Е. П. Моргунов
О. Н. Моргунова
В. В. Тынченко

Технологии разработки программ на основе инструментария с открытым исходным кодом

Вводный курс

Красноярск 2006

Федеральное агентство по науке и инновациям
Научно-исследовательский институт систем управления,
волновых процессов и технологий

Е. П. Моргунов
О. Н. Моргунова
В. В. Тынченко

Технологии разработки программ на основе инструментария с открытым исходным кодом

Вводный курс

Красноярск 2006

УДК 004.4
ББК 32.973.26
М 79

Рецензент:
доктор технических наук, профессор А. Н. Антамошкин
(Сибирский государственный аэрокосмический университет)

Моргунов, Е. П.

М 79 Технологии разработки программ на основе инструментария с открытым исходным кодом. Вводный курс : учеб. пособие / Е. П. Моргунов, О. Н. Моргунова, В. В. Тынченко ; НИИ СУВПТ. – Красноярск, 2006. – 148 с.

ISBN 5–98027–046–9

В учебном пособии описаны основные технологии разработки программ на основе инструментария с открытым исходным кодом. Освещаются такие вопросы, как установка такого инструментария, его настройка и практическое применение.

Пособие предназначено для студентов, обучающихся по специальностям 230102 – «Автоматизированные системы обработки информации и управления», 090106 – «Информационная безопасность телекоммуникационных систем» и по направлению 230100 – «Информатика и вычислительная техника». Оно может быть полезно широкому кругу пользователей, знакомых с основами программирования в операционной системе Windows, желающих самостоятельно познакомиться с технологиями разработки программ в среде операционной системы UNIX.

Утверждено к печати редакционно-издательским советом
НИИ СУВПТ

УДК 004.4
ББК 32.973.26

ISBN 5–98027–046–9 Печатается в авторской редакции

Оглавление

Введение

В настоящее время все большую популярность в мире приобретает свободное программное обеспечение (free software). Операционная система Linux является ярким примером успешной разработки крупномасштабного программного продукта с открытым исходным кодом. Существуют многочисленные средства разработки программ, такие, как компиляторы, отладчики, редакторы, библиотеки подпрограмм, поставляемые на условиях свободного программного обеспечения (ПО). Первоначально большинство этих средств разработки предназначались для использования в среде операционной системы (ОС) UNIX, но теперь многие из них перенесены и в среду ОС Windows.

В учебном пособии предпринята попытка показать наиболее важные инструменты и технологии, применяемые для разработки программ различных типов (базы данных, Интернет-приложения и др.). Принципиально важным в условиях ужесточения законодательства в сфере авторских прав является то, что все представленные здесь программные средства – свободно распространяемые. Конечно, в рамках небольшого пособия невозможно даже кратко коснуться всех существующих, зачастую конкурирующих, средств разработки программ. Как правило, у каждого программиста есть свои личные предпочтения. Поэтому при выборе программных продуктов нами не ставилась задача убедить читателя в преимуществах, например, системы управления базами данных (СУБД) PostgreSQL над СУБД MySQL. Хотя для иллюстрации взаимодействия прикладной программы с базой данных нами была выбрана СУБД PostgreSQL, но, ознакомившись с принципами такого взаимодействия, вы сможете самостоятельно организовать его на основе СУБД MySQL.

В пособии представлен вспомогательный инструментарий, применяемый на различных стадиях разработки программ: это текстовый редактор emacs, отладчик gdb, утилита make. Поскольку в современном мире программного обеспечения библиотеки подпрограмм играют очень важную роль, то показана также технология создания и использования библиотек, как статических, так и динамических (разделяемых). Уделено внимание разработке Интернет-приложений. И хотя в качестве языка программирования в данном случае нами выбран Perl, но, изучив принципы и основные приемы, вы сможете самостоятельно освоить технологию разработки приложений на языке PHP.

Использование инструментов и технологий иллюстрируется сначала в среде операционной системы FreeBSD 6, а затем – в среде ОС Windows. Для более четкого выделения специфики Windows в тексте используется специальная пометка:

В операционной системе Windows

В тексте пособия команды, вводимые пользователем, выделяются полужирным шрифтом. Например:

```
psql –d test –U root
```

Если вся команда не умещается на одной строке текста, то она переносится на следующие строки, но вы должны вводить ее на одной строке, например:

```
../configure --with-msw --enable-shared --enable-log
--enable-debug --enable-debug_gdb --enable-debug_cntxt
--enable-debug_info --enable-debug_flag --with-odbc
--enable-monolithic --disable-optimise --enable-intl
--enable-unicode --enable-dnd
```

Результаты работы команд операционной системы или программ напечатаны моноширинным шрифтом. Например, в ответ на команду

```
ldd ./demo_shared_lib
```

на экран будет выведено следующее:

```
demo_shared_lib:
     libgeometry.so.1 => not found (0x0)
     libc.so.6 => /lib/libc.so.6 (0x2807a000)
```

Тексты программ, которые приведены в учебном пособии, напечатаны также моноширинным шрифтом. Например:

```
# если нажата кнопка "Показать весь список"
if ( $query->param( 'show_db' ) )
{
  # вывод всего списка студентов
  show_db( $conn, $query );
}
```

В пособии часто будут даваться указания ввести ту или иную команду. Это означает, что вам следует не только набрать на клавиатуре текст этой команды, но также нажать клавишу Enter для ее выполнения. Для сокращения объема текста мы не будем повторять слова «Нажмите клавишу Enter» каждый раз.

Мы надеемся, что изучение материала, изложенного в нашем учебном пособии, будет способствовать расширению вашего профессионального кругозора и повышению уровня квалификации.

1. Установка среды MSYS и системы программирования MinGW в операционной системе Windows

В этой главе будет показано, как можно установить набор свободно распространяемых средств разработки программ для Windows, а также каким образом можно создать в среде операционной системы Windows среду разработки, аналогичную той, которая имеет место в операционной системе UNIX.

Поскольку средства разработки программ, распространяемые в исходных текстах, были «рождены» в мире UNIX, то в нашем пособии мы будем делать основной упор именно на эту операционную среду. Однако в настоящее время идеи свободно распространяемого программного обеспечения все глубже проникают и в Windows среду. В результате Windows-программистам стали доступны компиляторы языков C/C++, первоначально разработанные для использования в среде UNIX. Разработка программ в среде UNIX имеет ряд особенностей, в том числе и чисто технологических. Поэтому для тех программистов, которые, работая в среде Windows, хотели бы чувствовать себя в родной для них среде UNIX, была разработана специальная система **MSYS** (Minimal SYStem), которая и создает UNIX-подобную среду в рамках Windows. В результате стало возможно не только перенести исходный текст программы из среды ОС UNIX в среду ОС Windows, но и использовать для ее компиляции те же инструментальные средства, которые традиционно применяются в среде ОС UNIX.

Перейдем непосредственно к процедурам установки программного обеспечения.

1. Загрузите необходимые программные пакеты для установки среды MinGW.

Откройте домашнюю страницу проекта MinGW (Minimalist GNU for Windows), адрес которой в сети Интернет http://www.mingw.org.

ПРИМЕЧАНИЕ. Мы не будем давать детальных указаний относительно местонахождения каждого из программных продуктов, упоминающихся в этой главе, поскольку отыскать их можно, следуя по ссылкам с этой страницы.

Среда MinGW состоит из набора пакетов (продуктов). Имена файлов, соответствующих каждому пакету, содержат имя пакета, номер версии и дату создания. Как правило, они являются архивными файлами и имеют расширение tar.gz или tar.bz2. Если в имени файла присутствует суффикс src, то это означает, что архив содержит исходные тексты данного пакета (в настоящий момент вы можете без них обойтись).

Рекомендуем вам для начала ограничиться самым необходимым инструментарием и загрузить на свой компьютер следующий набор пакетов (номера версий в именах файлов мы заменяем символами «x»):

– MinGW Runtime (**mingw-runtime-x.tar.gz**);

– GCC Version 3 – компиляторы языков C и C++ (**gcc-core-x.tar.gz** и **gcc-g++-x.tar.gz**);

– GNU Source-Level Debugger – отладчик (**gdb-x.tar.bz2**);

– MinGW API for MS-Windows (**w32api-x.tar.gz**);

– GNU Binutils – различные утилиты (**binutils-x.tar.gz**);

– GNU Make (**mingw32-make-x.tar.gz**).

Найти текущие версии этих пакетов можно в категории «Current» на сайте проекта SourceForge, куда ведет ссылка со страницы http://www.mingw.org/download.shtml.

2. Установите пакеты.

Устанавливать MinGW нужно в каталог, имя которого не содержит пробелов. Процедура установки очень проста. Создайте каталог **C:\MinGW** и разархивируйте полученные архивные файлы непосредственно в нем.

ПРЕДОСТЕРЕЖЕНИЕ. Не устанавливайте этот программный продукт в каталог C:\Program Files.

Если в процессе разархивирования встречаются одноименные файлы, то рекомендуем присваивать им различные имена, например, с использованием суффикса, показывающего порядковый номер дубликата (2, 3 и т. д.). Такие дубликаты могут встретиться только для файлов с именем **info\dir**.

У вас должна получиться примерно такая структура подкаталогов в каталоге **C:\MinGW**:

```
bin
doc
include
info
lib
libexec
man
mingw32
```

3. Загрузите необходимые программные пакеты для установки среды MSYS.

Опять откройте домашнюю страницу проекта MinGW, найдите по цепочке ссылок и загрузите на свой компьютер пакеты:

– **MSYS-1.0.10.exe**;

– **msysDTK-1.0.1.exe** (это дополнительные средства для разработки программ: Developer's Toolkit).

4. Установите среду MSYS.

Запустите установочную программу **MSYS-1.0.10.exe**. Устанавливать среду MSYS нужно в каталог, имя которого не содержит пробелов. Рекомендуем установить ее в каталог **C:\msys**.

ПРЕДОСТЕРЕЖЕНИЕ. Не устанавливайте этот программный продукт в каталог C:\Program Files.

Запустите программу инсталляции дополнительных инструментов разработчика **msysDTK-1.0.1.exe**. Установить эти программы следует также в каталог **C:\msys**.

5. Выполните необходимые настройки в среде MSYS.

Для работы с инструментарием MinGW необходимо добавить так называемую «привязку» в файл **C:\msys\1.0\etc\fstab**. Сразу после инсталляции продукта этого файла не существует. Создайте его посредством копирования файла **fstab.sample**:

```
cd c:\msys\1.0\etc
copy fstab.sample fstab
```

Затем введите в файл **fstab** такую строку (первая колонка отделяется от второй символами табуляции, а не пробелами):

```
c:/MinGW        /mingw
```

В этой записи в первой колонке указан путь к каталогу в стиле Windows, а во второй колонке указана точка монтирования в стиле UNIX. Обратите внимание на то, что в качестве символа-разделителя имен каталогов в Windows-пути используется символ «/», как в системе UNIX.

Вы можете сделать и другие привязки, если в этом есть необходимость, например:

```
c:/perl             /perl
```

6. Загрузите среду MSYS.

Для запуска используйте группу MinGW, которая создается в меню «Программы» при установке этого продукта, а также ярлык на «Рабочем столе».

В дереве подкаталогов среды MSYS, **C:\msys**, будет создан подкаталог **home**, в котором будет и подкаталог с именем того пользователя, который произвел установку этой среды. При работе в среде MSYS рекомендуем использовать этот домашний каталог таким же образом, как это традиционно делается в ОС UNIX: создавайте ваши программы в этом каталоге.

