

Николай Иванов

**Самоучитель
Программирование
в Linux**

2-е издание

Санкт-Петербург
«БХВ-Петербург»

2012

УДК 681.3.06
ББК 32.973.26-018.2
И20

Иванов Н. Н.

И20 Программирование в Linux. Самоучитель. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2012. — 400 с.: ил.

ISBN 978-5-9775-0744-8

Рассмотрены фундаментальные основы программирования в Linux: инструментарий, низкоуровневый ввод-вывод, многозадачность, файловая система, межпроцессное взаимодействие и обработка ошибок. Книга главным образом ориентирована на практическое применение изложенных концепций. В ней есть все, что нужно начинающим, а углубленное изучение каждой темы делает ее ценной и для опытных программистов. Каждая тема проиллюстрирована большим числом примеров на языках C и C++ и Python, которые читатель сможет использовать в качестве образцов для собственных программ. На FTP-сервере издательства находятся исходные тексты программ.

Во втором издании материал актуализирован с учетом современных тенденций, добавлены 3 новые главы по программированию в Linux на языке Python, устранены замеченные ошибки.

Для начинающих и опытных Linux-программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Евгений Рыбаков</i>
Зав. редакцией	<i>Григорий Добин</i>
Редактор	<i>Елена Кашилакова</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Дизайн серии	<i>Инны Тачиной</i>
Оформление обложки	<i>Елены Беляевой</i>
Зав. производством	<i>Николай Тверских</i>

Подписано в печать 05.10.11.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 32,25.

Тираж 1200 экз. Заказ №

"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0744-8

© Иванов Н. Н., 2011
© Оформление, издательство "БХВ-Петербург", 2011

Оглавление

Введение	9
Предисловие.....	9
Условные обозначения.....	9
Примеры программ.....	10
Благодарности.....	10
Обратная связь.....	10
ЧАСТЬ I. ОСНОВЫ ПРОГРАММИРОВАНИЯ В LINUX	13
Глава 1. Создание программы	15
1.1. Исходный код.....	15
1.2. Компиляция.....	17
1.3. Компоновка.....	18
1.4. Многофайловые проекты.....	19
Глава 2. Автосборка	23
2.1. Обзор средств автосборки в Linux.....	23
2.2. Утилита make.....	25
2.3. Базовый синтаксис Makefile.....	25
2.4. Константы make.....	28
2.5. Рекурсивный вызов make.....	31
2.6. Получение дополнительной информации.....	36
Глава 3. Окружение	37
3.1. Понятие окружения.....	37
3.2. Чтение окружения: <i>environ</i> , <i>getenv()</i>	39
3.3. Модификация окружения: <i>setenv()</i> , <i>putenv()</i> , <i>unsetenv()</i>	41
3.4. Очистка окружения.....	45
Глава 4. Библиотеки	46
4.1. Библиотеки и заголовочные файлы.....	46
4.2. Подключение библиотек.....	47
4.3. Создание статических библиотек.....	48
4.4. Создание совместно используемых библиотек.....	52
4.5. Взаимодействие библиотек.....	55

Глава 5. Аргументы и опции программы	58
5.1. Аргументы программы.....	58
5.2. Использование опций	60
5.3. Использование длинных опций	63
5.4. Получение дополнительной информации.....	65
ЧАСТЬ II. НИЗКОУРОВНЕВЫЙ ВВОД-ВЫВОД В LINUX.....	67
Глава 6. Концепция ввода-вывода в Linux.....	69
6.1. Библиотечные механизмы ввода-вывода языка C.....	69
6.2. Концепция низкоуровневого ввода-вывода.....	73
6.3. Консольный ввод-вывод.....	74
6.4. Ввод-вывод в C++	75
Глава 7. Базовые операции ввода-вывода.....	78
7.1. Создание файла: <i>creat()</i>	78
7.2. Открытие файла: <i>open()</i>	82
7.3. Закрытие файла: <i>close()</i>	86
7.4. Чтение файла: <i>read()</i>	88
7.5. Запись файла: <i>write()</i>	91
7.6. Произвольный доступ: <i>lseek()</i>	94
Глава 8. Расширенные возможности ввода-вывода в Linux.....	104
8.1. Взаимодействие с библиотечными механизмами	104
8.2. Векторное чтение: <i>readv()</i>	108
8.3. Векторная запись: <i>writenv()</i>	111
8.4. Концепция "черных дыр"	114
ЧАСТЬ III. МНОГОЗАДАЧНОСТЬ	119
Глава 9. Основы многозадачности в Linux.....	121
9.1. Библиотечный подход: <i>system()</i>	121
9.2. Процессы в Linux	123
9.3. Дерево процессов.....	126
9.4. Получение информации о процессе	127
Глава 10. Базовая многозадачность	131
10.1. Концепция развилки: <i>fork()</i>	131
10.2. Передача управления: <i>execve()</i>	134
10.3. Семейство <i>exec()</i>	140
10.4. Ожидание процесса: <i>wait()</i>	147
Глава 11. Потoki.....	153
11.1. Концепция потоков в Linux.....	153
11.2. Создание потока: <i>pthread_create()</i>	155
11.3. Завершение потока: <i>pthread_exit()</i>	160
11.4. Ожидание потока: <i>pthread_join()</i>	161
11.5. Получение информации о потоке: <i>pthread_self()</i> , <i>pthread_equal()</i>	165
11.6. Отмена потока: <i>pthread_cancel()</i>	167
11.7. Получение дополнительной информации.....	169

