 4: компоновка.....	90
Препроцессор.....	93
Компилятор .....	97
Дерево абстрактного синтаксиса .....	98
Ассемблер.....	100
Компоновщик .....	101
Принцип работы компоновщика.....	102
Компоновщик можно обмануть!.....	110
Декорирование имен в C++ .....	114
Резюме .....	116
<b>Глава 3. Объектные файлы .....</b>	<b>117</b>
Двоичный интерфейс приложений.....	118
Форматы объектных файлов.....	119
Переносимые объектные файлы .....	121
Исполняемые объектные файлы.....	125
Статические библиотеки .....	129
Динамические библиотеки.....	138
Ручная загрузка разделяемых библиотек.....	142
Резюме .....	145
<b>Глава 4. Структура памяти процесса.....</b>	<b>146</b>
Внутреннее устройство памяти процесса .....	147
Исследование структуры памяти .....	148
Исследование статической схемы размещения в памяти .....	149
Сегмент BSS.....	151
Сегмент Data .....	153
Сегмент Text.....	157
Исследование динамической схемы размещения в памяти .....	159
Отражение памяти.....	160
Стек.....	164
Куча.....	166
Резюме .....	169

<b>Глава 5. Стек и куча</b> .....	171
Стек .....	172
Исследование содержимого стека.....	173
Рекомендации по использованию стековой памяти.....	179
Куча .....	183
Выделение и освобождение памяти в куче .....	185
Принцип работы кучи .....	193
Управление памятью в средах с ограниченными ресурсами .....	197
Среды с ограниченной памятью .....	198
Высокопроизводительные среды .....	200
Резюме .....	206
<b>Глава 6. ООП и инкапсуляция</b> .....	208
Объектно-ориентированное мышление .....	210
Как мы мыслим.....	211
Диаграммы связей и объектные модели .....	212
В коде нет никаких объектов .....	214
Атрибуты объектов.....	216
Предметная область .....	216
Отношения между объектами .....	217
Объектно-ориентированные операции .....	218
Объекты имеют поведение .....	221
Почему язык С не является объектно-ориентированным .....	221
Инкапсуляция .....	222
Инкапсуляция атрибутов.....	223
Инкапсуляция поведения .....	225
Принцип сокрытия информации .....	235
Резюме .....	242
<b>Глава 7. Композиция и агрегация</b> .....	244
Отношения между классами .....	244
Объекты и классы.....	245
Композиция.....	247
Агрегация .....	253
Резюме.....	259
<b>Глава 8. Наследование и полиморфизм</b> .....	260
Наследование.....	260
Природа наследования.....	261
Полиморфизм .....	277
Что такое полиморфизм .....	277
Зачем нужен полиморфизм.....	280
Полиморфное поведение в языке С .....	280
Резюме.....	288

<b>Глава 9. Абстракция данных и ООП в C++</b> .....	289
Абстракция данных.....	289
Объектно-ориентированные концепции в C++.....	293
Инкапсуляция.....	293
Наследование .....	296
Полиморфизм .....	302
Абстрактные классы.....	305
Резюме .....	306
<b>Глава 10. История и архитектура Unix</b> .....	307
История Unix.....	308
Multics OS и Unix .....	308
BCPL и B.....	309
Путь к C.....	310
Архитектура Unix.....	312
Философия.....	312
Многослойная структура Unix.....	314
Интерфейс командной оболочки для пользовательских приложений .....	317
Интерфейс ядра для кольца командной оболочки .....	322
Ядро.....	327
Аппаратное обеспечение .....	332
Резюме .....	334
<b>Глава 11. Системные вызовы и ядра</b> .....	335
Системные вызовы.....	335
Тщательное исследование системных вызовов.....	336
Выполнение системного вызова напрямую, в обход стандартной библиотеки C.....	337
Внутри функции syscall .....	340
Добавление системного вызова в Linux .....	342
Ядра Unix .....	355
Монолитные ядра и микроядра .....	356
Linux .....	357
Модули ядра .....	358
Резюме .....	364
<b>Глава 12. Последние нововведения в C</b> .....	365
C11 .....	366
Определение поддерживаемой версии стандарта C.....	366
Удаление функции gets .....	368
Изменения в функции fopen.....	368
Функции с проверкой диапазона.....	370