Попробуйте создать небольшую тестовую программу на языке C или C++ и скомпилировать ее в этой среде. Команда компиляции для программы на языке C будет примерно такой:

```
gcc –o my_prg.exe my_prg.c
```

В этой команде: **gcc** – имя компилятора, **-o my_prg.exe** – имя результирующего исполняемого файла, **my_prg.c** – имя исходного файла программы.

Для запуска полученной программы лучше следовать традиционным соглашениям операционной системы UNIX и предварять имя исполняемого файла, находящегося в текущем каталоге, символами «./», означающими текущий каталог.

В среде MSYS работают практически все основные UNIX-команды. Для получения подсказки по работе с программами можно использовать так называемый GNU-стиль, т. е. предварять имя параметра двумя символами дефис «-», например:

```
g++ --help
```

Для выхода из сеанса работы с MSYS можно нажать клавиши Ctrl-D или ввести команду

```
exit
```

Можно использовать среду MinGW без ее интеграции с MSYS. В этом случае рекомендуем добавить путь к каталогу **C:\MinGW\bin**, в котором содержатся исполняемые файлы этой системы, в команду PATH операционной среды текущего пользователя операционной системы Windows.

Среда MSYS предоставляет вам возможность запуска в двух вариантах: используя программу эмуляции терминала **rxvt.exe** либо **sh.exe**. Это можно выяснить, если внимательно ознакомиться с текстом командного файла **C:\msys\1.0\msys.bat**, который используется для запуска среды MSYS. В этом файле есть такие строки:

```
if EXIST rxvt.exe goto startrxvt
if EXIST sh.exe goto startsh
```

По умолчанию запускается терминал **rxvt.exe**, но если вы хотите, чтобы запускался **sh.exe**, то можете последовать рекомендации автора продукта MSYS. Автор предлагает просто переименовать файл **C:\msys\1.0\bin\rxvt.exe** (не удаляя его совсем, конечно). В результате переход к метке startrxvt осуществляться не будет, а будет запускаться **sh.exe**.

Мы же рекомендуем создать еще одну версию командного файла для запуска MSYS, назвать его можно, например, **C:\msys\1.0\msys_sh.bat**. В этой версии нужно закомментировать строку, передающую управление на метку startrxvt. Эта строка будет выглядеть так:

```
rem if EXIST rxvt.exe goto startrxvt
```

Затем нужно создать еще одну опцию (пункт) в меню Windows для запуска программ: Пуск –> Все программы –> MinGW –> MSYS, т. е. создать ярлык для запуска нового командного файла. После этого вы сможете выбирать тип терминала для запуска среды MSYS более удобным способом, т. е. без переименований файлов.

ПРИМЕЧАНИЕ. Изменение терминала для среды MSYS может потребоваться, если вы будете испытывать трудности при запуске консольных приложений. Например, при использовании в программе стандартной функции printf() вывод на экран производится не сразу, а с задержкой, возможно даже до завершения программы.

Напоследок сообщим вам, что в среде MSYS работает очень удобная функция командных интерпретаторов UNIX, которая позволяет при работе в командной строке дополнять вводимые имена файлов и каталогов с помощью клавиши Tab. Это выглядит следующим образом: например, вы с помощью команды cd хотите перейти в каталог **/usr/local/lib**. Вы вводите команду и первые символы пути

```
cd /usr/lo
```

и нажимаете клавишу Tab. Командный интерпретатор дополнит имя до **/usr/local/**. Затем вы вводите символ l (латинская буква «el»):

```
cd /usr/local/l
```

и опять нажимаете клавишу Tab. Командный интерпретатор дополнит имя до **/usr/local/lib**. Теперь остается лишь нажать клавишу Enter.

Общее правило таково: достаточно вводить лишь минимальное количество первых символов имени файла или каталога, чтобы по ним можно было однозначно определить нужное вам имя. Если введенных вами символов недостаточно для однозначной идентификации ваших желаний, то будет произведено дополнение лишь до той части имени файла или каталога, которую можно однозначно определить, а затем вы должны снова «помочь» интерпретатору команд – ввести следующие символы и опять нажать клавишу Tab. Например, если в текущем каталоге есть подкаталоги с именами **my_programs** и **my_texts**, то если вы введете

```
cd m
```

и нажмете клавишу Tab, то получите следующую картину:

```
cd my_
```

Для того чтобы однозначно указать, в какой из подкаталогов следует перейти, вы должны ввести еще один символ – «p» или «t», а затем нажать клавишу Tab.

2. Текстовые редакторы

Хотя мы живем в эпоху интегрированных сред разработки (Integrated Development Environment), тем не менее, автономные текстовые редакторы не исчезли как класс инструментов и продолжают оставаться основным инструментом для написания исходных текстов программ.

Мы рассмотрим три редактора, в том числе и редактор **vi.** Можно сказать, что **vi** – легендарный редактор.

2.1. Текстовый редактор vi

Этот текстовый редактор является одним из базовых средств ОС UNIX и устанавливается по умолчанию при установке операционной системы. Он используется в основном системными администраторами, но и прикладному программисту элементарные навыки владения редактором **vi** не повредят. Интересно, что текст книги Уильяма Стивенса «UNIX: взаимодействие процессов», объем которой составляет более 500 страниц, был создан им в редакторе vi. Конечно, рисунки и макет книги были подготовлены с использованием других средств, но непосредственно текстовая часть книги была создана именно в этом редакторе.

Несмотря на свою внешнюю невзрачность, редактор **vi** имеет множество возможностей. Мы познакомимся только с самыми основными из них, а именно: ввод текста, удаление текста, запись изменений в файл, выход из редактора. Тем же, кто хочет узнать больше об этом замечательном редакторе, можно дать все тот же совет: используйте электронную справочную систему – **man vi**.

Для запуска редактора и создания нового файла введите:

```
vi my_file.txt
```

Вы увидите черный экран, внизу которого есть строка состояния. Если попытаться просто вводить текст, то у вас ничего не получится. Данный редактор имеет два режима работы: командный и, условно говоря, «рабочий», т. е. режим редактирования. При запуске редактор переходит в командный режим. Чтобы начать вводить текст, нажмите клавишу i. Попробуйте вводить текст, как в обычном редакторе. Попробуйте удалить символ. Внимательно наблюдайте реакцию редактора. Если вы переместите курсор на строку вверх, то редактор сам перейдет в командный режим. Если же вы хотите перевести его в командный режим принудительно, то нажмите клавишу Escape. Иногда не повредит и двукратное нажатие клавиши Escape, если вы не уверены, что редактор уже переведен в командный режим.

Приведем краткую сводку команд редактора:

a ввести текст ПОСЛЕ курсора;

i ввести текст ПЕРЕД курсором;
o ввести пустую строку ПОД строкой, на которой стоит курсор;
O ввести пустую строку НАД строкой, на которой стоит курсор;
dd удалить строку, на которой стоит курсор;
x удалить символ, на котором стоит курсор;
yy скопировать в буфер строку, на которой стоит курсор;
p вставить строку из буфера ПОСЛЕ строки, на которой стоит курсор.

Например, чтобы удалить строку текста, на которой стоит курсор, сначала нужно перейти в командный режим, а уже затем дважды нажимать клавишу d. Однако если вам требуется удалить более одной строки текста, то достаточно однократного перехода в командный режим, после чего вы можете удалять строки, перемещаясь по ним с помощью клавишей управления курсором или команд j и k (см. ниже).

Для перемещения по файлу можно использовать такие команды (в дополнение к клавишам управления курсором):

h переместить курсор ВЛЕВО на один символ;
l (буква «el») переместить курсор ВПРАВО на один символ;
j переместить курсор ВНИЗ на одну строку;
k переместить курсор ВВЕРХ на одну строку.

Для выполнения поиска в файле перейдите в командный режим, нажав клавишу Escape, а затем нажмите клавишу «/» и введите искомый текст, после чего нажмите клавишу Enter. Если такой текст есть в файле, то курсор будет установлен на первый символ этого фрагмента текста.

Команды для записи изменений в файл (после ввода команд нужно нажимать клавишу Enter):

:w записать файл с тем же именем, какое он имел при запуске редактора;
:w new_file_name записать файл с именем new_file_name.

Команды для выхода из редактора (после ввода команд нужно нажимать клавишу Enter):

:q прекратить редактирование файла и выйти из редактора. Если файл был изменен, а изменения в файле не были сохранены, то редактор не позволит выйти;
:q! прекратить редактирование файла и выйти из редактора. Если файл был изменен, а изменения в файле не были сохранены, то они будут потеряны. Говоря коротко, эта команда позволяет выйти из редактора без сохранения изменений.

ПРИМЕЧАНИЕ. Обратите внимание на символ ":".

Ну, вот теперь вы сможете блеснуть мастерством и на глазах у какого-нибудь изумленного приверженца графических оболочек запросто отредактировать файл в редакторе **vi**.

В среде MSYS также можно запустить редактор **vi**. Однако в этой среде используется его обновленный вариант, который называется **vim** (Vi Improved). Для запуска можно использовать любую из двух команд: vi или vim.

2.2. Редактор joe

2.2.1. Установка и настройка текстового редактора joe

Выполним установку текстового редактора **joe** из исходных текстов. Это многооконный редактор, имеющий богатые возможности для редактирования исходных текстов программ на различных языках программирования. Исходные тексты этого программного продукта можно получить в сети Интернет по адресу http://sourceforge.net/projects/joe-editor.

ПРИМЕЧАНИЕ. Вы можете выполнять все команды, приведенные ниже, в среде файлового менеджера **Demos Commander**. При этом можно пользоваться его способностью сохранять историю введенных команд. Эти команды можно просматривать с помощью комбинаций клавишей Ctrl-E (просмотр назад) и Ctrl-X (просмотр вперед). Выбрав нужную команду, ее можно при необходимости отредактировать (не забывайте, что для перемещения курсора по командной строке нужно сначала нажать клавиши Ctrl-P), а затем выполнить, нажав клавишу Enter.

Как и при установке файлового менеджера **deco**, смонтируйте носитель (CD диск или устройство флэш-памяти), на котором находится архивный файл **joe-3.5.tar.gz** (или **joe-3.5.tgz**). Затем с помощью команд **cd** и **ls** отыщите архивный файл и скопируйте этот архив в каталог **/root**. Затем выйдите из каталога **/cdrom** (или **/mnt**) и размонтируйте устройство:

```
cp joe-3.5.tar.gz /root
cd /root
umount /cdrom
```

Теперь разархивируйте архив с исходными текстами редактора **joe**:

```
tar xzvf joe-3.5.tar.gz
```

В результате будет создан подкаталог **joe-3.5**, в котором разместятся файлы исходных текстов. Перейдите в этот подкаталог и выполните ряд команд (обратите внимание на символы «./» в одной из команд):

```
cd joe-3.5
./configure
make
```

```
make install
```

Если на экран не было выведено сообщений об ошибках, то программа **joe** успешно установлена. Чтобы запустить ее, введите команду

```
joe
```

Если редактор запустится, то введите в его окне какой-нибудь текст (пока что это можно сделать только с использованием латинского алфавита). Попробуйте выполнить операцию удаления введенных символов с помощью клавишей Backspace и Delete. Как вы можете убедиться, клавиша Delete работает так же, как и клавиша Backspace. Конечно, это не очень удобно, но вскоре мы покажем, как можно добиться от клавиши Delete традиционного функционирования.