Глава 12. Расширенная многозадачность	171
12.1. Уступчивость процесса: <i>nice()</i>	171
12.2. Семейство <i>wait()</i>	174
12.3. Зомби.....	178
ЧАСТЬ IV. ФАЙЛОВАЯ СИСТЕМА	181
Глава 13. Обзор файловой системы в Linux	183
13.1. Аксиоматика файловой системы в Linux	183
13.2. Типы файлов.....	184
13.3. Права доступа.....	186
13.4. Служебные файловые системы.....	188
13.5. Устройства.....	189
13.6. Монтирование файловых систем.....	191
Глава 14. Чтение информации о файловой системе.....	192
14.1. Семейство <i>statvfs()</i>	192
14.2. Текущий каталог: <i>getcwd()</i>	196
14.3. Получение дополнительной информации.....	199
Глава 15. Чтение каталогов	200
15.1. Смена текущего каталога: <i>chdir()</i>	200
15.2. Открытие и закрытие каталога: <i>opendir()</i> , <i>closedir()</i>	203
15.3. Чтение каталога: <i>readdir()</i>	204
15.4. Повторное чтение каталога: <i>rewinddir()</i>	205
15.5. Получение данных о файлах: семейство <i>stat()</i>	206
15.6. Чтение ссылок: <i>readlink()</i>	213
Глава 16. Операции над файлами.....	217
16.1. Удаление файла: <i>unlink()</i>	217
16.2. Перемещение файла: <i>rename()</i>	224
16.3. Создание ссылок: <i>link()</i>	226
16.4. Создание каталога: <i>mkdir()</i>	228
16.5. Удаление каталога: <i>rmdir()</i>	232
Глава 17. Права доступа	234
17.1. Смена владельца: <i>chown()</i>	234
17.2. Смена прав доступа: семейство <i>chmod()</i>	234
Глава 18. Временные файлы	243
18.1. Концепция использования временных файлов.....	243
18.2. Создание временного файла: <i>mkstemp()</i>	244
18.3. Закрытие и удаление временного файла	244
ЧАСТЬ V. МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ	251
Глава 19. Обзор методов межпроцессного взаимодействия в Linux.....	253
19.1. Общие сведения о межпроцессном взаимодействии в Linux.....	253
19.2. Локальные методы межпроцессного взаимодействия.....	254
19.3. Удаленное межпроцессное взаимодействие.....	258

Глава 20. Сигналы	260
20.1. Понятие сигнала в Linux	260
20.2. Отправка сигнала: <i>kill()</i>	262
20.3. Обработка сигнала: <i>sigaction()</i>	264
20.4. Сигналы и многозадачность	265
20.5. Получение дополнительной информации	269
Глава 21. Использование общей памяти	270
21.1. Выделение памяти: <i>shmget()</i>	270
21.2. Активизация совместного доступа: <i>shmat()</i>	271
21.3. Отключение совместного доступа: <i>shmdt()</i>	271
21.4. Контроль использования памяти: <i>shmctl()</i>	272
21.5. Использование семафоров	275
21.6. Контроль за семафорами: <i>semctl()</i>	277
Глава 22. Использование общих файлов	281
22.1. Размещение файла в памяти: <i>mmap()</i>	281
22.2. Освобождение памяти: <i>munmap()</i>	282
22.3. Синхронизация: <i>msync()</i>	283
Глава 23. Каналы	287
23.1. Создание канала: <i>pipe()</i>	287
23.2. Перенаправление ввода-вывода: <i>dup2()</i>	290
23.3. Получение дополнительной информации	294
Глава 24. Именованные каналы FIFO	295
24.1. Создание именованного канала	295
24.2. Чтение, запись и закрытие FIFO	296
Глава 25. Сокеты	299
25.1. Типы сокетов	299
25.2. Создание и удаление сокетов	300
25.3. Назначение адреса: <i>bind()</i>	301
25.4. Соединение сокетов: <i>connect()</i>	304
25.5. Прослушивание сокета: <i>listen()</i>	306
25.6. Принятие запроса на подключение: <i>accept()</i>	306
25.7. Прием и передача данных через сокет	310
25.8. Получение дополнительной информации	313
ЧАСТЬ VI. РАБОТА НАД ОШИБКАМИ И ОТЛАДКА	315
Глава 26. Выявление и обработка ошибок	317
26.1. Типы ошибок	317
26.2. Сообщения об ошибках	320
26.3. Макрос <i>assert()</i>	321
Глава 27. Ошибки системных вызовов	325
27.1. Чтение ошибки: <i>errno</i>	325
27.2. Сообщение об ошибке: <i>strerror()</i> , <i>perror()</i>	327

Глава 28. Использование отладчика gdb	330
28.1. Добавление отладочной информации	330
28.2. Запуск отладчика.....	331
28.3. Трансляция программы под отладчиком	334
28.4. Точки останова.....	340
28.5. Получение дополнительной информации.....	344
ЧАСТЬ VII. ПРОГРАММИРОВАНИЕ В LINUX НА ЯЗЫКЕ PYTHON	345
Глава 29. Язык Python	347
29.1. Несколько слов о языке Python.....	347
29.2. Инструментарий.....	349
29.3. Первая программа.....	350
29.4. Структура программы	352
Глава 30. Типы данных	356
30.1. Переменные	357
30.2. Целые числа.....	358
30.3. Числа с плавающей точкой	360
30.4. Строки.....	362
30.5. Списки.....	366
Глава 31. Программирование на языке Python	369
31.1. Логические операции.....	369
31.2. Сообщения об ошибках	371
31.3. Ветвления	373
31.4. Циклы.....	375
31.5. Функции.....	378
ПРИЛОЖЕНИЯ	379
Приложение 1. Именованные константы	381
Приложение 2. Коды ошибок системных вызовов.....	383
Приложение 3. Сигналы Linux.....	386
Приложение 4. Примеры программ	388
Предметный указатель	395

Введение

Предисловие

В современном мире операционная система Linux играет особую роль. Некогда считалось, что "внутренности" операционных систем могут быть доступны и понятны только особому кругу избранных. Благодаря Linux, созданной тысячами увлеченных программистов-добровольцев, мир увидел обратную сторону медали. Теперь каждый желающий может свободно изучать или изменять исходный код операционной системы. Если вы тоже желаете взглянуть на Linux изнутри, то надеюсь, что эта книга станет вашим верным помощником.

Эта книга дает базовые универсальные знания и умения, не связанные с конкретными областями программирования, но необходимые любому Linux-программисту в равной степени. Цель книги — донести до читателя понимание концепций файлов и процессов в Linux, научить самостоятельно работать с данной системой и развивать ее.

