Операционные системы и системное программное обеспечение

Лабораторная работа №1

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.

Целью работы является знакомство с базовым инструментарием Linux-программиста, пишущего на языке C: текстовый редактор, компилятор и компоновщик — наиболее часто используемые инструменты создания программ.

1. СОЗДАНИЕ ПРОГРАММЫ

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.

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 pico (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.

```
#include <stdio.h>
#include <time.h>
int main (void)
{
   time_t nt = time (NULL);
   printf ("%s", ctime (&nt));
   return 0;
}
```

Листинг 1

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. Инструкция 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).

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), 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 Sep 4 09:37:01 2019
```

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 позволяет 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 программу. Для этого уберем в исходном файле первую инструкцию функции main(), которая объявляет переменную nt. Теперь снова попробуем откомпилировать полученный 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.)
```

Очень важно научиться понимать сообщения об ошибках, выводимых компилятором. В нашем случае сообщается, что произошло "нечто" в файле myclock.c внутри функции main(). Далее говорится, что строка номер 6 содержит ошибку (error): переменная 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 выводится только один раз.

Иногда вместо ошибки (error) выдается предупреждение (warning). В этом случае компиляция не останавливается, но до сведения программиста доводится информация 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 "кусок" 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.

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.

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.

Теперь вернемся к нашему примеру (листинг 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
```

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 появляется 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 2), в котором будет находиться объявление (прототип) функции print_up(). Эта функция переводит символы строки в верхний регистр и выводит полученный результат на экран.

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

Листинг 2

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 3), в котором будет находиться тело функции print_up().

```
#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");
}
```

Листинг 3

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 ctype.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 4), который будет содержать функцию main(), необходимую для компоновки и запуска программы.

```
#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;
}
```

Листинг 1.4

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".

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
```

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.

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.

В этой теме 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 теме 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.

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

Листинг 2.1

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
```

Осталось только вызвать скрипт, и проект будет создан:

```
$ ./make_printup
```

На первый взгляд, все прекрасно. Но 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, cmake, 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.

В рамках данной работы 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 makefайла, 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.

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 makefайлах 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 связки. Эти элементы несут основную нагрузку в make-файле. При помощи целевых связок 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 связка.

Для правильного составления make-файла необходимо определить основную цель сборки проекта. Затем следует выявить промежуточные цели, если таковые существуют. Вернемся к примеру из предыдущей темы (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 makefайлах за каждую цель отвечает своя целевая связка.

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 связки. Каждая целевая связка 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).

```
# 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
```

Листинг 2.2

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 связки. Первая связка 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 связки. Последняя (clean) требует особого рассмотрения:

1. Сначала указывается имя цели (clean).

2. После двоеточия следует пустой список зависимостей. Это значит, что данная связка 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 избавить проект 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 умещающуюся 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).

```
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)
```

Листинг 2.3

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.

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).

```
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)
```

Листинг 2.4

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).

```
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)
```

Листинг 2.5

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).