Данный редактор имеет следующую особенность: многих из его функций активизируются при наборе комбинации из *трех* клавишей. Например, для получения подсказки необходимо использовать комбинацию Ctrl-K-H. Комбинации из трех клавишей нужно набирать таким образом: нажав клавишу Control и удерживая ее, нажать *поочередно* две символьные клавиши, в данном случае это клавиши K и H. Чтобы убрать с экрана текст подсказки, наберите комбинацию Ctrl-K-H еще раз.

Для сохранения введенного текста в файле служит комбинация клавишей Ctrl-K-D, для выхода из редактора (точнее, для закрытия файла) с сохранением внесенных изменений текста – Ctrl-K-X, а для выхода без сохранения изменений – Ctrl-C. Чтобы выделить блок текста, нужно установить курсор в начало нужного фрагмента и нажать клавиши Ctrl-K-B, затем перевести курсор в конец фрагмента текста и нажать клавиши Ctrl-K-K. Для копирования выделенного блока установите курсор в нужное место и нажмите клавиши Ctrl-K-C, а для перемещения блока – Ctrl-K-M. Мы рекомендуем изучить основные приемы работы с редактором с помощью его подсказки. В тексте этой подсказки в качестве обозначения клавиши Control используется символ ^. Рекомендуем также воспользоваться и электронным руководством, вызвав его с помощью команды

```
man joe
```

Теперь сделаем ряд настроек в конфигурационном файле редактора. Этот файл находится в каталоге **/usr/local/etc/joe** и называется **joerc**.

Прежде чем вносить изменения в файл **joerc**, сделайте его копию (в имя копии файла можно для наглядности добавить расширение orig, т. е. original):

```
cd /usr/local/etc/joe
cp joerc joerc.orig
```

Для внесения изменений в конфигурационный файл **joerc** вы можете воспользоваться либо встроенным редактором файлового менеджера **deco**, либо самим редактором **joe**, используя который, не забывайте о том, что клавиша Delete работает так же, как и клавиша Backspace.

Обратите внимание на то, что файл **joerc** содержит не только конфигурационные параметры, но и инструкции по их использованию.

Внесем следующие изменения в конфигурацию редактора.

1. Найдите строку (строка номер 60), в начале которой располагается параметр **-asis**. Он отвечает за правильное отображение букв русского алфавита (вопросы русификации операционной системы будут рассмотрены немного позднее). Обратите внимание, то перед этим параметром в начале строки стоит один пробел. Нужно удалить этот пробел, тем самым параметр будет активизирован.

2. Поскольку использовать комбинации из трех клавишей не очень удобно, назначим для нескольких часто используемых операций функциональные клавиши:

– клавишу F1 для вызова подсказки (помощи). Для этого выполните следующие действия: перейдите в файле **joerc** в район строки номер 554; в этом месте находятся три строки, начинающиеся с ключевого слова help, которое является кодовым наименованием операции вызова экранной подсказки. Добавьте после этих строк еще одну строку с ключевым словом help, а в качестве кода клавишей введите «.k1» (кавычки вводить не нужно). Обратите внимание, что ключевое слово отделяется от кода клавишей двумя символами табуляции (табуляция используется и во всех остальных случаях, которые мы сейчас опишем);

– клавишу F5 для перехода к началу файла. Выполните аналогичные действия: перейдите в файле **joerc** в район строки номер 795, в этом месте находятся три строки, начинающиеся с ключевого слова bof, которое является кодовым наименованием операции перехода к началу файла. Добавьте после этих строк еще одну строку с ключевым словом bof, а в качестве кода клавишей введите «.k5» (кавычки вводить не нужно);

– клавишу F6 для перехода к концу файла. Поскольку схема действий вам уже ясна, будем давать лишь краткие указания: строка номер 819, ключевое слово eof (операция перехода к началу файла). Добавьте после последней из строк с этим ключевым словом еще одну строку с ключевым словом eof, а в качестве кода клавишей введите «.k6» (без кавычек);

– клавишу F7 для поиска текстовой строки в файле. В районе строки номер 830 есть строки с ключевым словом ffirst. Добавьте еще одну строку, а в качестве кода клавишей введите «.k7»;

– клавишу F2 для сохранения файла. В районе строки номер 882 есть строки с ключевым словом save. Добавьте еще одну строку, а в качестве кода клавишей введите «.k2».

3. Исправьте «поведение» клавиши Delete по уже знакомой вам схеме. В районе строки номер 805 есть строки с ключевым словом delch. До-

бавьте еще одну строку, а в качестве кода клавишей введите «^?» (без кавычек).

ПРИМЕЧАНИЕ. Номера строк в файле **joerc** указаны приблизительно, т. к. они могут изменяться после вставки новых строк в процессе проведения настроек.

Мы сделали лишь самые необходимые настройки.

2.2.2. Работа с текстовым редактором joe

Сделаем лишь краткий обзор основных возможностей редактора. Одной из сильных его сторон является возможность работы с несколькими файлами одновременно. Чтобы открыть в редакторе несколько файлов, передайте их имена в командной строке:

```
joe file1 file2 file3
```

Для перехода между различными окнами редактора, содержащими открытые файлы, служат клавиши Ctrl-K-N (переход к следующему окну) и Ctrl-K-P (переход к предыдущему окну).

Открыть файл можно и непосредственно из редактора с помощью клавишей Ctrl-K-E. Получив приглашение для ввода имени файла, можно имя не вводить, а вместо этого нажатием клавиши Tab вывести на экран список файлов текущего каталога и выбрать файл из списка. Можно также перемещаться по структуре каталогов в поисках нужного файла.

Иногда встречается такая ошибка: невозможно сохранить файл. Это может объясняться тем, что начинающие пользователи операционной системы UNIX (в частности, FreeBSD), перемещаясь по каталогам системы, запускают редактор, находясь не в своем домашнем каталоге, а в том каталоге, в котором этот пользователь не имеет права на запись файлов. Если введен уже значительный объем текста, потерять который нежелательно, то при сохранении файла, когда будет предложено ввести его имя, следует ввести не только имя, но и полный путь к нему, причем, этот путь должен вести в ваш домашний каталог. Например: **/home/stud/my_file.txt**.

В операционной системе Windows

Поскольку доступны исходные тексты редактора, то можно попытаться скомпилировать редактор **joe** в среде MSYS. Автор редактора рекомендует использовать параметры --disable-curses и --disable-termcap при запуске утилиты **configure**. Процедура компилирования в среде MSYS такая же, как и в ОС UNIX (обратите внимание на два символа дефис):

```
./configure --disable-curses --disable-termcap
make
```

```
make install
```

К сожалению, при выполнении команды **make** компилятор выдает сообщения об ошибках. Рекомендуем вам самостоятельно разобраться в причинах этих ошибок и попытаться их устранить.

2.3. Редактор emacs

Это очень мощный редактор, который известен уже более десяти лет. Его автор Ричард Столлмен (Richard Stallman) является одним из главных проводников и защитников идеи свободно распространяемого программного обеспечения.

Получить исходные тексты редактора можно по адресу http://www.gnu.org. Поскольку на этом сайте организован поиск представленных на нем программных продуктов, то в окне для поиска введите «emacs» и нажмите клавишу Enter. Выбрав последнюю из версий редактора, сохраните архивный файл (в формате tar.gz) на своем компьютере.

Теперь мы можем приступить к установке редактора **emacs** в среде операционной системы UNIX (например, FreeBSD). Но прежде необходимо убедиться, что на жестком диске достаточно свободного места – требуется около 200 Мб. Установка выполняется в соответствии с традициями UNIX-системы. Подробно процедура установки описана в файле INSTALL. Мы ограничимся ее краткой версией.

1. Разархивируйте файл с исходными текстами редактора с помощью программы **tar** (знаком «x» обозначена версия программного продукта):

```
tar xzvf emacs-x.tar.gz
```

2. Перейдите в каталог, в котором находятся разархивированные файлы исходных текстов редактора, и выполните команды

```
./configure > conf.log 2>&1 &
tail -f conf.log
```

Программа **configure** создает **Makefile** для компиляции исходных текстов. Переадресация стандартного вывода и вывода сообщений об ошибках в первой команде позволяет сохранить результаты работы команды в файле-журнале. Вторая из этих команд позволит вам видеть на экране ход процесса, идущего в фоновом режиме.

Конечно, можно ограничиться и простым вариантом:

```
./configure
```

3. Теперь скомпилируем исходные тексты программ, также направляя потоки вывода и сообщений об ошибках в файл-журнал:

```
make > make.log 2>&1 &
tail -f make.log
```

Конечно, и в этом случае можно ограничиться и простым вариантом:

```
make
```

4. Если компиляция прошла успешно, сообщений об ошибках не последовало, то проверьте возможность запуска редактора с помощью такой команды:

```
src/emacs –q
```

ПРИМЕЧАНИЕ. Запуск редактора длится довольно долго – около 10–15 секунд.

Поскольку на данном этапе главное – убедиться в возможности запуска редактора, то вы можете сразу покинуть его, нажав комбинацию клавишей Ctrl-X и затем сразу же Ctrl-C.

ПРИМЕЧАНИЕ. Для редактора emacs характерно использование подобных двухэтапных комбинаций клавишей.

---

5. Если редактор успешно запускается, то можно установить его в те системные каталоги, которые традиционно для этого используются (например, исполняемый файл emacs будет установлен в каталог **/usr/local/bin**). Выполните команды:

```
make install > make_install.log 2>&1 &
tail -f make_install.log
```

6. В случае успеха и этой операции можно запустить графическую среду с помощью команды **startx** и затем запустить редактор в этой среде. Например, в среде KDE можно открыть системную консоль и затем выполнить команду

```
emacs
```

ПРИМЕЧАНИЕ. Редактор может работать не только в графической среде, но и как консольная программа.

---

7. Для изучения богатых возможностей редактора можно использовать его руководство, к которому можно обратиться из меню Help.

Редактор «умеет» очень многое, например, из него можно вызывать отладчик **gdb**.

На сайте http://www.gnu.org можно найти уже скомпилированный вариант редактора **emacs**. Установка редактора в среде Windows гораздо проще, чем в среде UNIX. Она состоит из двух этапов.

1. Создайте каталог и разархивируйте в него архивный файл, полученный с сайта http://www.gnu.org. Вы сразу получите готовое дерево подкаталогов редактора.

ВАЖНОЕ ПРИМЕЧАНИЕ. Рекомендуется не использовать пробелы в имени каталога, в который вы устанавливаете редактор. Можно предложить такое имя: **C:\emacs**.

2. Запустите программу **addpm.exe** из подкаталога **bin** редактора, которая создаст подменю Gnu Emacs в списке программ, вызываемом нажатием кнопки Пуск. Вы сможете запускать редактор с помощью меню.

Однако редактор можно запустить и с помощью программы **runemacs.exe**, которая находится в подкаталоге **emacs\bin**. Эту программу можно запускать, например, из файлового менеджера FAR.

Контрольные вопросы и задания

1. Какие режимы работы имеет редактор **vi**? Как перейти в командный режим? Какая команда редактора vi служит для ввода текста в режиме добавления? Как удалить строку текста в редакторе **vi**?

2. Как в редакторе **vi** сохранить отредактированный файл? Как выйти из редактора без сохранения внесенных изменений?

3. Создайте текстовый файл в редакторе **vi** и потренируйте основные навыки работы в нем: ввод текста, удаление текста, добавление пустых строк и удаление строк. Этих элементарных знаний хватит для редактирования, например, файла паролей в команде **vipw**, о которой речь пойдет позднее.

4. Попробуйте ввести текст на русском языке в редакторе **emacs** и сохранить этот текст. Если в операционной системе UNIX это не получается, то проверьте правильность локализации («русификации») вашей системы.