Невозвращаемые функции.....	371
Макрос для обобщенных типов.....	372
Unicode.....	372
Анонимные структуры и анонимные объединения.....	378
Многопоточность .....	380
Немного о C18 .....	380
Резюме.....	380
<b>Глава 13. Конкуренентность .....</b>	<b>381</b>
Введение в конкурентность .....	381
Параллелизм .....	383
Конкуренентность .....	384
Планировщик заданий.....	385
Процессы и потоки.....	387
Порядок выполнения инструкций .....	388
Когда следует использовать конкурентность.....	390
Разделяемые состояния.....	397
Резюме.....	402
<b>Глава 14. Синхронизация .....</b>	<b>404</b>
Проблемы с конкурентностью .....	404
Естественные проблемы с конкурентностью .....	406
Постсинхронизационные проблемы.....	416
Методы синхронизации .....	417
Холостые циклы и циклические блокировки .....	418
Механизм ожидания/уведомления .....	421
Семафоры и мьютексы.....	424
Системы с несколькими вычислительными блоками.....	429
Циклические блокировки.....	434
Условные переменные .....	436
Конкуренентность в POSIX.....	438
Ядра с поддержкой конкурентности .....	438
Многопроцессность .....	440
Многопоточность .....	443
Резюме.....	444
<b>Глава 15. Многопоточное выполнение .....</b>	<b>446</b>
Потоки.....	447
POSIX-потоки .....	450
Порождение POSIX-потоков .....	452
Пример состояния гонки .....	457
Пример гонки данных.....	465
Резюме.....	468

<b>Глава 16. Синхронизация потоков</b> .....	469
Управление конкурентностью в POSIX .....	470
POSIX-мьютексы.....	470
Условные переменные POSIX .....	473
POSIX-барьеры .....	477
POSIX-семафоры.....	480
POSIX-потоки и память.....	488
Сегмент стека .....	488
Сегмент кучи .....	493
Видимость памяти .....	498
Резюме .....	500
<b>Глава 17. Процессы</b> .....	501
API для выполнения процессов .....	501
Создание процесса.....	504
Выполнение процесса.....	509
Разные методы создания и выполнения процессов.....	512
Этапы выполнения процесса.....	512
Разделяемые состояния.....	513
Методы разделения ресурсов .....	514
Разделяемая память в POSIX.....	516
Файловая система.....	526
Сравнение многопоточности и многопроцессности .....	528
Многопоточность.....	528
Локальная многопроцессность .....	529
Распределенная многопроцессность.....	530
Резюме .....	531
<b>Глава 18. Синхронизация процессов</b> .....	532
Локальное управление конкурентностью.....	533
Именованные POSIX-семафоры .....	534
Именованные мьютексы .....	538
Первый пример.....	538
Второй пример .....	542
Именованные условные переменные .....	552
Этап 1: класс разделяемой памяти.....	553
Этап 2: класс разделяемого 32-битного целочисленного счетчика .....	556
Этап 3: класс разделяемого мьютекса.....	558
Этап 4: класс разделяемой условной переменной.....	562
Этап 5: основная логика.....	565
Распределенное управление конкурентностью .....	570
Резюме .....	572

<b>Глава 19. Локальные сокеты и IPC</b> .....	573
Методы межпроцессного взаимодействия .....	574
Коммуникационные протоколы .....	576
Характеристики протоколов .....	578
Взаимодействие в рамках одного компьютера .....	581
Файловые дескрипторы .....	581
POSIX-сигналы .....	582
POSIX-каналы .....	586
Очереди сообщений POSIX .....	588
Сокеты домена Unix .....	591
Введение в программирование сокетов .....	592
Компьютерные сети .....	592
Что такое программирование сокетов .....	605
У сокетов есть собственные дескрипторы! .....	611
Резюме .....	612
<b>Глава 20. Программирование сокетов</b> .....	613
Краткий обзор программирования сокетов .....	614
Проект «Калькулятор» .....	616
Иерархия исходного кода .....	617
Сборка проекта .....	620
Запуск проекта .....	621
Прикладной протокол .....	622
Библиотека сериализации/десериализации .....	625
Сервис калькулятора .....	630
Сокеты домена Unix .....	632
Потоковый сервер на основе UDS .....	632
Потоковый клиент на основе UDS .....	640
Датаграммный сервер на основе UDS .....	643
Датаграммный клиент на основе UDS .....	647
Сетевые сокеты .....	649
TCP-сервер .....	650
TCP-клиент .....	651
UDP-сервер .....	652
UDP-клиент .....	653
Резюме .....	654
<b>Глава 21. Интеграция с другими языками</b> .....	655
Что делает интеграцию возможной .....	656
Получение необходимых материалов .....	657
Библиотека для работы со стекком .....	658
Интеграция с C++ .....	664