Условные обозначения

Новые термины, а также слова и фразы, на которые следует обратить особое внимание, выделяются в книге *курсивом*. Адреса сети Интернет выделяются жирным шрифтом, например: **<http://www.kernel.org>**. Элементы листингов (имена функций и переменных, типы данных, именованные константы и т. д.) помечаются в основном тексте пропорциональным шрифтом.

В примерах интерактивной работы с оболочкой символ `§` (доллар) обозначает приглашение командной строки:

```
§ uname  
Linux
```

Листинги программ представлены в виде обособленных блоков.

Листинг 0.1. Пример helloworld.c

```
#include <stdio.h>

int main (void)
{
    printf ("Hello World!\n");
    return 0;
}
```

Примеры программ

Исходные тексты рассмотренных в книге программ на языках C, C++ и Python можно скачать по ссылке [ftp://85.249.45.166/ 9785977507448.zip](ftp://85.249.45.166/9785977507448.zip), а также на странице книги на сайте www.bhv.ru.

Для отыскания файла, соответствующего конкретному листингу, используйте вспомогательную табл. П4.1 в *приложении 4*.

Благодарности

Хочу выразить глубокую признательность всем, кто помогал мне в работе над этой книгой. Это, в первую очередь, мои родные и близкие, которые с пониманием относились к непрерывному стуку клавиш и всячески поддерживали меня: жена Аня, мама Юлия Петровна, теща Татьяна Германовна. Еще хочу поблагодарить моих друзей Диму К., Надю К., Иру С., Сашу М., Иру Х. и Максима Ш., которые время от времени напоминали мне о том, что жизнь состоит не только из работы.

Отдельное спасибо руководителю проекта издательства "БХВ-Петербург" Евгению Рыбакову, предложившему мне написать эту книгу, терпеливо отвечавшему на мои "глупые" вопросы и оказавшему помощь в решении некоторых проблем технического характера.

Благодарю парней из форума LinuxForum.com, которые помогли мне в отладке примера на C++. Спасибо Ричарду Столлману, основателю проекта GNU, Линусу Торвальдсу, написавшему ядро Linux, и создателям текстового процессора OpenOffice.org Writer, в котором была написана эта книга.

Если вы не знакомы с языком C++, но хотите на нем программировать, то лучше всего начать с книги "Самоучитель C++" Герберта Шилдта, признанного автора книг по языку C++.

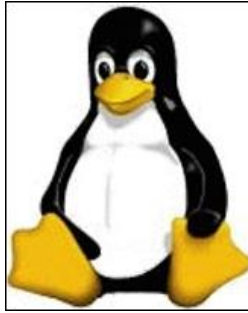
Обратная связь

Вы держите в руках второе издание книги, в котором были исправлены некоторые ошибки, а материал был переработан в соответствии с современным положением дел в области программирования для Linux. Кроме того, книга была дополнена но-

вой частью с главами, посвященными программированию в Linux на языке Python. В первых 28 главах книги мы будем писать программы на "родном" для Linux и проверенном годами языке C, а в *главах 29—31* опробуем свои силы в языке Python.

Много лет назад я узнал об одном из вечных законов программирования, согласно которому даже в самой простой и тщательно отлаженной программе имеется хотя бы одна ошибка. Мой опыт и опыт моих коллег доказывает, что это действительно так. Поэтому я прошу вас, уважаемый читатель, не верить в мою безупречность и сообщать мне об ошибках и опечатках, допущенных в книге. Это поможет сделать последующие переиздания лучше и качественнее.

Я не люблю критику, но с удовольствием выслушаю ваши идеи, пожелания и дополнения, относящиеся к этой книге. Пишите мне по адресу **nnivanov@mail.ru**. Я постараюсь ответить каждому.

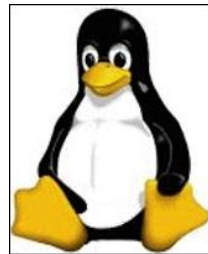


ЧАСТЬ I

Основы программирования в Linux

- Глава 1. Создание программы
- Глава 2. Автосборка
- Глава 3. Окружение
- Глава 4. Библиотеки
- Глава 5. Аргументы и опции программы

ГЛАВА 1



Создание программы

В начале книги рассматриваются технические вопросы создания программ в Linux. Процесс программирования обычно разделяют на несколько этапов, содержание которых определяется поставленной задачей. Прохождение каждого такого этапа требует наличия определенных инструментов, совокупный набор которых называется *инструментарием*. Эта глава описывает базовый инструментарий Linux-программиста, пишущего на языке C: *текстовый редактор*, *компилятор* и *компоновщик* — наиболее часто используемые инструменты создания программ. В последующих главах книги инструментарий будет постепенно дополняться новыми элементами.

Обычно программисты придерживаются некоторых общих приемов написания программ — *идиом программирования*. Описанная в *разд. 1.4* концепция создания многофайловых проектов — одна из таких идиом.

1.1. Исходный код

Создание любой программы начинается с постановки задачи, проектирования и написания исходного кода (source code). Обычно исходный код программы записывается в один или несколько файлов, которые называют *исходными файлами* или *исходниками*.

ПРИМЕЧАНИЕ

Иногда под словосочетанием "исходный код" понимают совокупность всех исходных файлов конкретного проекта (например, "исходный код ядра Linux").

Исходные файлы обычно создаются и набираются в текстовом редакторе. В принципе, для написания исходных кодов подойдет любой текстовый редактор. Но желательно, чтобы это был редактор с "подсветкой" синтаксиса, т. е. выделяющий визуально ключевые слова используемого языка программирования. В результате исходный код становится более наглядным, а программист делает меньше опечаток и ошибок.

В современных дистрибутивах Linux представлен большой выбор текстовых редакторов. Наибольшей популярностью среди программистов пользуются редакторы двух семейств:

- *vi* (Visual Interface) — полноэкранный редактор, созданный Биллом Джоем (Bill Joy) в 1976 г. С тех пор было написано немало клонов *vi*. Практически все Unix-подобные системы комплектуются той или иной версией этого текстового редактора. Наиболее популярные клоны *vi* в Linux — *vim* (Vi IMproved), *Elvis* и *nvi*;
- *Emacs* (Editor MACroS) — текстовый редактор, разработанный Ричардом Столлманом (Richard Stallman). Из всех существующих версий Emacs наиболее популярными являются GNU Emacs и XEmacs.