5. Вкратце ознакомьтесь со всеми пунктами меню редактора **emacs**. Найдите вводное руководство (tutorial) по работе с редактором **emacs** и ознакомьтесь с ним.

3. Языки программирования

Наверное, многие согласятся с тем, что базовым языком программирования является C/C++. Но кроме него есть и другие языки, которые могут быть полезны в конкретных ситуациях. Один из них – язык Perl. В данной главе вы сможете познакомиться с основами программирования на этом языке, т. е.:

– научиться создавать программы на языке Perl и запускать их;
– изучить типы данных этого языка;
– изучить управляющие конструкции, применяемые в этом языке;
– изучить основные приемы работы с файлами;
– познакомиться с регулярными выражениями языка Perl.

3.1. Язык C/C++

В операционной системе UNIX (в частности, FreeBSD), компиляторы языка C/C++ устанавливаются, как правило, по умолчанию. Для запуска компилятора C используется команда **cc** или **gcc**, а для запуска компилятора C++ служит команда **g++**. Поскольку наше пособие предназначено для тех студентов, которые уже имеют некоторый опыт программирования на языке C/C++, то мы не будем вдаваться в дальнейшие детали, касающиеся этого языка. Скажем только, что доступ к описаниям функций этого языка можно получить с помощью команды **man**, например:

```
man printf
```

Имя библиотеки, которую требуется подключить при компиляции программы, использующей конкретную библиотечную функцию, можно найти в верхней части man-страницы, посвященной этой функции. Там же указано и имя включаемого файла, например:

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

В операционной системе Windows

В этой операционной системе компилятор языка C/C++ не устанавливается по умолчанию. Но среда MSYS включает в себя этот компилятор.

3.2. Язык Perl

3.2.1. Вводные сведения

Perl – это язык программирования, который первоначально предназначался для обработки текстовых файлов: поиска и замены строк симво-

лов, сортировки, подсчета количеств каких-либо строк и т. п. Этот язык относится к классу так называемых языков сценариев (по-английски – **scripting languages**), поэтому часто программы, написанные на языке Perl, называют **скриптами**.

В настоящее время Perl является одним из языков, применяемых для программирования Internet-приложений. Он входит в состав операционной системы UNIX (в частности, FreeBSD) «по умолчанию». Ряд системных программ в этой ОС написан на языке Perl. Этот язык отличается тем, что позволяет создавать полезные программы, используя даже небольшое подмножество функций языка. Существует реализация Perl для Windows. По своему синтаксису Perl очень похож на язык C, поэтому тем, кто знаком с языком C, будет сравнительно легко изучить Perl.

В операционной системе Windows

Чтобы получить возможность писать программы на языке Perl в среде Windows, нужно воспользоваться бесплатным пакетом ActivePerl от компании ActiveState. Его можно найти на сайте http://www.activestate.com.

Процедура установки этого пакета довольно проста. Он имеет формат .msi (Microsoft Installer). Выполнения каких-либо настроек в процессе установки пакета не требуется. В системную переменную PATH добавляется каталог **C:\perl\bin**.

СОВЕТ. Мы рекомендуем установить Perl в каталог **C:\perl** (вместо диска **C:** может быть и другой диск), а не в каталог **C:\Program Files**.

3.2.2. Первая программа на языке Perl в операционной системе UNIX

Запустите редактор **joe** и создайте в нем следующий текст:

```
#!/usr/bin/perl -w
print "Первая программа на языке Perl\n";
exit( 0 );
```

Сохраните его под именем, например, **first.pl**. Расширение «**.pl**» необязательно, но, как правило, программы на языке Perl имеют именно такое расширение. Чтобы вашу программу можно было исполнить, вы должны назначить соответствующие права доступа к файлу программы. Это можно сделать таким образом:

```
chmod 755 first.pl
```

Теперь можно запустить эту программу. Поскольку в ОС UNIX по умолчанию поиск программ в текущем каталоге не производится, то необходимо в команде запуска программы указывать символы «./», означающие текущий каталог.

```
./first.pl
```

Теперь объясним назначение каждой строки этой маленькой программы. Первая строка указывает полный путь к интерпретатору языка Perl. Хотя он является интерпретатором, но работают программы на языке Perl очень быстро, т. к. сначала производится преобразование исходного текста во внутренний формат, а затем уже выполнение. Поэтому многократно выполняемые фрагменты программного кода преобразуются во внутренний код только один раз. Символы **-w** в первой строке служат указанием для проверки объявлений всех переменных в программе. В нашей программе пока нет переменных. Обратите внимание на символы **#!** в начале первой строки. Их наличие обязательно. Полный путь к интерпретатору Perl может быть не только **/usr/bin/perl**, но и **/usr/local/bin/perl**. Как правило, сразу после инсталляции операционной системы FreeBSD интерпретатор Perl помещается в каталог **/usr/bin**, а в том случае, когда администратор системы устанавливает более новую версию Perl, он помещает ее в каталог **/usr/local/bin**. Но зачастую просто делают символическую ссылку в каталоге **/usr/local/bin** на **/usr/bin/perl**, позволяя таким образом использовать в программах оба пути: **/usr/bin/perl** и **/usr/local/bin/perl**, которые указывают на один и тот же интерпретатор Perl. Эта команда выглядит так:

```
ln -s /usr/bin/perl /usr/local/bin/perl
```

В следующей строке помещен вызов функции **print**. В языке Perl имеется и функция **printf**, которая по своим возможностям аналогична такой же функции из языка C, но в простых случаях используется функция print. Обратите внимание на отсутствие круглых скобок – это не ошибка. Perl допускает их отсутствие в том случае, когда оно не мешает интерпретатору произвести синтаксический разбор выражения. Символы **\n** означают переход на новую строку.

В последней строке помещена функция **exit** для завершения программы. Она возвращает значение 0 операционной системе.

Как и в языке C, в конце оператора ставится точка с запятой.

В операционной системе Windows

Создать исходный текст программы можно в редакторе Блокнот, входящем в состав операционной системы. Можно использовать для этой

цели встроенный редактор файлового менеджера FAR Manager или аналогичного ему.

Первая строка программы, содержащая полный путь к интерпретатору Perl, при запуске в среде Windows может отсутствовать.

ВНИМАНИЕ. Однако при использовании программы на Perl в качестве CGI-скрипта первая строка должна содержать либо полный путь к интерпретатору Perl, либо только его имя, если в системную переменную PATH добавлен путь к каталогу, в котором интерпретатор находится. Примеры можно найти в главе, посвященной CGI-программированию.

Для запуска программы в среде Windows необходимо сделать так:

```
perl first.pl
```

Запускать программу удобнее из файлового менеджера FAR Manager или аналогичного ему, поскольку программа выводит сообщения на стандартный вывод. Можно использовать и интерфейс командной строки Windows, но это не так удобно. Запускать программу с помощью «мыши» нельзя, т. к. окно, в которое выводится результат работы, закрывается сразу после завершения работы программы – прочитать выведенное сообщение невозможно.

3.2.3. Типы данных языка Perl

В языке Perl имеется три типа данных: **скаляры**, **массивы** и **ассоциативные массивы** (по-другому они называются **хеш-массивами**, или просто хешами). Покажем способы объявления переменных для этих типов данных, но сначала приведем некоторые объяснения. Если переменная объявлена с помощью ключевого слова **my**, то это означает, что переменная будет локальной, т. е. ее область видимости ограничена той функцией, в которой она объявлена, а время жизни – временем выполнения этой функции. Такие переменные можно объявлять и вне тела какой-либо функции (процедуры). В этом случае переменная доступна во всех процедурах, определения (тела) которых располагаются в тексте программы после ее объявления. Как правило, для объявления переменных используется ключевое слово **my**. Приведем несколько примеров объявлений переменных (обратите внимание на символы **$**, **@**, **%** перед именами переменных, массивов и хеш-массивов):

Скаляры

```
my $student;            # неинициализированная переменная
my $student = "Иванов"; # инициализированная переменная -
                        # строка
my $age = 19;           # инициализированная переменная -
```

```
                        # целое значение
my $weight = 75.8;      # инициализированная переменная -
                        # числовое значение с десятичной
                        # частью
```

Символом # обозначаются комментарии. Все символы на строке, расположенные после этого символа, игнорируются при выполнении программы.

Скалярные переменные не имеют конкретного типа, подобного типу данных в языке C. Важным отличием от языка C является то, что все заботы о выделении памяти для строк символов, массивов и хеш-массивов берет на себя Perl. В объявлении

```
my $student = "Иванов";
```

значением переменной $student будет не указатель на строку, а сама строка. Поэтому для соединения (конкатенации) строк используется простая операция «.»:

```
my $last_name = "Иванов ";
my $first_name = "Иван";
my $full_name;
$full_name = $last_name . $first_name;
```

При выполнении операций над скалярами Perl определяет по **содержимому** переменной, как с ней поступить.

Массивы

```
# массив фамилий студентов
# (обратите внимание на символ @ перед именем массива)
my @students = ( 'Иванов', 'Петров', 'Сидоров' );
# пустой массив
my @empty_array = ();
```

Для обращения к элементу массива используется тот же подход, что и в языке C: $students[2] даст нам значение 'Сидоров'. Обратите внимание, что при обращении к элементу массива символ @ изменяется на $. Счет элементов в массиве начинается с нуля. Символьные строки ограничиваются либо одинарными, либо двойными кавычками. В ряде случаев это имеет *принципиальное значение*, что будет показано в дальнейшем. Для записи значения в массив можно просто присвоить значение элементу:

```
$students[ 5 ] = 'Новиков';
```

Если в массиве нет элемента с номером 5 (а наш массив @students более короткий – всего три элемента), то в этом случае необходимое число

элементов будет добавлено автоматически. При этом элементы с индексами 3 и 4 получат неопределенные значения, которые в языке Perl обозначаются как **undef**.

Хеш-массивы

```
# пустой хеш-массив
# (обратите внимание на символ % перед именем хеш-массива)
my %empty_hash = ();
# хеш-массив, в котором записаны сведения о росте студентов
my %height = ( 'Иванов' => 176,
               'Петров' => 165,
               'Сидоров' => 190
             );
```

В хеш-массивах данные хранятся парами: так называемый **ключ** и значение. Символы => используются для удобства, но можно вместо них ставить запятую. В нашем примере символьные строки 'Иванов', 'Петров', 'Сидоров' являются ключами, а соответствующие им числа – значениями. Чтобы извлечь требуемое значение из хеш-массива, нужно знать ключ для этого значения, но не нужно знать место хранения (например, индекс в обычном массиве указывает на место хранения элемента данных). Поэтому, чтобы получить рост студента Иванова, мы пишем:

```
$height{ 'Иванов' }
```

Обратите внимание, что при обращении к элементу символ % изменяется на $, а ключ помещается в фигурные скобки. Для записи значения в хеш-массив можно просто присвоить значение элементу с указанным ключом:

```
$height{ 'Новиков' } = 187;
```

3.2.4. Вторая программа на языке Perl в операционной системе UNIX

Сейчас мы можем написать более сложную программу. Рекомендуем внимательно читать все комментарии, приведенные в тексте этой и всех следующих программ, поскольку комментарии носят учебный характер, в них содержатся полезные и важные сведения, советы, подсказки и т. д.