Декорирование имен в C++ .....	665
Код на C++ .....	667
Интеграция с Java.....	672
Написание кода на Java.....	672
Написание машинно-зависимой части .....	677
Интеграция с Python.....	685
Интеграция с Go .....	689
Резюме .....	691
<b>Глава 22. Модульное тестирование и отладка.....</b>	<b>693</b>
Тестирование программного обеспечения .....	694
Уровни тестирования .....	695
Модульное тестирование.....	696
Тестовые дублеры.....	704
Компонентное тестирование .....	706
Библиотеки тестирования для C .....	707
CMocka .....	708
Google Test.....	717
Отладка .....	721
Категории программных ошибок .....	722
Отладчики .....	723
Средства проверки памяти .....	725
Средства отладки потоков .....	726
Профилировщики производительности.....	727
Резюме .....	728
<b>Глава 23. Системы сборки .....</b>	<b>730</b>
Что такое система сборки .....	731
Make .....	732
CMake — не система сборки! .....	740
Ninja.....	744
Bazel.....	746
Сравнение систем сборки .....	749
Резюме .....	749
<b>Послесловие .....</b>	<b>751</b>



# 12 Последние нововведения в C

Прогресс не остановить. Язык программирования C является стандартом ISO, постоянно пересматриваемым в попытках сделать его лучше и привнести в него новые возможности. Но это не значит, что C становится проще; по мере развития языка в нем появляются новые и сложные концепции.

В этой главе я проведу краткий обзор новшеств C11. Как вы, наверное, знаете, стандарт C11 пришел на смену C99 и позже был заменен стандартом C18. Иными словами, C18 — самая последняя версия языка C, а C11 — предыдущая.

Интересно, что в C18 не появилось никаких новых возможностей; эта версия содержит лишь исправления ошибок, найденных в C11. Таким образом, все, что мы говорим о C11, фактически относится и к C18 — то есть к самому последнему стандарту C. Как видите, в C наблюдаются постоянные улучшения... вопреки мнению о том, что этот язык давно умер!

В данной главе будет представлен краткий обзор следующих тем:

- способы определения версии C и написания кода, совместимого с разными версиями этого языка;
- новые средства оптимизации и защиты исходного кода, такие как *невозвращаемые* функции и функции *с проверкой диапазона*;
- новые типы данных и методы компоновки памяти;
- функции с обобщенными типами;
- поддержка Unicode в C11, которой не хватало в предыдущих стандартах этого языка;
- анонимные структуры и объединения;
- встроенная поддержка многопоточности и методов синхронизации в C11.

Начнем эту главу с обсуждения стандарта C11 и его нововведений.

## C11

Разработка нового стандарта для технологии, которая используется на протяжении более 30 лет, — непростая задача. На C написаны миллионы (если не миллиарды!) строчек кода, и если вы хотите добавить новые возможности, то это нужно делать так, чтобы не затронуть существующий код. Новшества не должны создавать новые проблемы для имеющихся программ и не должны содержать ошибки. Такой взгляд на вещи может показаться идеалистическим, но это то, к чему нам следует стремиться.

Приведенный ниже PDF-документ находится на сайте *Open Standards* и выражает обеспокоенность и мысли участников сообщества C перед началом работы над C11: <http://www.open-std.org/JTC1/SC22/wg14/www/docs/n1250.pdf>. Его полезно почитать, поскольку в нем собран опыт разработки нового стандарта для языка, на котором уже было написано несколько тысяч проектов.

Мы будем рассматривать выпуск C11 с учетом всего вышесказанного. Будучи опубликованным впервые, стандарт C11 был далек от идеала и имел некоторые серьезные дефекты, со списком которых можно ознакомиться по адресу <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2244.htm>.

Через семь лет после выхода C11 был представлен стандарт C18, который должен был исправить недостатки предшественника. Стоит отметить, что C18 также *неофициально* называют C17, но это один и тот же стандарт. На странице, приведенной в ссылке выше, можно просмотреть перечень дефектов и их текущее состояние. Если состояние дефекта помечено как C17, то это значит, он был исправлен в рамках C18. Это показывает, насколько сложным и щепетильным может быть процесс формирования стандарта с таким большим количеством пользователей, как у языка C.

В следующих разделах речь пойдет о новых возможностях C11. Но прежде, чем мы по ним пройдемся, необходимо убедиться в том, что у нас есть компилятор, совместимый с данным стандартом. Об этом мы позаботимся в следующем разделе.