```
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 $^

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

Листинг 2.6

Makefile не только уменьшился в размере, но и стал более гибким. Теперь при изменении целей или списков зависимостей не требуется параллельно изменять инструкции. Итак, имея определенный багаж знаний, можно окончательно модернизировать Makefile для проекта printup (листинг 2.7).

```
CC=gcc
CCFLAGS=-Wall
CLEAN=rm -f
PROGRAM_NAME=printup
OBJECT_FILES=*.o
SOURCE_FILES=print_up.c main.c

$(PROGRAM_NAME): $(OBJECT_FILES)
	$(CC) $(CCFLAGS) -o $@ $^

$(OBJECT_FILES): $(SOURCE_FILES)
	$(CC) $(CCFLAGS) -c $^

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

Листинг 2.7

Мы усовершенствовали Makefile до такой степени, что для добавления в проект нового исходного файла достаточно будет дописать его имя в константу SOURCE_FILES. В этой версии make-файла появляется новая константа CCFLAGS с "магическим" значением -Wall. Посредством этой константы компилятору могут передаваться какие-то общие опции. В нашем случае опция -Wall включает все виды предупреждений (warnings). Таким образом, если компилятор "заподозрит" что-то неладное, то немедленно сообщит об этом.

2.5. Рекурсивный вызов make

Иногда программные проекты разделяют на несколько независимых подпроектов.

В этом случае каждый подпроект имеет свой make-файл. Но рано или поздно понадобится все соединить в один большой проект. Для этого используется концепция рекурсивного вызова make, предполагающая наличие главного make-файла, который инициирует автосборку каждого подпроекта, а затем объединяет полученные файлы.

Посредством опции -C можно передать автосборщику make имя каталога, в котором следует искать Makefile. Это позволяет вызвать make для каждого под-проекта из главного make-файла. Чтобы понять, как это осуществляется на практике, рассмотрим пример многокомпонентного программного проекта.

Итак, задача состоит в написании программы, которая выводит текущую версию Linux, а затем повторяет вывод заглавными буквами. Проект разбивается на два подпроекта. В одном реализуется функция чтения версии Linux, в другом — перевод всех символов полученного результата в верхний регистр. Конечным продуктом каждого подпроекта будет объектный файл. В рамках главного проекта эти файлы будут компоноваться в один бинарник.

Сначала нужно создать два каталога с именами readver и toup. В первом каталоге будет размещаться проект чтения версии Linux, во втором — проект перевода полученного результата в верхний регистр. Оба подпроекта будут содержать по одному исходному и по одному заголовочному файлу.

Теперь создайте в каталоге readver файлы readver.h (листинг 2.8) и readver.c (листинг 2.9).

```
#define STR_SIZE 1024
int readver (char * str);
```

Листинг 2.8

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

int readver (char * str)
{
   int i;
   FILE * fp = fopen ("/proc/version", "r");
   if (!fp) {
      fprintf (stderr, "Cannot open /proc/version\n");
      return 1;
   }
   for (i = 0; (i < STR_SIZE) && ((str[i] = fgetc(fp)) != EOF); i++);
   str[i] = '\0';
   fclose (fp);
   return 0;
}
```

Листинг 2.9

С заголовочным файлом все понятно: здесь определяется соглашение по использованию функции readver(), а также объявляется макроконстанта STR_SIZE, которая показывает максимальный размер буфера для считывания файла. Функция readver(), реализованная в readver.c, читает файл /proc/version, в котором находится информация о текущей версии Linux. Эта функция записывает результат (содержимое файла /proc/version) в строку, переданную в качестве аргумента. Вся ответственность за выделение памяти лежит на вызывающей стороне.

Сначала /proc/version открывается в режиме "только для чтения" (read-only). После стандартной проверки начинается посимвольное считывание файла и занесение результата в строку str, которая завершается нуль-терминатором ('\0'). При успешном завершении функция возвращает ноль, в противном случае — ненулевое значение.

Теперь в каталоге readver следует создать make-файл, который будет собирать подпроект чтения версии Linux (листинг 2.10).

```
CC=gcc
CCFLAGS=-Wall
CLEAN=rm -f
readver.o: readver.c
	$(CC) $(CCFLAGS) -c $^
clean:
	$(CLEAN) *.o
```

Листинг 2.10

Следующий шаг — создание подпроекта, реализующего механизм перевода полученных из /proc/version данных в верхний регистр. Для этого необходимо перейти в каталог toup и создать в нем заголовочный файл toup.h (листинг 2.11), исходный файл toup.c (листинг 2.12) и собственно make-файл (листинг 2.13)

```
void toup (char * str);
```

Листинг 2.11

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "toup.h"
void toup (char * str)
{
   int i;
   for (i = 0; i < strlen (str); i++)
      str[i] = toupper (str[i]);
}
```

Листинг 2.12

```
CC=gcc
CCFLAGS=-Wall
CLEAN=rm -f

toup.o: toup.c
   $(CC) $(CCFLAGS) -c $^

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

Листинг 2.13

Функция toup() не выводит строку на экран, а просто конвертирует каждый ее символ в верхний регистр, оставляя вызывающей стороне задачу выделения памяти.