Запустите редактор **joe** и создайте в нем следующий текст:

```
#!/usr/bin/perl -w
print "Вторая программа на языке Perl\n";

# скалярные переменные
```

```
my $last_name = "Иванов ";
my $first_name = "Иван";
my $full_name;

# используем операцию конкатенации строк
$full_name = $last_name . $first_name;

# используем операцию конкатенации строк
print "Полное имя: ". $full_name . "\n";

# теперь вставим имя переменной внутрь строки в двойных
# кавычках
print "Повторим еще раз: $full_name\n";

# то же, но кавычки одинарные

# ПРИМЕЧАНИЕ. Когда используются двойные кавычки, тогда все
# переменные внутри такой строки заменяются их значениями.
# В случае одинарных кавычек такой подстановки значений
# не происходит.
print 'Найдите отличия от предыдущей команды: $full_name\n';

# пропустим пустую строку
print "\n";

# Массивы

# массив фамилий студентов
my @students = ( 'Иванов', 'Петров', 'Сидоров' );
# пустой массив
my @empty_array = ();

# используем операцию конкатенации строк
print $students[ 0 ] . "\n";

# теперь вставим имя массива внутрь строки в двойных кавычках
print "$students[ 1 ]\n";

# запишем значение в массив
$students[ 5 ] = 'Новиков';

# теперь выведем значения (обратите внимание на
# предупреждения)
print "$students[ 3 ]\n";
print "$students[ 4 ]\n";
print "$students[ 5 ]\n";

# Хеш-массивы

my %empty_hash = ();  # пустой хеш-массив

# хеш-массив, в котором записаны сведения о росте студентов
my %height = ( 'Иванов' => 176,
```

```
                'Петров' => 165,
                'Сидоров' => 190
              );

# получим рост студента Иванова
print $height{ 'Иванов' } . "\n";

# можно и так
print "Рост студента Петрова: $height{ 'Петров' }\n";

# запишем значение в хеш-массив
$height{ 'Новиков' } = 187;

# теперь выведем его
print "$height{ 'Новиков' }\n";

exit( 0 );
```

Перед запуском программы не забудьте предоставить владельцу файла с помощью команды **chmod** право выполнения программы.

При выполнении программы анализируйте сообщения, выводимые на экран, и сопоставляйте их с исходным текстом. Рекомендуем также внести какие-либо изменения в исходный текст и вновь запустить программу. В качестве изменений можно порекомендовать создать еще один массив или хеш-массив, изменить число элементов в массиве и т. д.

В операционной системе Windows

Создайте текст программы и запустите ее, следуя рекомендациям, приведенным для первой программы на языке Perl.

3.2.5. Процедуры языка Perl

Этот вопрос мы изучим на примере третьей программы. Запустите редактор **joe** и создайте в нем следующий текст:

```
#!/usr/bin/perl -w

print "Процедуры в языке Perl\n";

proc_1();

# печатаем строку, возвращаемую процедурой proc_2()
print proc_2() . "\n";

exit( 0 );

sub proc_1
```

```
{
  print "Процедура proc_1\n";
}

sub proc_2
{
  print "Процедура proc_2\n";
  # возвращаем строку (а не указатель, в отличие от языка C)
  return "FINISH";
}
```

Как вы видите, для создания процедур (функций) используется ключевое слово **sub**.

Перед запуском программы не забудьте назначить право выполнения программы владельцу файла.

3.2.6. Организация выбора и циклов в языке Perl

Конструкции для организация выбора и циклов в языке Perl похожи на аналогичные конструкции в языке C. Рассмотрим их на примере очередной программы.

```
#!/usr/bin/perl -w

#use strict;

print "Конструкции выбора и циклы на языке Perl\n";

# объявим скалярные переменные
my $i;      # индекс в массиве
my $stud;   # фамилия студента

# массив фамилий студентов
my @students = ( 'Иванов', 'Петров', 'Сидоров' );

# запишем значение в массив
# ПРИМЕЧАНИЕ. Perl сам увеличит длину массива на три элемента.
$students[ 5 ] = 'Новиков';

print "Вывод массива\n";

# теперь выведем значения (обратите внимание на
# предупреждения)
# ПРИМЕЧАНИЕ. Этот цикл организован, как на языке C.
#             Переменная $#students хранит индекс последнего
#             элемента массива.
for ( $i = 0; $i <= $#students; $i++ )
{
  print "Элемент номер $i равен: $students[ $i ]\n";
```

```
}

print "\nВывод массива с проверкой значений\n";
# теперь будем выполнять проверку значений перед выводом
# их на печать
for ( $i = 0; $i <= $#students; $i++ )
{
  # обратите внимание на операцию сравнения eq - она означает
  # проверку на равенство строковых значений переменных,
  # для проверки на неравенство служит операция ne,
  # (например, $students[ $i ] ne 'Петров')
  if ( $students[ $i ] eq 'Петров' )
  {
    print "Элемент 'Петров' имеет номер $i\n";
  }
  # обратите внимание: не else if, а elsif; undef означает
  # отсутствие определенного значения, обычно такое значение
  # имеют неинициализированные переменные
  elsif ( $students[ $i ] eq undef )
  {
    print "Элемент номер $i не имеет определенного " .
          "значения\n";
  }
}

print "\nВывод массива без использования индекса в массиве\n";

# теперь покажем организацию цикла без использования индекса
# в массиве
foreach $stud ( @students )
{
  print "Студент $stud\n";
}

print "\nДальнейшее упрощение цикла с использованием "
     ."переменной \$_\n";
# ПРИМЕЧАНИЕ. Обратите внимание на символ "\" перед
#             символом "$". Это называется экранированием.
#             Оно необходимо, т.к. символ "$" имеет в языке
#             Perl специальное значение.

# Теперь покажем организацию цикла также и без использования
# переменной, которой присваиваются поочередно значения
# элементов массива
foreach ( @students )
{
  # переменная $_ является одним из самых часто используемых
  # элементов языка Perl. Значение ей присваивается неявно,
  # без прямого участия программиста. Эта переменная всегда
  # присутствует во всех операциях, когда происходит перебор
  # элементов массива, чтение строк из файла и т.д. Она
  # позволяет избежать обилия переменных.
  if ( $_ eq undef )   # переменная $_ равна текущему элементу
```

```
  {
    print "Неопределенное значение элемента\n";
  }
  else
  {
    print "$_\n";
  }
}

print "\nВыход из цикла с использованием оператора last\n";

# теперь покажем выход из цикла
foreach ( @students )
{
  # переменная $_ равна текущему элементу
  if ( $_ eq 'Петров' )
  {
    print "Мы нашли Петрова - выходим из цикла\n";
    last;                # выход из цикла
  }
  else
  {
    print "$_\n";
  }
}

# Хеш-массивы

# хеш-массив, в котором записаны сведения о росте студентов
my %height = ( 'Иванов' => 176,
               'Петров' => 165,
               'Сидоров' => 190
             );

print "\nВывод хеш-массива\n";

# получить значения всех элементов хеш-массива можно так:
# ПРИМЕЧАНИЕ. В этом случае функция keys выбирает из
#             хеш-массива значения всех ключей (т.е. фамилии
#             студентов) и формирует из них массив,
#             а конструкция foreach перебирает все элементы
#             этого массива, присваивая поочередно их значения
#             переменной $stud_name.
foreach $stud_name ( keys %height )
{
  print "Рост студента $stud_name: $height{ $stud_name }\n";
}

print "\nВывод хеш-массива с использованием переменной \$_\n";

# получить значения всех элементов хеш-массива можно так:
# ПРИМЕЧАНИЕ. В этом случае функция keys выбирает
#             из хеш-массива значения всех ключей
```

```
#             (т.е. фамилии студентов) и формирует из них
#             массив, а конструкция foreach перебирает все
#             элементы этого массива, присваивая поочередно их
#             значения "теневой" переменной $_.
foreach ( keys %height )
{
  # для сравнения на точное равенство числовых значений
  # используется оператор "==", на неравенство: >, <, >=, <=
  if ( $height{ $_ } == 165 )
  {
    print "Студент $_ имеет низкий рост: $height{ $_ }\n";
  }
  else
  {
    print "Студент $_ имеет высокий рост: $height{ $_ }\n";
  }
}

exit( 0 );
```

Эта программа выводит информации столько, что она не умещается на одном экране. Для просмотра всей выведенной информации есть три способа. Первый заключается в использовании известной вам команды **more** таким образом:

```
./program.pl | more
```

Второй способ заключается в том, чтобы позволить программе вывести все, что она хочет, а затем нажать клавишу Scroll Lock. После ее нажатия можно использовать клавиши управления курсором для просмотра экрана вверх и вниз. В таком экранном буфере хранится примерно 130–150 строк текста (т. е. пять–шесть объемов экрана). Для прекращения просмотра буфера опять нажмите клавишу Scroll Lock.

Третий способ – это использование переадресации стандартного вывода в файл, который затем можно просмотреть с помощью любого доступного средства:

```
./program.pl > some_file.txt
```

Попробуйте убрать модификатор -w, вы заметите, что предупреждения на экран выводиться не будут. Попробуйте теперь добавить в начало файла после самой первой строки **#!/usr/bin/perl** такую строку:

```
use strict;
```

Вообще-то эта строка уже приведена в тексте программы, но она закомментирована. Вы просто удалите символ #, чтобы раскомментировать ее, и вы увидите, что теперь Perl требует от вас предварительного объявле-

ния всех переменных. Объявите переменную $stud_name в любом месте программы до ее первого использования.

В операционной системе Windows

В операционной системе Windows клавиша Scroll Lock не имеет таких возможностей, как описано выше.

3.2.7. Работа с файлами

Работать с файлами можно, направляя их на стандартный ввод вашей программы с помощью переадресации ввода. Создайте программу:

```
#!/usr/bin/perl -w

print "Работа с файлами\n";

my $i = 0;

# ПРИМЕЧАНИЕ. Подумайте, как запустить эту программу, чтобы
#             подать ей какой-нибудь файл на стандартный ввод.
#             Считываем файл построчно со стандартного ввода и
#             построчно выводим его, нумеруя строки
while ( <STDIN> )
{
  # вспомните, что означают символы "." В данном случае
  # переменная $_ "впитывает" в себя целую строку файла
  print ++$i . " " . $_;
}

exit( 0 );
```

Теперь покажем, как открыть файл внутри программы.

```
#!/usr/bin/perl -w

print "Работа с файлами\n";

my $i = 0;

# переменная ARGV является массивом, хранящим параметры,
# переданные программе в командной строке.
# ВАЖНОЕ ОТЛИЧИЕ от языка C: имя самой программы не является
# элементом этого массива, таким образом, элемент $ARGV[ 0 ] -
# это первый параметр, $ARGV[ 1 ] - второй параметр и т.д.
if ( $#ARGV != 1 )     # требуем ДВА параметра
{
  # выводим сообщение об ошибке на стандартное устройство
```

```
  # ошибок.
  # ПРИМЕЧАНИЕ. Обратите внимание на отсутствие запятой
  # после слова STDERR.
  print STDERR "Укажите параметры: имя входного файла и " .
               "имя результирующего\n";
  exit( 1 );    # возвращаем операционной системе значение 1
}

# откроем файл в режиме чтения

# ПРИМЕЧАНИЕ. Функция open() при успешном открытии файла
#             возвращает ненулевое значение. Символы "$!"
#             означают текст сообщения об ошибке, выдаваемый
#             операционной системой. Попробуйте указать
#             неверное имя исходного файла, чтобы увидеть это
#             сообщение.
#             Функция die аналогична функции exit(), но
#             предварительно выводит сообщение, указанное
#             в кавычках.
#             Операция "||" - это логическое "ИЛИ". Логическое
#             "И" обозначается "&&".
#             READ - дескриптор файла: их записывают без
#             кавычек, как правило, заглавными буквами.
open( READ, "< $ARGV[ 0 ]" ) ||
    die "Не могу открыть файл $ARGV[ 0 ]: $!\n";

# откроем файл в режиме записи.
# ПРИМЕЧАНИЕ. Для открытия файла в режиме дополнения вместо
#             префикса ">" нужно поставить ">>", в режиме
#             чтения/записи "+<" или "+>>"
open( WRITE, "> $ARGV[ 1 ]" ) ||
    die "Не могу открыть файл $ARGV[ 1 ]: $!\n";

# считываем файл построчно и построчно выводим его,
# нумеруя строки
while ( <READ> )
{
  # вспомните, что означают символы "." В данном случае
  # переменная $_ "впитывает" в себя целую строку файла.
  # ПРИМЕЧАНИЕ. Обратите внимание на отсутствие запятой после
  #             слова WRITE.
  print WRITE ++$i . " " . $_;
}

# закроем файлы:
close( READ ) || die "Не могу закрыть файл $ARGV[ 0 ]: $!\n";
close( WRITE ) || die "Не могу закрыть файл $ARGV[ 1 ]: $!\n";

exit( 0 );
```