## Определение поддерживаемой версии стандарта C

На момент написания этих строк с момента выхода стандарта C11 прошло почти восемь лет. И потому было бы логично ожидать, что он уже поддерживается многими компиляторами. И это действительно так. Открытые компиляторы, такие как `gcc` и `clang`, имеют полную поддержку C11, но при необходимости могут переключаться на C99 и даже более ранние версии C. В данном разделе я покажу, как с помощью специального макроса определить версию C и в зависимости от нее использовать поддерживаемые возможности языка.

Если ваш компилятор поддерживает разные версии стандарта C, то первым делом нужно проверить, какая версия является текущей. Каждый стандарт C определяет

специальный макрос, позволяющий сделать это. До сих пор мы использовали `gcc` в Linux и `clang` в macOS. В `gcc 4.1 C11` предоставляется в качестве одного из поддерживаемых стандартов.

Рассмотрим следующий пример, чтобы понять, как на этапе выполнения узнать текущую версию стандарта C, используя уже определенный макрос (листинг 12.1).

**Листинг 12.1.** Определение версии стандарта C (`ExtremeC_examples_chapter12_1.c`)

```
#include <stdio.h>

int main(int argc, char** argv) {
    #if __STDC_VERSION__ >= 201710L
        printf("Hello World from C18!\n");
    #elif __STDC_VERSION__ >= 201112L
        printf("Hello World from C11!\n");
    #elif __STDC_VERSION__ >= 199901L
        printf("Hello World from C99!\n");
    #else
        printf("Hello World from C89/C90!\n");
    #endif
    return 0;
}
```

Как видите, данный код может различать разные версии стандарта C. Чтобы продемонстрировать, как разные версии приводят к разному выводу, скомпилируем этот исходный код несколько раз с применением разных стандартов C, поддерживаемых компилятором.

Чтобы заставить компилятор использовать определенный стандарт C, ему нужно передать параметр `-std=CXX`. Взгляните на следующую команду и на вывод, который она генерирует (терминал 12.1).

**Терминал 12.1.** Компиляция примера 12.1 с помощью разных версий стандарта C

```
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out
$ ./ex12_1.out
Hello World from C11!
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out -std=c11
$ ./ex12_1.out
Hello World from C11!
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out -std=c99
$ ./ex12_1.out
Hello World from C99!
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out -std=c90
$ ./ex12_1.out
Hello World from C89/C90!
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out -std=c89
$ ./ex12_1.out
Hello World from C89/C90!
$
```

Как видите, в новых компиляторах по умолчанию используется C11. В более старых версиях для включения C11 может понадобиться параметр `-std`. Обратите внимание на комментарии в начале файла. Я использовал многострочный формат, `/* ... */`, вместо однострочного, `//`. Дело в том, что однострочные комментарии не поддерживались в стандартах, предшествовавших C99. Поэтому пришлось сделать комментарии многострочными, чтобы код компилировался со всеми версиями C.

## Удаление функции `gets`

Из C11 была убрана знаменитая функция `gets`. Она была подвержена атакам с *переполнением буфера*, и в предыдущих версиях ее решили сделать *нерекомендуемой*. Позже она была удалена в рамках стандарта C11. Следовательно, старый исходный код, в котором используется эта функция, нельзя скомпилировать с помощью C11.

Вместо `gets` можно использовать функцию `fgets`. Вот отрывок из справочной страницы `gets` в macOS.



### Соображения безопасности

Функция `gets()` не подходит для безопасного использования. Ввиду отсутствия проверки диапазона и неспособности вызывающей программы надежно определить длину следующей входной строки, применение этой функции позволяет недобросовестным пользователям вносить произвольные изменения в функциональность запущенной программы с помощью атаки переполнения буфера. В любых ситуациях настоятельно рекомендуется задействовать функцию `fgets()` (см. FSA).

## Изменения в функции `fopen`

Функция `fopen` обычно используется для открытия файла и возвращения его дескриптора. Понятие *файла* в Unix очень абстрактно и может не иметь ничего общего с файловой системой. Функция `fopen` имеет следующие сигнатуры (листинг 12.2).