Теперь, когда оба подпроекта готовы, поднимемся на уровень выше и создадим исходный файл upver.c (листинг 2.14), в котором будут вызываться функции readver() и toup().

```
#include <stdio.h>
#include <stdlib.h>
#include <readver.h>
#include <toup.h>
int main (void)
{
   char * str = (char*) malloc (STR_SIZE * sizeof(char));
   if (str == NULL) {
      fprintf (stderr, "Cannot allocate memory\n");
      return 1;
   }
   if (readver (str) != 0) {
      fprintf (stderr, "Failed\n");
      return 1;
   }
   printf ("%s\n", str);
   toup (str);
   printf ("%s\n", str);
   free (str);
   return 0;
}
```

Листинг 2.14

Обратите внимание, что заголовочные файлы readver.h и toup.h записываются не в кавычках, как мы привыкли, а в угловых скобках. Это значит, что компилятор (точнее — препроцессор) будет искать данные файлы в специально отведенных каталогах (наподобие /usr/include или /usr/local/include). Перед нами встает задача добавления в этот список каталогов readver и toup. Для этого используется опция компилятора -I, которая указывает каталог, в котором находятся заголовочные файлы.

```
CC=gcc
CCFLAGS=-Wall
MAKE=make
CLEAN=rm -f
PROGRAM_NAME=upver
OBJECTS=readver/readver.o toup/toup.o

$(PROGRAM_NAME): make-readver make-toup upver.o
	$(CC) $(CCFLAGS) -o $(PROGRAM_NAME) $(OBJECTS) upver.o

upver.o: upver.c
	$(CC) $(CCFLAGS) -c -Ireadver -Itoup $^

make-readver:
	$(MAKE) -C readver readver.o

make-toup:
	$(MAKE) -C toup toup.o
clean:
	$(CLEAN) *.o $(PROGRAM_NAME)
	$(MAKE) -C readver clean
	$(MAKE) -C toup clean
```

Листинг 2.15

Цели make-readver и make-toup не являются файлами. Это так называемые псевдоцели. Соответствующие им целевые модули вызывают утилиту make с опцией -C.

Проще говоря, целевой модуль make-readver собирает подпроект чтения версии Linux, а модуль make-toup — второй подпроект, переводящий символы результата в верхний регистр. Аналогичным образом происходит очистка (цель clean): сначала очищаются файлы главного проекта, а затем выполняется очистка подпроектов их собственными средствами.

Как видно из рассмотренного примера, любой программный проект можно разделить на подпроекты, используя концепцию рекурсивного вызова make. Подобным образом, например, разрабатывается ядро Linux: в настоящее время в его исходниках насчитывается более 1400 make-файлов.

2.6. Получение дополнительной информации

В этой теме принципы автоматической сборки программ в Linux описаны лишь частично. Дополнительную информацию по этой теме можно получить из следующих источников:

- man 1 make — man-страница утилиты GNU make;

- info make — info-страница GNU make;
- http://www.gnu.org/software/autoconf/manual/html_node/ — Autoconf;
- http://www.gnu.org/software/automake/manual/html_node/ — Automake;
- http://sources.redhat.com/autobook/autobook/autobook_toc.html — Autotools;
- http://tmake.sourceforge.net/ — tmake;
- http://www.cmake.org/HTML/Documentation.html — cmake;
- http://www.snake.net/software/imake-stuff/ — imake;
- http://doc.qt.nokia.com/ — qmake;
- http://www.fastmake.org/doc.html — Fastmake.