3.2.8. Регулярные выражения языка Perl

Регулярные выражения (**regular expressions**) – это средство, предназначенное для анализа и обработки символьных строк: поиска шаблонов, замены одних фрагментов текста на другие. Регулярные выражения являются очень мощным средством, позволяющим выполнить сложные операции, используя компактные операторы.

```
#!/usr/bin/perl -w

print "Работа с регулярными выражениями\n";
my $i = 0;

# ПРИМЕЧАНИЕ. Подумайте, как запустить эту программу, чтобы
#             подать ей какой-нибудь файл на стандартный ввод.

# Считываем файл построчно со стандартного ввода и построчно
# выводим его, нумеруя строки, отбрасывая при этом комментарии
# (как полные строки, так и комментарии, которые находятся на
# одной строке с операторами), отбрасывая также и пустые
# строки (или строки, состоящие из одних пробелов).
# ПРИМЕЧАНИЕ. Наша программа очень простая, поэтому она
#             выполняет работу с небольшой ошибкой, которая
#             может проявиться, если на вход этой программе
#             подать, например, ее собственный текст.
#             Попробуйте найти проявление ошибки.
while ( <STDIN> )
{
  # в данном случае переменная $_ "впитывает" в себя целую
  # строку файла
  # ПРИМЕЧАНИЕ. Данное регулярное выражение означает
  #             следующее: во-первых, оператор =~ служит для
  #             выполнения действий над строкой, которая
  #             хранится в переменной $_; символ "s" означает
  #             замену одного фрагмента строки на другой -
  #             заменяемый фрагмент помещается между первой
  #             парой символов "/", а заменяющий - между
  #             второй парой символов "/". В данном случае
  #             заменяющим фрагментом является пустая строка.
  #             Внутри заменяемого фрагмента символ "#"
  #             понимается буквально, символ "." означает
  #             любой символ, символ "*" означает, что
  #             количество этих самых любых символов может
  #             быть также любым - от 0 и больше. После
  #             применения такого регулярного выражения
  #             к переменной $_ она изменится. Если в ней
  #             хранилась строка-комментарий, то останется
  #             пустая строка. Если комментарий занимал лишь
  #             часть строки, то он будет отброшен. Если же в
  #             строке не было символа "#", то она не
  #             претерпит никаких изменений.
```

```
  $_ =~ s/#.*//;

  # теперь сравним полученную строку с требуемым шаблоном,
  # который допускает наличия в строке только пробельных
  # символов (пробел, табуляция, новая строка), на что
  # указывает комбинация "\s". Символ "+" указывает, что этих
  # символов должно быть не меньше одного.
  # Символы "^" и "$" означают начало и конец строки
  # соответственно. Они не хранятся в строке, а являются
  # условными.
  # ПРИМЕЧАНИЕ. Обратите внимание, что в предыдущем шаблоне
  #             был символ "s" перед шаблоном, а внутри
  #             шаблона было три символа "/". В данном шаблоне
  #             нет символа "s" перед первым символом "/", а
  #             самих символов "/" всего два. Это объясняется
  #             тем, что в первом случае мы должны были найти
  #             определенный фрагмент текста и заменить его
  #             другим (пусть даже пустым) фрагментом, а во
  #             втором случае нам нужно лишь найти
  #             определенный фрагмент, не модифицируя его.
  if ( $_ =~ /^\s+$/ )
  {
    # переход к началу цикла, минуя оставшуюся часть тела
    # цикла
    next;
  }

  # вспомните, что означают символы "."
  print ++$i . " " . $_;
}

exit( 0 );
```

3.2.9. Вызов справки по языку Perl

Для вызова справки по языку Perl используйте команду **man**:

```
man perl
```

Обратите внимание, что справочная информация по языку разделена на отдельные секции, список которых приведен в заглавной справочной странице (мы говорим «страница», т. к. полное название электронных руководств в ОС UNIX – manual pages, а на жаргоне они называются просто «маны»). Например, для получения справки по структурам данных, используемым в языке Perl, нужно ввести

```
man perldata
```

Для просмотра справки используйте клавиши управления курсором, а для выхода – клавишу Q.

Для получения информации о параметрах запуска Perl введите

```
perl -h
```

В операционной системе Windows

В состав пакета ActivePerl входит полная документация в формате html. Для просмотра документации откройте файл **index.html** в каталоге **C:\perl\html** (конечно, логический диск может быть и другим). Если изображение в браузере будет искажаться (например, первые несколько колонок текста в левом и правом фреймах html-документа не видны на экране), то можно попробовать отключить использование таблицы стилей. Простейший способ сделать это – переименовать файл **Active.css** (удалять его, конечно, не стоит).

Контрольные вопросы и задания

1. К классу каких языков относится язык Perl?

2. Для чего нужна первая строка программы на языке Perl:
```
#!/usr/bin/perl -w
```

3. Какие типы данных есть в этом языке?

4. Что такое хеш-массивы? Что такое **ключ** в хеш-массиве? Как можно обратиться к элементу хеш-массива? Как можно записать новое значение в хеш-массив?

5. Какие способы организации циклов на языке Perl вы знаете?

6. Что означает переменная **$_** и как она используется?

7. Каким образом можно соединить две символьных строки на языке Perl? Может ли функция возвращать символьную строку? В чем отличие от языка C?

8. Что такое **регулярные выражения**? Приведите пример простейшего из них?

9. Как открыть файл в программе на языке Perl в режиме дополнения, в режиме чтения/записи, в режиме только чтения?

10. Каким образом можно организовать построчное чтение текстового файла в программе на языке Perl?

11. Что означает выражение $ARGV[0] в языке Perl? В чем отличие от языка C?

12. Внимательно изучите все программы, приведенные в этой главе. Все комментарии в этих программах носят учебный характер и потому их нужно тщательно изучить. Затем попытайтесь модифицировать программы и, выполняя их, смотрите, что у вас получается. Модифицирование может заключаться в добавлении новых переменных, изменении числа элементов в массивах или хеш-массивах, оформлении отдельных фрагментов программы в виде процедур, открытии в программах файлов в различных режимах (вместо режима дополнения – в режиме чтения) и т. д. Фантазируйте, ведь программирование – процесс творческий. Иначе ничему не научитесь! Руководствуйтесь известным афоризмом К. Пруткова: «Бросая в воду камешки, смотри на круги, ими образуемые, чтобы такое занятие не было пустою забавою». За точность воспроизведения цитаты не ручаемся, но смысл именно такой.

4. Библиотеки

Библиотеки подпрограмм являются мощным средством повышения надежности программ и ускорения процесса их разработки. Мы рассмотрим процесс создания библиотек на языке C/C++, а также порядок создания модулей (пакетов) на языке Perl. Эти модули (пакеты) можно считать аналогом библиотек в их традиционном понимании.

Как и в предыдущих главах учебного пособия, мы сначала рассмотрим процедуры создания библиотек в среде операционной системы UNIX (на примере FreeBSD), а затем коротко опишем особенности этого процесса, имеющие место в среде операционной системы Windows.

4.1. Язык C/C++

Библиотеки содержат те подпрограммы, которые часто используются при разработке прикладных и системных программ. Как правило, библиотека содержит подпрограммы, предназначенные для решения определенной задачи (или ряда сходных задач), например, таковы библиотеки для создания пользовательского интерфейса.

Библиотеки могут различаться по способу создания и использования. Выделяют следующие виды библиотек: статические, разделяемые, а также библиотеки динамической загрузки (компоновки).

4.1.1. Статические библиотеки

Статические библиотеки в операционной системе UNIX – это просто архивные файлы, содержащие группу откомпилированных модулей. Процедура создания подобных библиотек начинается с написания исходных текстов тех функций, которые предполагается включить в библиотеку.

Выберем в качестве примера такую предметную область, которая была бы понятна каждому студенту младших курсов. Пусть это будет элементарная геометрия.

Выделим две подобласти в выбранной предметной области: вычисление параметров для круга (окружности) и для квадрата. Для каждой подобласти создадим отдельный исходный модуль. Каждый из двух модулей **circle.c** и **square.c** будет содержать две функции. Таким образом, это очень простой случай, который служит только в качестве иллюстрации и не может рассматриваться как пример реальной библиотеки.

Программа **circle.c**

```
// ------------------------------------------------------
// Модуль: функции для работы с кругом и окружностью
// ------------------------------------------------------

#define   PI   3.14159265
```

```
// вычисление площади круга
// параметр - радиус окружности
float circle_area( float rad )
{
  float area;
  // площадь равна: пи * радиус в квадрате
  area = PI * rad * rad;
  return ( area );
  // можно сократить текст, написав:
  // return ( PI * rad * rad );
}

// вычисление длины окружности
// параметр - радиус окружности
float circle_len( float rad )
{
  float len;
  // длина окружности равна: 2 * пи * радиус
  len = 2 * PI * rad;
  return ( len );
}
```

Программа **square.c**

```
// ------------------------------------------------------------
// Модуль: функции для работы с квадратом
// ------------------------------------------------------------

// вычисление площади квадрата
// параметр - сторона квадрата
float square_area( float side )
{
  float area;
  // площадь равна: длина стороны в квадрате
  area = side * side;
  return ( area );
}

// вычисление периметра квадрата
// параметр - сторона квадрата
float square_perim( float side )
{
  float len;
  // периметр равен: 4 * длина стороны
  len = 4 * side;
  return ( len );
}
```

Обратите внимание, что в текстах этих модулей нет функции main(). Также необходимо помнить о том, что все функции и переменные, которые

должны быть доступны для вызова (обращения) извне библиотеки, должны объявляться без ключевого слова **static**.

На следующем шаге необходимо скомпилировать исходные тексты модулей в объектные модули:

```
gcc -c circle.c
gcc -c square.c
```

Получим два объектных файла: **circle.o** и **square.o**. Из них уже можно создать библиотеку. Команда для ее создания будет такой:

```
ar rcs libgeometry.a circle.o square.o
```

В этой команде следующие компоненты:

– **ar** – программа-библиотекарь;

– параметр **r** означает, что необходимо добавить новые модули в библиотеку (модули из библиотеки можно также удалять);

– параметр **s** указывает, что в библиотеке должен быть создан индекс, который служит для ускорения поиска в ней необходимых модулей;

– последний параметр **c** означает, что файл библиотеки необходимо создать, поскольку его еще нет.