Среди других распространенных в Linux редакторов следует отметить *rico* (Pine COmposer), *jed* и *mcedit* (Midnight Commander EDITor). Они не обладают мощью *vi* или Emacs, но достаточно просты и удобны в использовании. В Linux также имеется множество текстовых редакторов с графическим интерфейсом: *kate*, *gedit*, *nedit*, *bluefish*, *jedit* (этот список можно продолжать очень долго). Редакторы *vim* и GNU Emacs тоже имеют собственные графические расширения.

Обычно программирование начинается с примера, выводящего на экран приветствие "Hello World!". Отступим от этой давней традиции и напишем сразу что-нибудь полезное, например программу часов.

Для начала создайте в своем текстовом редакторе файл `myclock.c` (листинг 1.1).

Листинг 1.1. Файл `myclock.c`

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

int main (void)
{
    time_t nt = time (NULL);
    printf ("%s", ctime (&nt));
    return 0;
}
```

Это исходный код нашей первой программы. Рассмотрим его по порядку:

1. Заголовочный файл `stdio.h` делает доступными механизмы ввода-вывода стандартной библиотеки языка C. Нам он нужен для вызова функции `printf()`.
2. Заголовочный файл `time.h` включается в программу, чтобы сделать доступными функции `time()` и `ctime()`, работающие с датой/временем.
3. Собственно программа начинается с функции `main()`, в теле которой создается переменная `nt`, имеющая тип `time_t`. Переменные этого типа предназначены для хранения числа секунд, прошедших с начала эпохи отсчета компьютерного времени (полночь 1 января 1970 г.).
4. Функция `time()` заносит в переменную `nt` текущее время.

5. Функция `ctime()` преобразовывает время, исчисляемое в секундах от начала эпохи (Epoch), в строку, содержащую привычную для нас запись даты и времени.
6. Полученная строка выводится на экран функцией `printf()`.
7. Инstrukция `return 0;` осуществляет выход из программы.

1.2. Компиляция

Чтобы запустить программу, ее необходимо сначала перевести с понятного человеку исходного кода в понятный компьютеру исполняемый код. Такой перевод называется *компиляцией* (compilation).

Чтобы откомпилировать программу, написанную на языке C, нужно "пропустить" ее исходный код через *компилятор*. В результате получается *исполняемый* (бинарный) код. Файл, содержащий исполняемый код, обычно называют *исполняемым файлом* или *бинарником* (binary).

ЗАМЕЧАНИЕ

Компилятор не всегда генерирует код, готовый к непосредственному выполнению. Эти случаи будут рассмотрены в *разд. 1.3*.

Компилятором языка C в Linux обычно служит программа `gcc` (GNU C Compiler) из пакета компиляторов GCC (GNU Compiler Collection). Чтобы откомпилировать нашу программу (листинг 1.1), следует вызвать `gcc`, указав в качестве аргумента имя исходного файла:

```
$ gcc myclock.c
```

Если компилятор не нашел ошибок в исходном коде, то в текущем каталоге появится файл `a.out`. Теперь, чтобы выполнить программу, требуется указать командной оболочке путь к исполняемому файлу. Поскольку текущий каталог обычно обозначается точкой, то запуск программы можно осуществить следующим образом:

```
$ ./a.out
```

```
Wed Nov 8 03:09:01 2006
```

Исполняемые файлы программ обычно располагаются в каталогах, имена которых перечислены через двоеточие в особой переменной `PATH`. Чтобы просмотреть содержимое этой переменной, введите следующую команду:

```
$ echo $PATH
```

```
/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/lib/qt4/bin
```

Если бинарник находится в одном из этих каталогов, то для запуска программы достаточно ввести ее имя (например, `ls`). В противном случае потребуется указание пути к исполняемому файлу.

Имя `a.out` не всегда подходит для программы. Один из способов исправить положение — просто переименовать полученный файл:

```
$ mv a.out myclock
```

Но есть способ лучше. Можно запустить компилятор с опцией `-o`, которая позволя-ет явно указать имя файла на выходе:

```
$ gcc -o myclock myclock.c
```

Наша программа не содержит синтаксических ошибок, поэтому компилятор молча "проглатывает" исходный код. Проведем эксперимент, нарочно испортив програм-му. Для этого уберем в исходном файле первую инструкцию функции `main()`, кото-рая объявляет переменную `nt`. Теперь снова попробуем откомпилировать получен-ный исходный код:

```
$ gcc -o myclock myclock.c
myclock.c: In function 'main':
myclock.c:6: error: 'nt' undeclared (first use in this function)
myclock.c:6: error: (Each undeclared identifier is reported only once
myclock.c:6: error: for each function it appears in.)
```

ЗАМЕЧАНИЕ

Комментарии используются программистами не только для пояснений и заметок. Пе-ред компиляцией комментарии исключаются из исходного кода, как если бы их вооб-ще не было. Поэтому программу `myclock` можно "испортить", просто закомментировав первую инструкцию функции `main()`.

Как и ожидалось, компиляция не удалась. Обратите внимание, что находящийся в текущем каталоге исполняемый файл `myclock` — это "детище" предыдущей компи-ляции. Новый файл не был создан.

ЗАМЕЧАНИЕ

Иногда говорят, что "компилятор ругается". Это программистский жаргон, означающий, что компиляция не удалась из-за ошибок в исходном коде.

Очень важно научиться понимать сообщения об ошибках, выводимых компилято-ром. В нашем случае сообщается, что произошло "нечто" в файле `myclock.c` внутри функции `main()`. Далее говорится, что строка номер 6 содержит ошибку (`error`): пе-ременная `nt` не была объявлена к моменту ее первого использования в данной функции. В последних двух строках приводится пояснение: для каждой функции, где встречается необъявленный идентификатор (имя), сообщение об ошибке выво-дится только один раз.

Иногда вместо ошибки (`error`) выдается предупреждение (`warning`). В этом случае компиляция не останавливается, но до сведения программиста доводится информа-ция о потенциально опасной конструкции исходного кода.

Теперь верните недостающую строку обратно или раскомментируйте, поскольку файл `myclock.c` нам еще понадобится.

1.3. Компоновка

В предыдущем разделе говорилось о том, что компилятор переводит исходный код программы в исполняемый. Но это не всегда так.

В достаточно объемных программах исходный код обычно разделяется для удобства на несколько частей, которые компилируются отдельно, а затем соединяются воедино. Каждый такой "кусочек" содержит *объектный код* и называется *объектным модулем*.

Объектные модули записываются в *объектные файлы*, имеющие расширение `.o`. В результате объединения объектных файлов могут получаться исполняемые файлы (обычные запускаемые бинарники), а также библиотеки, о которых пойдет речь в *главе 4*.

Для объединения объектных файлов служит *компоновщик (линковщик)*, а сам процесс называют *компоновкой* или *линковкой*. В Linux имеется компоновщик GNU ld, входящий в состав пакета GNU binutils.

ПРИМЕЧАНИЕ

Иногда компоновщик называют также *загрузчиком*.

Ручная компоновка объектных файлов — довольно неприятный процесс, требующий передачи программе ld большого числа параметров, зависящих от многих факторов. К счастью, компиляторы из коллекции GCC сами вызывают линковщик с нужными параметрами, когда это необходимо.

ПРИМЕЧАНИЕ

Программист может самостоятельно передавать компоновщику дополнительные параметры через компилятор. Эта возможность будет рассмотрена в *главе 4*.

Теперь вернемся к нашему примеру (листинг 1.1). В предыдущем разделе компилятор "молча" вызвал компоновщик, в результате чего получился исполняемый файл. Чтобы отказаться от автоматической компоновки, нужно передать компилятору опцию `-c`:

```
$ gcc -c myclock.c
```

Если компилятор не нашел ошибок, то в текущем каталоге должен появиться объектный файл `myclock.o`. Других объектных файлов у нас нет, поэтому будем компоновать только его. Это делается очень просто:

```
$ gcc -o myclock myclock.o
```

ЗАМЕЧАНИЕ

В UNIX существуют различные форматы объектных файлов. Наиболее популярные среди них — `a.out` (Assembler OUTPUT) и `COFF` (Common Object File Format). В Linux чаще всего встречается открытый формат объектных и исполняемых файлов `ELF` (Executable and Linkable Format).

1.4. Многофайловые проекты

Современные программные проекты редко ограничиваются одним исходным файлом. Распределение исходного кода программы на несколько файлов имеет ряд существенных преимуществ перед однофайловыми проектами.

- ❑ Использование нескольких исходных файлов накладывает на *репозиторий* (рабочий каталог проекта) определенную логическую структуру. Такой код легче читать и модернизировать.
- ❑ В однофайловых проектах любая модернизация исходного кода влечет повторную компиляцию всего проекта. В многофайловых проектах, напротив, достаточно откомпилировать только измененный файл, чтобы обновить проект. Это экономит массу времени.
- ❑ Многофайловые проекты позволяют реализовывать одну программу на разных языках программирования.
- ❑ Многофайловые проекты позволяют применять к различным частям программы разные лицензионные соглашения.

Обычно процесс сборки многофайлового проекта осуществляется по следующему алгоритму:

1. Создаются и подготавливаются исходные файлы. Здесь есть одно важное замечание: каждый файл должен быть целостным, т. е. не должен содержать незавершенных конструкций. Функции и структуры не должны разрываться. Если в рамках проекта предполагается создание исполняемой программы, то в одном из исходных файлов должна присутствовать функция `main()`.
2. Создаются и подготавливаются заголовочные файлы. У заголовочных файлов особая роль: они устанавливают соглашения по использованию общих идентификаторов (имен) в различных частях программы. Если, например, функция `func()` реализована в файле `a.c`, а вызывается в файле `b.c`, то в оба файла требуется включить директивой `#include` заголовочный файл, содержащий объявление (прототип) нашей функции. Технически можно обойтись и без заголовочных файлов, но в этом случае функцию можно будет вызвать с произвольными аргументами, и компилятор, за отсутствием соглашений, не выведет ни одной ошибки. Подобный "слепой" подход потенциально опасен и в большинстве случаев свидетельствует о плохом стиле программирования.
3. Каждый исходный файл отдельно компилируется с опцией `-c`. В результате получается набор объектных файлов.
4. Полученные объектные файлы соединяются компоновщиком в одну исполняемую программу.

Если необходимо скомпоновать несколько объектных файлов (`OBJ1.o`, `OBJ2.o` и т. д.), то применяют следующий простой шаблон:

```
$ gcc -o OUTPUT_FILE OBJ1.o OBJ2.o ...
```

Рассмотрим программу, которая принимает в качестве аргумента строку и переводит в ней все символы в верхний регистр, т. е. заменяет все строчные буквы на заглавные. Чтобы не усложнять пример, будем преобразовывать только латинские (англоязычные) символы.

Для начала создадим файл `print_up.h` (листинг 1.2), в котором будет находиться объявление (прототип) функции `print_up()`. Эта функция переводит символы строки в верхний регистр и выводит полученный результат на экран.

Листинг 1.2. Файл print_up.h

```
void print_up (const char * str);
```

Итак, объявление функции `print_up()` устанавливает соглашение, по которому любой исходный файл, включающий `print_up.h` директивой `#include`, обязан вызвать функцию `print_up()` с одним и только одним аргументом типа `const char*`. Теперь создадим файл `print_up.c` (листинг 1.3), в котором будет находиться тело функции `print_up()`.

Листинг 1.3. Файл print_up.c

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include "print_up.h"

void print_up (const char * str)
{
    int i;
    for (i = 0; i < strlen (str); i++)
        printf ("%c", toupper (str[i]));

    printf ("\n");
}
```

Функция `print_up()` просматривает в цикле всю строку, посимвольно преобразовывая ее в верхний регистр. Вывод также производится посимвольно. В заключение выводится символ новой строки. Функция `toupper()`, объявленная в файле `sture.h` и являющаяся частью стандартной библиотеки языка C, возвращает переданный ей символ в верхнем регистре, если это возможно. Без дополнительных манипуляций эта функция не работает с кириллическими (русскоязычными) символами, но сейчас это не важно. Теперь создадим третий файл `main.c` (листинг 1.4), который будет содержать функцию `main()`, необходимую для компоновки и запуска программы.