**Листинг 12.2.** Различные сигнатуры функций семейства `fopen`

```
FILE* fopen(const char *pathname, const char *mode);
FILE* fdopen(int fd, const char *mode);
FILE* freopen(const char *pathname, const char *mode, FILE *stream);
```

Все сигнатуры, приведенные выше, принимают входной параметр `mode`. Это строка, которая определяет режим открытия файла. В терминале 12.2 приведено описание,

взятое из справочной страницы `fopen` в FreeBSD. В нем объясняется, как следует использовать `mode`.

**Терминал 12.2.** Отрывок из справочной страницы `fopen` в FreeBSD

```
$ man 3 fopen
...
The argument mode points to a string beginning with one of the following letters:

    "r"    Open for reading. The stream is positioned at the beginning
           of the file. Fail if the file does not exist.

    "w"    Open for writing. The stream is positioned at the beginning
           of the file. Create the file if it does not exist.

    "a"    Open for writing. The stream is positioned at the end of
           the file. Subsequent writes to the file will always end up
           at the then current end of file, irrespective of
           any intervening fseek(3) or similar. Create the file
           if it does not exist.

An optional "+" following "r", "w", or "a" opens the file
for both reading and writing. An optional "x" following "w" or
"w+" causes the fopen() call to fail if the file already exists.
An optional "e" following the above causes the fopen() call to set
the FD_CLOEXEC flag on the underlying file descriptor.
The mode string can also include the letter "b" after either
the "+" or the first letter.

...
$
```

Режим `x`, описанный в данном отрывке, был представлен вместе со стандартом C11. Чтобы открыть файл для записи, функции `fopen` нужно передать режим `w` или `w+`. Но проблема вот в чем: если файл уже существует, то режимы `w` и `w+` сделают его пустым.

Поэтому если программист хочет добавить что-то в файл, не стирая имеющееся содержимое, то должен задействовать другой режим, `a`. Следовательно, перед вызовом `fopen` ему нужно проверить существование файла, используя API файловой системы, такой как `stat`, и затем выбрать подходящий режим в зависимости от результата. Но теперь программист может сначала попробовать режим `wx` или `wx+`, и если файл уже существует, то `fopen` вернет ошибку. После этого можно продолжить, применяя режим `a`.

Таким образом, открытие файла требует меньше шаблонного кода, поскольку нам больше не нужно проверять его существование с помощью API файловой системы. Теперь файл можно открыть в любом режиме, используя одну лишь функцию `fopen`.

В С11 также появился API `fopen_s`. Это безопасная версия `fopen`. Согласно документации, которая находится по ссылке <https://en.cppreference.com/w/c/io/fopen>, функция `fopen_s` выполняет дополнительную проверку предоставленных ей буферов и их границ, что позволяет обнаружить любые несоответствия.

## Функции с проверкой диапазона

Программам на языке С, которые работают с массивами строк и байтов, присуща одна серьезная проблема: они могут легко выйти за пределы диапазона, определенного для буфера или байтового массива.

Напомню, буфер — область памяти, которая служит для хранения массива байтов или строковой переменной. Выход за ее границы приводит к *переполнению буфера*, чем могут воспользоваться злоумышленники, чтобы организовать атаку (которую обычно называют *атакой переполнения буфера*). Это приводит либо к *отказу в обслуживании* (denial of service, DoS), либо к *эксплуатации* атакуемой программы.

Большинство таких атак обычно начинаются с функции, которая работает с массивами символов или байтов. В число *уязвимых* попадают функции обработки строк наподобие `strcpy` и `strcat`, находящиеся в `string.h`. У них нет механизмов проверки границ, которые могли бы предотвратить атаки переполнения буфера.

Однако в С11 появился новый набор функций *с проверкой диапазона*. Они имеют те же имена, что и функции для работы со строками, но с суффиксом `_s` в конце. Он означает, что они являются *безопасной* (secure) разновидностью традиционных функций и проводят дополнительные проверки на этапе выполнения, защищаясь от уязвимостей. Среди функций с проверкой диапазона, появившихся в С11, можно выделить `strcpy_s` и `strcat_s`.

Эти функции принимают дополнительные аргументы для входных буферов, которые исключают выполнение опасных операций. Например, функция `strcpy_s` имеет следующую сигнатуру (листинг 12.3).

**Листинг 12.3.** Сигнатура функции `strcpy_s`

```
errno_t strcpy_s(char *restrict dest, rsize_t destsz, const char *restrict src);
```

Как видите, второй аргумент — длина буфера `dest`. С его помощью функция проводит определенные проверки на этапе выполнения; например, она убеждается в том, что строка `src` не длиннее буфера `dest`, предотвращая тем самым запись в невыделенную память.