Вместо параметра **s** может быть использована специальная команда ranlib, позволяющая создать индекс для уже сформированной библиотеки.

Имя библиотеки не может быть произвольным. Оно должно содержать префикс **lib** и расширение **.a**.

Все приведенные команды можно собрать в единый командный файл, который мы назовем **static_lib.sh**.

Файл **static_lib.sh**

```
#!/bin/sh
# ------------------------------------------------------------
# Создание статической библиотеки
# ------------------------------------------------------------

# Компилируем исходные модули для библиотеки
gcc -c circle.c
gcc -c square.c

# Включаем скомпилированные модули в библиотеку libgeometry.a
ar rcs libgeometry.a circle.o square.o
```

Создав этот командный файл, назначьте права доступа к нему:

```
chmod 755 static_lib.sh
```

Теперь в случае внесения изменений в исходные тексты модулей, входящих в состав библиотеки, можно ее переформировать, запустив данный командный файл:

```
./static_lib.sh
```

Чтобы узнать, какие модули входят в состав созданной библиотеки, используется команда **ar** с параметром **t**:

```
ar t libgeometry.a
```

Она выведет на экран следующее:

```
circle.o
square.o
```

Можно получить более подробную информацию относительно функций, переменных и других объектов, представленных в библиотеке. Такой сервис предоставляет команда **nm**:

```
nm libgeometry.a
```

Для нашей простой библиотеки она выведет:

```
circle.o:
00000000 T circle_area
00000024 T circle_len

square.o:
00000000 T square_area
0000001c T square_perim
```

Для того чтобы лучше понимать вывод программы nm, рекомендуем обратиться к электронному руководству **man**.

Итак, наша простая библиотека создана. Проиллюстрируем ее применение на простом примере.

Программа demo_lib.c

```
// ------------------------------------------------------------
// Программа, использующая функции из библиотеки
// ------------------------------------------------------------

#include <stdio.h>

#include "geometry.h"

int main( void )
```

```
{
  float rad;
  float side;

  printf( "Hello, World!\n\n" );

  printf( "Введите радиус окружности: " );
  scanf( "%f", &rad );
  printf( "\n%f\n", rad );
  printf( "Длина окружности: %f\n", circle_len( rad ) );
  printf( "Площадь круга: %f\n", circle_area( rad ) );

  printf( "\nВведите длину стороны квадрата: " );
  scanf( "%f", &side );
  printf( "\n%f\n", side );
  printf( "Периметр квадрата: %f\n", square_perim( side ) );
  printf( "Площадь квадрата: %f\n", square_area( side ) );

  return 0;
}
```

Обратите внимание на наличие в начале программы директивы включения заголовочного файла **geometry.h**. Этот файл содержит прототипы библиотечных функций. Его необходимо включать в каждый модуль, содержащий обращения (вызовы) к библиотечным функциям. Приведем текст этого файла.

Файл **geometry.h**

```
// ------------------------------------------------------------
// Прототипы функций из библиотеки
// ------------------------------------------------------------

float circle_area( float rad );
float circle_len( float rad );

float square_area( float side );
float square_perim( float side );
```

Теперь мы можем скомпилировать тестовую программу, использующую созданную библиотеку:

```
gcc -o demo_static_lib demo_lib.c -L. -lgeometry
```

В этой команде параметр «–L.» означает, что поиск библиотек необходимо производить не только в стандартных каталогах операционной системы, но и в текущем каталоге, который обозначается символом «.». Параметр –lgeometry указывает компилятору (точнее говоря, редактору связей, или компоновщику) имя нестандартной библиотеки. При этом префикс «lib» и расширение «.a» опускаются.

Для запуска полученной программы введите

```
./demo_static_lib
```

Если при тестировании библиотеки ошибок не обнаружено, то теперь файл **geometry.h** необходимо разместить в системном каталоге **/usr/local/include**. Если предполагается дальнейшее развитие разрабатываемой библиотеки, то имеет смысл создать подкаталог, например, **geometry**, в каталоге **/usr/local/include** и поместить наш заголовочный файл в этот подкаталог. В принципе возможно размещение файла **geometry.h** в системном каталоге **/usr/include**, но мы не рекомендуем этого делать, поскольку в данном каталоге традиционно располагаются только заголовочные файлы, входящие в состав операционной системы.

Аналогично и саму библиотеку следует перенести в системный каталог **/usr/local/lib**. После проведенных перемещений файлов команда для компиляции программы **demo_lib.c** изменится (вводить эту команду необходимо в одной командной строке, не нажимая клавишу Enter, пока не введена вся команда):

```
gcc -o demo_static_lib demo_lib.c -I/usr/local/include
-L/usr/local/lib -lgeometry
```

В этом варианте команды компиляции тестовой программы параметр –L указывает на тот каталог, в который вы перенесли библиотеку. В команде появился еще один параметр: -I/usr/local/include. Он указывает компилятору каталог для поиска заголовочных (включаемых) файлов. При этом необходимо учитывать, что если вы разместили файл **geometry.h** непосредственно в каталоге **/usr/local/include**, то эта команда будет работать. Но если вы поместили файл **geometry.h** в подкаталог **geometry**, то тогда необходимо в тексте программы **demo_lib.c** изменить имя файла в директиве #include таким образом:

```
#define "geometry/geometry.h"
```

Таким образом, путь, указанный в параметре –I, и путь (или только имя файла), указанный в директиве #include, должны в совокупности представлять собой полный путь к заголовочному файлу **geometry.h**.

Добавим еще, что если вы поместили библиотеку **libgeometry.a** в системный каталог **/lib** или **/usr/lib**, то параметр –L компилятору не нужен. Однако следует учитывать, что в этих каталогах традиционно находятся только библиотеки, входящие в состав операционной системы.

При использовании среды MSYS общая стратегия создания статической библиотеки остается такой же, как и в среде ОС FreeBSD. Однако следует помнить, что среда MSYS это все-таки не естественная среда ОС FreeBSD. Это выражается в том, что возможны проблемы (к счастью, решаемые) при выполнении консольных приложений. Такие проблемы решаются путем выбора терминала **sh.exe** вместо **rxvt.exe**. Как это сделать, описано в главе 1.

Нужно также учитывать, что при запуске среды MSYS выполняется операция монтирования файловых систем в стиле UNIX на каталоги Windows. Например, каталогу /usr/local/lib соответствует Windows-каталог **C:\msys\1.0\local\lib**. При копировании библиотеки **libgeometry.a** в каталог **/usr/local/lib** файл библиотеки окажется в каталоге **C:\msys\1.0\local\lib**. Аналогично, при копировании заголовочного файла **geometry.h** в каталог **/usr/local/include** он окажется в Windows-каталоге **C:\msys\1.0\local\include**.

Запускать исполняемые файлы, скомпилированные со статическими библиотеками, можно не только в среде MSYS, но и непосредственно в среде Windows. Для запуска консольных приложений можно использовать командную строку Windows или файловый менеджер, например, FAR.

Важный вопрос – тип кодировки символов, используемой в данной операционной системе. В системе FreeBSD это кодировка KOI8-R, а в среде Windows традиционной является кодировка CP1251, но может также использоваться и кодировка CP866. Для смены кодировки в командной строке Windows служит команда **chcp**. Для перекодирования исходных текстов программ можно использовать специальные программы-перекодировщики. Подходит для этой цели и редактор **emacs**.

Когда вы скопируете заголовочный файл и файл библиотеки в отведенные для них места, следующая команда должна выполняться так же, как и в ОС FreeBSD:

```
gcc -o demo_static_lib demo_lib.c -I/usr/local/include
-L/usr/local/lib -lgeometry
```

Запустить полученный исполняемый файл **demo_static_lib.exe** можно как в среде MSYS, так и непосредственно в Windows.

4.1.2. Разделяемые библиотеки

Термин «разделяемая» (shared) означает – совместно используемая. Основная идея библиотеки этого типа такова. При компоновке исполняемого файла объектный код модулей, входящих в библиотеку, не включает-

ся (не интегрируется) в исполняемый файл. В исполняемом файле есть лишь ссылки на библиотечные функции, машинный (объектный) код которых загружается в оперативную память компьютера при загрузке разделяемой библиотеки. Загрузка же самой библиотеки в память выполняется при запуске исполняемого файла. В том случае, если такая библиотека уже присутствует в оперативной памяти, то вторая ее копия не загружается.

Разделяемая библиотека не является, в отличие от статической библиотеки, архивным файлом, а имеет более сложную структуру. Поскольку код библиотечных функций может быть размещен в памяти в области, адрес которой на этапе компоновки исполняемого файла еще не известен, то компиляция исходных текстов этих функций производится с параметром -fPIC (Position Independent Code). А при создании библиотеки из объектных модулей используется параметр –shared, который указывает на тип результирующего файла – разделяемая библиотека.

Для удобства использования команд соберем их в командный файл **shared_lib.sh**.

Файл **shared_lib.sh**

```
#!/bin/sh
# ------------------------------------------------------------
# Создание разделяемой библиотеки
# ------------------------------------------------------------

# Компилируем исходные модули для библиотеки
gcc -fPIC -c circle.c
gcc -fPIC -c square.c

# Включаем скомпилированные модули в библиотеку libgeometry.so
gcc -shared -Wl,-soname=libgeometry.so.1 \
-o libgeometry.so.1.0.1 circle.o square.o
```

Сделаем пояснения к команде формирования библиотеки. В процессе создания и использования разделяемой библиотеки используются три вида имен. Первое из имен – это то имя, которое передается компоновщику (редактору связей) при создании исполняемого файла. В нашем случае это имя **libgeometry.so**, причем в командной строке оно указывается в форме –lgeometry. Второе имя – это так называемое soname (на русский язык этот термин можно перевести, как «имя разделяемого объекта»). В нашем случае это имя **libgeometry.so.1**. Обратите внимание, что в команде создания библиотеки параметр –Wl является параметром компоновщика. Все выражение -Wl,-soname=libgeometry.so.1 не содержит пробелов. Значение параметра soname записывается в специальное внутреннее поле разделяемой библиотеки. Так как библиотека является файлом, то она имеет и третье имя, т. е. имя файла. В нашем случае это **libgeometry.so.1.0.1**. В этом имени цифры 1.0.1 означают номер версии, младший номер версии и номер выпуска (release). Номер выпуска может не использоваться. Такая

трехуровневая система именования библиотек позволяет гибко использовать новые и старые версии библиотеки как при создании новых программ, использующих новую версию библиотеки, так и при запуске старых программ, построенных на основе старых версий одной и той же библиотеки.

И последнее замечание к команде создания библиотеки. Символ «\» позволяет перенести длинную команду на следующую строку. Сразу после этого символа должен идти символ новой строки, т. е. введя символ «\», нужно сразу нажать клавишу Enter.

Использование разделяемой библиотеки можно рассматривать как двухэтапный процесс. На первом этапе, т. е. при создании исполняемого файла, компоновщик (редактор связей) должен суметь найти разделяемую библиотеку, чтобы проверить, что ни один из символов, на которые имеются ссылки в исполняемом файле, в библиотеке не отсутствует. На втором этапе, т. е. на этапе запуска исполняемого файла, системный динамический загрузчик должен быть в состоянии найти разделяемую библиотеку в одном из системных каталогов, чтобы загрузить ее в оперативную память.