Листинг 1.4. Файл main.c

```
#include <string.h>
#include <stdio.h>
#include "print_up.h"

int main (int argc, char ** argv)
{
    if (argc < 2) {
        fprintf (stderr, "Wrong arguments\n");
        return 1;
    }
}
```

```

    print_up (argv[1]);
    return 0;
}

```

Сначала вспомним, что аргументы командной строки передаются в программу через функцию `main()`:

- `argc` — целое число, содержащее количество аргументов командной строки;
- `argv` — массив строк (двумерный массив символов), в котором находятся аргументы.

Следует помнить, что в первый аргумент командной строки обычно заносится имя программы. Таким образом, `argv[0]` — это имя программы, `argv[1]` — первый переданный при запуске аргумент, `argv[2]` — второй аргумент и т. д. до элемента `argv[argc-1]`.

Вообще говоря, аргументы в программу можно передавать не только из командной оболочки. Поэтому понятие "аргументы командной строки" не всегда отражает действительное положение вещей. В связи с этим, чтобы избежать неоднозначности, будем в дальнейшем пользоваться более точной формулировкой "аргументы программы".

ПРИМЕЧАНИЕ

Об аргументах программы и способах их обработки будет подробно рассказано в *главе 5*.

Теперь нужно собрать проект воедино. Сначала откомпилируем каждый файл с расширением `.c`:

```

$ gcc -c print_up.c
$ gcc -c main.c

```

В результате компиляции в репозитории программы должны появиться объектные файлы `print_up.o` и `main.o`, которые следует скомпоновать в один бинарный файл:

```

$ gcc -o printup print_up.o main.o

```

ПРИМЕЧАНИЕ

Если компилятор `gcc` вызывается с опцией `-c`, но без опции `-o`, то имя выходного файла получается заменой расширения `.c` на `.o`. Например, из файла `foo.c` получается файл `foo.o`.

Осталось только запустить и протестировать программу:

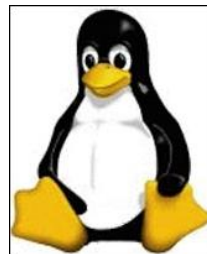
```

$ ./printup
Wrong arguments
$ ./printup Hello
HELLO

```

Обратите внимание на то, что заголовочный файл `print_up.h` не компилируется. Заголовочные файлы вообще никогда отдельно не компилируются. Дело в том, что на стадии *препроцессирования* (условно первая стадия компиляции) все директивы `#include` заменяются на содержимое указанных в них файлов.

ГЛАВА 2



Автосборка

Сборкой называется процесс подготовки программы к непосредственному использованию. Простейший пример сборки — компиляция и компоновка. Более сложные проекты могут также включать в себя дополнительные промежуточные этапы (операции над файлами, конфигурирование и т. п.). Существуют также языки программирования, позволяющие запускать программы сразу после подготовки исходного кода, минуя стадию сборки. Примером такого языка является Python, о котором речь пойдет в *главах 29—31*.

В этой главе описывается инструментарий для автоматической сборки программных проектов в Linux, написанных на языках семейства C/C++. Отдельно рассмотрены некоторые идиомы, связанные с процессом автосборки.

2.1. Обзор средств автосборки в Linux

В предыдущей главе рассматривался простейший многофайловый проект. Для его сборки нам приходилось сначала компилировать каждый исходник, а затем компоновать полученные объектные файлы в единый бинарник.

Собирать программы вручную неудобно, поэтому программисты, как правило, прибегают к различным приемам, позволяющим автоматизировать этот процесс. Самый простой способ — написать сценарий оболочки (shell-скрипт), который будет автоматически выполнять все то, что вы обычно вводите вручную. Тогда многофайловый проект из предыдущей главы можно собрать, например, при помощи скрипта, приведенного в листинге 2.1.

Листинг 2.1. Скрипт `make_printup`

```
#!/bin/sh
gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

ПРИМЕЧАНИЕ

Любая командная оболочка является также интерпретатором собственного языка программирования. В результате ей можно передавать набор команд в виде одного файла. Подобные файлы называются *скриптами (или сценариями) оболочки*. Каждый такой сценарий начинается с последовательности символов `#!` (решетка и восклицательный знак), после которой следует (без пробела) полный путь к исполняемому файлу оболочки, под которой будет выполняться скрипт. Такую строку называют `shebang` или `hashbang`. В Linux ссылка `/bin/sh` обычно указывает на оболочку `bash`.

Теперь файлу `make_printup` необходимо дать права на выполнение:

```
$ chmod +x make_printup
```

ПРИМЕЧАНИЕ

О правах доступа будет подробно рассказано в *главах 7 и 13*.

Осталось только вызвать скрипт, и проект будет создан:

```
$ ./make_printup
```

На первый взгляд, все прекрасно. Но настоящий программист должен предвидеть все возможные проблемы, и при детальном рассмотрении перспективы использования скрипта оболочки для сборки проекта уже не кажутся такими радужными. Перечислим некоторые проблемы.

- ❑ Скрипты оболочки статичны. Их работа не зависит от состояния текущей задачи. Даже если нужно заново откомпилировать только один файл, скрипт будет собирать проект "с нуля".
- ❑ В скриптах плохо просматриваются связи между различными элементами проекта.
- ❑ Скрипты не обладают возможностью самодиагностики.

К счастью, Linux имеет в наличии достаточно большой арсенал специализированных средств для автоматической сборки программных проектов. Такие средства называют *автосборщиками* или *утилитами автоматической сборки*. Благодаря специализированным автосборщикам программист может сосредоточиться, собственно, на программировании, а не на процессе сборки.

Все утилиты автосборки в Linux можно разделить на несколько условных категорий.

- ❑ Семейство `make` — это различные реализации стандартного для Unix-подобных систем автосборщика `make`. Из представителей данного семейства наибольшей популярностью в Linux пользуются утилиты GNU `make`, `imake`, `pmake`, `smake`, `fastmake` и `tmake`.
- ❑ Надстройки над `make` — утилиты семейства `make` работают с особыми файлами, в которых содержится вся информация о сборке проекта. Такие файлы называют `make-файлами` (`makefiles`). При работе с классическими `make-утилитами` эти файлы создаются и редактируются вручную. Надстройки над `make` автоматизируют процесс создания `make-файлов`. Наиболее популярные представители этого семейства в Linux — утилиты из пакета GNU Autotools (`automake`, `autoconf`, `libtool` и т. д.).

- Специализированные автосборщики — обычно такие утилиты создаются для удобства при работе с определенными проектами. Типичный представитель этого семейства — утилита `qmake` (`Qt make`) — автосборщик для проектов, использующих библиотеку `Qt`. Библиотека `Qt` была создана норвежской компанией `Trolltech`, но в настоящее время принадлежит корпорации `Nokia`.

Изучить каждый из перечисленных автосборщиков в рамках данной книги не представляется возможным. Поэтому мы будем пользоваться самой популярной в `Linux` утилитой автосборки `GNU make`. Далее, говоря о `make`, будем подразумевать `GNU make`.

2.2. Утилита `make`

Утилита `make` (`GNU make`) — наиболее популярное и проверенное временем средство автоматической сборки программ в `Linux`. Даже "гигант" автосборки, пакет `GNU Autotools`, является лишь надстройкой над `make`. Автоматическая сборка программы на языке `C` обычно осуществляется по следующему алгоритму.

1. Подготавливаются исходные и заголовочные файлы.
2. Подготавливаются `make`-файлы, содержащие сведения о проекте. Порой даже крупные проекты обходятся одним `make`-файлом. Вообще говоря, `make`-файл может называться как угодно, однако обычно выбирают одно из трех стандартных имен (`Makefile`, `makefile` или `GNUmakefile`), которые распознаются автосборщиком автоматически.
3. Вызывается утилита `make`, которая собирает проект на основании данных, полученных из `make`-файла. Если в проекте используется нестандартное имя `make`-файла, то его нужно указать после опции `-f` при вызове автосборщика.

На протяжении всей книги, чтобы не путаться, для `make`-файлов мы будем указывать имя `Makefile`.

ЗАМЕЧАНИЕ

Разработчики `GNU make` рекомендуют использовать имя `Makefile`. В этом случае у вас больше шансов, что `make`-файл будет стоять обособленно в отсортированном списке содержимого репозитория.

2.3. Базовый синтаксис `Makefile`

Итак, чтобы работать с `make`, необходимо создать файл с именем `Makefile`. В `make`-файлах могут присутствовать следующие конструкции:

- *Комментарии.* В `make`-файлах допустимы однострочные комментарии, которые начинаются символом `#` (решетка) и действуют до конца строки.
- *Объявления констант.* Константы в `make`-файлах служат для подстановки. Они во многом схожи с константами препроцессора языка `C`.

- *Целевые связи.* Эти элементы несут основную нагрузку в make-файле. При помощи целевых связей задаются зависимости между различными частями программы, а также определяются действия, которые будут выполняться при сборке программы. В любом make-файле должна быть хотя бы одна целевая связь.

Для правильного составления make-файла необходимо определить основную цель сборки проекта. Затем следует выявить промежуточные цели, если таковые существуют. Вернемся к примеру из предыдущей главы (см. листинги 1.2—1.4). В нем основной целью является формирование бинарного файла `printup`. Чтобы его получить, требуются файлы `print_up.o` и `main.o`. Это *промежуточные цели*. В make-файлах за каждую цель отвечает своя целевая связь.

После определения целей нужно выявить зависимости. В нашем примере основная цель (файл `printup`) может быть достигнута только при наличии файлов `print_up.o` и `main.o`. А файл `print_up.o` может быть получен только при наличии исходного файла `print_up.c` (см. листинг 1.3) и заголовочного файла `print_up.h` (см. листинг 1.2). Аналогичным образом файл `main.o` может быть получен только при наличии файлов `main.c` (см. листинг 1.4) и `print_up.h` (см. листинг 1.2).

Итак, мы знаем, что в Makefile обязательны только целевые связи. Каждая целевая связь состоит из следующих компонентов:

- *Имя цели.* Если целью является файл, то указывается его имя. После имени цели следует двоеточие.
- *Список зависимостей.* Здесь просто перечисляются через пробел имена файлов или имена промежуточных целей. Если цель ни от чего не зависит, то этот список будет пустым.
- *Инструкции.* Это команды, которые должны выполняться для достижения цели. Например, в целевой связке `print_up.o` инструкцией будет являться команда компиляции файла `print_up.c`. Каждая инструкция пишется на новой строке и начинается с символа табуляции. Обратите внимание, что некоторые текстовые редакторы (например, `mcedit`) по умолчанию заменяют табуляцию группой пробелов. В этом случае для редактирования Makefile следует воспользоваться другим редактором или настроить существующий. Иногда целевая связка не подразумевает выполнение каких-либо команд, а призвана только установить зависимости. В таком случае список инструкций оставляют пустым.

Теперь проверим систему в действии, организовав автосборку проекта `printup` из предыдущей главы. Сначала создаем Makefile (листинг 2.2).

Листинг 2.2. Make-файл программы `printup`

```
# Makefile for printup

printup: print_up.o main.o
    gcc -o printup print_up.o main.o

print_up.o: print_up.c print_up.h
    gcc -c print_up.c
```

```
main.o: main.c
    gcc -c main.c

clean:
    rm -f *.o
    rm -f printup
```

Далее вызываем утилиту `make` с указанием цели, которую нужно достичь. В нашем случае это будет выглядеть так:

```
$ make printup
gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

Итак, проект собран. Осталось только во всем разобраться. Первая строка в `Makefile` — это комментарий. Затем следуют целевые связи. Первая связь отвечает за создание исполняемого файла `printup` и формируется следующим образом:

1. Сначала записывается имя цели (`printup`).
2. После двоеточия перечисляются зависимости (`print_up.o` и `main.o`).
3. На следующей строке после знака табуляции пишется правило для получения бинарника `printup`.

Аналогичным образом оформляются остальные целевые связи. Последняя (`clean`) требует особого рассмотрения:

1. Сначала указывается имя цели (`clean`).
2. После двоеточия следует пустой список зависимостей. Это значит, что данная связь не требует наличия каких-либо файлов и не предполагает предварительного выполнения промежуточных целей.
3. На следующих двух строках прописаны инструкции, удаляющие объектные файлы и бинарник.

Эта цель очищает проект от всех файлов, автоматически созданных при сборке. Итак, чтобы очистить проект, достаточно набрать следующую команду:

```
$ make clean
rm -f *.o
rm -f printup
```

Очистка проекта обычно выполняется в следующих случаях:

- при подготовке исходного кода к отправке конечному пользователю или другому программисту, когда нужно избавиться от лишних файлов;
- при изменении или добавлении в проект заголовочных файлов;
- при изменении `make`-файла.

Вообще говоря, при запуске `make` имя цели можно не указывать. Тогда основной целью будет считаться первая цель в `Makefile`. Следовательно, в нашем случае, чтобы собрать проект, достаточно вызвать `make` без аргументов:

```
$ make
gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

Иногда требуется вписать в make-файл нечто длинное, например инструкцию, не уместящуюся в одной строке. В таком случае строки условно соединяются символом \ (обратная косая черта):

```
gcc -Wall -pedantic -g -o my_very_long_output_file one.o two.o \
three.o four.o five.o
```

Автосборщик при обработке make-файла будет интерпретировать такую конструкцию как единую строку.

2.4. Константы make

В make-файлах для параметризации процесса сборки можно использовать константы. Для объявления и инициализации констант предусмотрен следующий шаблон:

```
NAME=VALUE
```

Здесь `NAME` — это имя константы, `VALUE` — ее значение. Имя константы не должно начинаться с цифры. Значение может содержать любые символы, включая пробелы. Признак окончания значения константы — конец строки. Иначе говоря, любые символы, стоящие между знаком "равно" и символом переноса строки, будут являться значением константы.

Значение константы можно подставить в любую часть make-файла (кроме комментария). Если имя константы состоит из одного символа, то для подстановки достаточно добавить перед именем литеру `$` (доллар). Когда имя состоит из нескольких символов, для подстановки применяется следующий шаблон:

```
$(NAME)
```

Теперь модернизируем make-файл проекта `printup`, добавив в него константы (листинг 2.3).

Листинг 2.3. Make-файл программы `printup2`

```
CC=gcc
CLEAN=rm -f
PROGRAM_NAME=printup

$(PROGRAM_NAME): print_up.o main.o
    $(CC) -o $(PROGRAM_NAME) print_up.o main.o

print_up.o: print_up.c
    $(CC) -c print_up.c
```

```
main.o: main.c
    $(CC) -c main.c

clean:
    $(CLEAN) *.o
    $(CLEAN) $(PROGRAM_NAME)
```

Итак, мы заменили имя компилятора, команду удаления и имя конечной программы символическими именами. Теперь можно, например, сменить имя программы, просто изменив значение константы `PROGRAM_NAME`.

ПРИМЕЧАНИЕ

Файл `print_up.h` был исключен из списков зависимостей. При отсутствии заголовочного файла компилятор всегда сообщает об этом. Таким образом, включение `print_up.h` в список зависимостей хотя и не ошибочно, но явно избыточно.

Обратите внимание, что константы допустимы при объявлении и инициализации других констант. В результате наш Makefile можно значительно модернизировать (листинг 2.4).

Листинг 2.4. Make-файл программы `printup3`

```
CC=gcc
CLEAN=rm
CLEAN_FLAGS=-f
CLEAN_COMMAND=$(CLEAN) $(CLEAN_FLAGS)
PROGRAM_NAME=printup

$(PROGRAM_NAME): print_up.o main.o
    $(CC) -o $(PROGRAM_NAME) print_up.o main.o

print_up.o: print_up.c
    $(CC) -c print_up.c

main.o: main.c
    $(CC) -c main.c

clean:
    $(CLEAN_COMMAND) *.o
    $(CLEAN_COMMAND) $(PROGRAM_NAME)
```

На самом деле, объявляемые пользователем константы по существу не являются константами, поскольку их можно переопределять, т. е. повторно присваивать им значения. В этом легко убедиться, если слегка изменить предыдущий Makefile (листинг 2.5).

Листинг 2.5. Make-файл программы printup4

```
CC=gcc
CLEAN=some_value
PROGRAM_NAME=printup

$(PROGRAM_NAME): print_up.o main.o
    $(CC) -o $(PROGRAM_NAME) print_up.o main.o

print_up.o: print_up.c
    $(CC) -c print_up.c

main.o: main.c
    $(CC) -c main.c

CLEAN=rm -f

clean:
    $(CLEAN) *.o
    $(CLEAN) $(PROGRAM_NAME)
```

Из листинга 2.5 видно, что константа `CLEAN` изменяется. Тем не менее такое переопределение редко встречается на практике, поэтому понятие "константа" вполне пригодно.

Утилита `make` поддерживает также целый ряд специализированных констант. Две из них используются в целевых связках и представляют особый интерес:

- ❑ `$$` — содержит имя текущей цели;
- ❑ `$$^` — содержит список зависимостей в текущей связке.

Если теперь переписать `Makefile`, добавив в него эти две константы, то получится довольно симпатичный результат (листинг 2.6).

Листинг 2.6. Make-файл программы printup5

```
CC=gcc
CLEAN=rm -f
PROGRAM_NAME=printup

$(PROGRAM_NAME): print_up.o main.o
    $(CC) -o $$ $^

print_up.o: print_up.c
    $(CC) -c $^

main.o: main.c
    $(CC) -c $^
```