Когда разделяемая библиотека создана, то для ее использования нужно создать символические ссылки с именами, о которых речь шла выше. Выполните следующие команды в текущем каталоге (в котором и находится файл **libgeometry.so.1.0.1**):

```
ln -sf libgeometry.so.1.0.1 libgeometry.so.1
ln -sf libgeometry.so.1 libgeometry.so
```

Команда создания исполняемого файла из исходного файла **demo_lib.c**, текст которого приведен выше, выглядит так (с учетом того, что заголовочный файл **geometry.h** уже находится в системном каталоге, а разделяемая библиотека **libgeometry.so.1.0.1** – пока еще в текущем каталоге):

```
gcc -o demo_shared_lib demo_lib.c -I/usr/local/include -L.
-lgeometry
```

С помощью команды **ldd** можно узнать, какие разделяемые библиотеки используются при запуске исполняемого файла **demo_shared_lib**:

```
ldd ./demo_shared_lib
```

На экран будет выведено следующее:

```
demo_shared_lib:
     libgeometry.so.1 => not found (0x0)
     libc.so.6 => /lib/libc.so.6 (0x2807a000)
```

Из этого сообщения следует, что динамический загрузчик не может найти разделяемую библиотеку **libgeometry.so.1**. Чтобы все-таки протестировать созданную библиотеку, находящуюся в текущем каталоге, запустите программу **demo_shared_lib** таким образом:

```
LD_LIBRARY_PATH="."  ./demo_shared_lib
```

Если библиотечные функции работают правильно, то можно скопировать библиотеку в каталог **/usr/lib** и сделать такие же символические ссылки, какие вы делали в текущем каталоге:

```
cp libgeometry.so.1.0.1 /usr/lib
cd /usr/lib
ln -sf libgeometry.so.1.0.1 libgeometry.so.1
ln -sf libgeometry.so.1 libgeometry.so
```

Вернитесь в свой рабочий каталог и снова выполните команду

```
ldd ./demo_shared_lib
```

Теперь картина будет другая:

```
demo_shared_lib:
     libgeometry.so.1 => /usr/lib/libgeometry.so.1 (0x2807a000)
     libc.so.6 => /lib/libc.so.6 (0x2807c000)
```

После копирования библиотеки в системный каталог команду для компилирования исполняемого файла можно упростить:

```
gcc -o demo_shared_lib demo_lib.c -I/usr/local/include
-lgeometry
```

Предположим, что мы захотели улучшить нашу библиотеку и с этой целью создали новые версии обоих модулей, из которых состоит библиотека. Главное изменение заключается в возможности выводить не только результат вычислений, но и формулу, по которой проводился расчет.

Вот новые исходные тексты.

Программа **circle2.c**

```
// ------------------------------------------------------------
// Модуль: функции для работы с кругом и окружностью
// Версия 2
// ------------------------------------------------------------

#include <stdio.h>

#define   PI   3.14159265
```

```
// вычисление площади круга
// параметры - радиус окружности
//             и признак вывода формулы на экран
float circle_area( float rad, int formula )
{
  float area;
  // площадь равна: пи * радиус в квадрате
  area = PI * rad * rad;

  if ( formula )
    printf( "Формула для расчета площади круга: "\
            "пи * радиус в квадрате\n" );

  return ( area );
  // краткая запись: return ( PI * rad * rad );
}

// вычисление длины окружности
// параметры - радиус окружности
//             и признак вывода формулы на экран
float circle_len( float rad, int formula )
{
  float len;
  // длина окружности равна: 2 * пи * радиус
  len = 2 * PI * rad;

  if ( formula )
    printf( "Формула для расчета длины окружности: " \
            "2 * пи * радиус\n" );

  return ( len );
}
```

Программа **square2.c**

```
// ------------------------------------------------------------
// Модуль: функции для работы с квадратом
// Версия 2
// ------------------------------------------------------------

#include <stdio.h>

// вычисление площади квадрата
// параметры - сторона квадрата
//             и признак вывода формулы на экран
float square_area( float side, int formula )
{
  float area;
  // площадь равна: длина стороны в квадрате
  area = side * side;

  if ( formula )
```

```
    printf( "Формула для расчета площади квадрата: "\
            "длина стороны в квадрате\n" );

  return ( area );
}

// вычисление периметра квадрата
// параметры - сторона квадрата
//             и признак вывода формулы на экран
float square_perim( float side, int formula )
{
  float len;
  // периметр равен: 4 * длина стороны
  len = 4 * side;

  if ( formula )
    printf( "Формула для расчета периметра квадрата: " \
            "длина стороны * 4\n" );

  return ( len );
}
```

Файл **geometry2.h**

```
// ------------------------------------------------------------
// Прототипы функций из библиотеки
// Версия 2
// ------------------------------------------------------------

float circle_area( float rad, int formula );
float circle_len( float rad, int formula );

float square_area( float side, int formula );
float square_perim( float side, int formula );
```

Заголовочный файл **geometry2.h** скопируйте в каталог **/usr/local/include** (или **/usr/local/include/geometry**, если вы его создавали). Не забывайте, что значение параметра компилятора –I и имя заголовочного файла в директиве #include должны образовывать полный путь к этому файлу, как было показано выше.

Командный файл **shared_lib2.sh** для создания библиотеки будет таким:

Файл **shared_lib2.sh**

```
#!/bin/sh
# ------------------------------------------------------------
# Создание разделяемой библиотеки
# Версия 2
# ------------------------------------------------------------
```

```
# Компилируем исходные модули для библиотеки
gcc -fPIC -c circle2.c
gcc -fPIC -c square2.c

# Включаем скомпилированные модули в библиотеку libgeometry.so
gcc -shared -Wl,-soname=libgeometry.so.2 \
-o libgeometry.so.2.0.1 circle2.o square2.o
```

Создав файл библиотеки с именем **libgeometry.so.2.0.1**, скопируйте его в каталог **/usr/lib** и создайте в этом каталоге следующие символические ссылки:

```
cp libgeometry.so.2.0.1 /usr/lib
cd /usr/lib
ln -sf libgeometry.so.2.0.1 libgeometry.so.2
ln -sf libgeometry.so.2 libgeometry.so
```

Поскольку мы внесли изменения в библиотечные функции, необходимо внести изменения и в программу, предназначенную для тестирования библиотеки.

Программа **demo_lib2.c**

```
// ------------------------------------------------------------
// Программа, использующая функции из библиотеки
// Версия 2
// ------------------------------------------------------------

#include <stdio.h>

#include "geometry2.h"

#define  PRINT_FORMULA  1

int main( void )
{
  float rad;
  float side;

  printf( "Hello, World!\n\n" );

  printf( "Введите радиус окружности: " );
  scanf( "%f", &rad );
  printf( "\n%f\n", rad );
  printf( "Длина окружности: %f\n",
          circle_len( rad, PRINT_FORMULA ) );
  printf( "Площадь круга: %f\n",
          circle_area( rad, PRINT_FORMULA ) );

  printf( "\nВведите длину стороны квадрата: " );
  scanf( "%f", &side );
```

```
  printf( "\n%f\n", side );
  printf( "Периметр квадрата: %f\n",
          square_perim( side, PRINT_FORMULA ) );
  printf( "Площадь квадрата: %f\n",
          square_area( side, PRINT_FORMULA ) );

  return 0;
}
```

Скомпилируйте эту программу:

```
gcc -o demo_shared_lib2 demo_lib2.c -I/usr/local/include
-lgeometry
```

Обратите внимание, что в этой команде имя библиотеки не изменилось. Однако за счет применения символических ссылок компиляция выполняется с использованием новой версии библиотеки **libgeometry.so.2.0.1**. Это можно проверить таким образом:

```
ldd demo_shared_lib demo_shared_lib2

demo_shared_lib:
     libgeometry.so.1 => /usr/lib/libgeometry.so.1 (0x2807a000)
     libc.so.6 => /lib/libc.so.6 (0x2807c000)
demo_shared_lib2:
     libgeometry.so.2 => /usr/lib/libgeometry.so.2 (0x2807a000)
     libc.so.6 => /lib/libc.so.6 (0x2807c000)
```

Чтобы выполнить новую версию программы, перейдите в свой рабочий каталог и введите команду

```
./demo_shared_lib2
```

Теперь мы покажем, что можно изменить ряд функций, входящих в библиотеку, но, сохраняя совместимость с предыдущей версией этих функций, обойтись без перекомпиляции программы, которая обращается к данной разделяемой библиотеке.

Поскольку во все четыре библиотечные функции мы внесем лишь небольшие изменения косметического характера, то полные тексты модулей **circle2_1.c** и **square2_1.c** приводить не будем. Представление о сути изменений можно получить на основе следующего фрагмента модуля **circle2_1.c**.

Фрагмент программы **circle2_1.c**

```
  . . . . . .
  if ( formula )
    printf( "Формула для расчета площади круга: \n" \
```

```
            "--- пи * радиус в квадрате ---\n" );
  . . . . . .
```

Соответственно откорректируем командный файл **shared_lib2_1.sh**.

Файл **shared_lib2_1.sh**

```
#!/bin/sh
# ------------------------------------------------------------
# Создание разделяемой библиотеки
# Версия 2.1
# ------------------------------------------------------------

# Компилируем исходные модули для библиотеки
gcc -fPIC -c circle2_1.c
gcc -fPIC -c square2_1.c

# Включаем скомпилированные модули в библиотеку libgeometry.so
gcc -shared -Wl,-soname=libgeometry.so.2 \
-o libgeometry.so.2.1.1 circle2_1.o square2_1.o
```

Обратите внимание, что так называемое имя soname не изменилось, хотя имя файла библиотеки стало другим: младший номер версии теперь равен 1.

Как и прежде, скопируйте созданную библиотеку в каталог /usr/lib.

Теперь ссылка **libgeometry.so.2** должна указывать на новую версию библиотеки, т. е. на файл **libgeometry.so.2.1.1**:

```
ln -sf libgeometry.so.2.1.1 libgeometry.so.2
```

После всех манипуляций с символическими ссылками у вас должна получиться следующая конфигурация:

```
lrwxr-xr-x  1 root  wheel    16 12 дек 19:55 libgeometry.so ->
libgeometry.so.2
lrwxr-xr-x  1 root  wheel    20 12 дек 19:23 libgeometry.so.1
-> libgeometry.so.1.0.1
-rw-rw-rw-  1 root  wheel  4976 12 дек 18:46 libgeome-
try.so.1.0.1
lrwxr-xr-x  1 root  wheel    20 12 дек 20:21 libgeometry.so.2
-> libgeometry.so.2.1.1
-rw-rw-rw-  1 root  wheel  5401 12 дек 20:05 libgeome-
try.so.2.0.1
-rw-rw-rw-  1 root  wheel  5449 12 дек 20:19 libgeome-
try.so.2.1.1
```

Теперь запустите программу **demo_shared_lib2**, не выполняя перекомпиляцию. Вы видите, что вступили в действие внесенные нами в биб-

лиотечные функции незначительные изменения, касающиеся формата вывода формул.

ВНИМАНИЕ. При наличии как статической, так разделяемой библиотеки с одним и тем же именем, компоновщик будет использовать разделяемую библиотеку. Поэтому иногда может быть целесообразно присвоить этим вариантам библиотеки разные имена.

В операционной системе Windows

В операционной системе Windows концепция разделяемых библиотек реализована в виде так называемых библиотек динамической компоновки – Dynamic Link Libraries (DLL).

При использовании среды MSYS общая стратегия создания разделяемой библиотеки остается такой же, как и в среде ОС FreeBSD.

Поскольку нашей целью является дать студенту лишь общее представление о создании и использовании библиотек в среде Windows, то ограничимся краткими примерами. Более подробно технология создания и использования библиотек описана в литературе по операционной системе Windows.

Командный файл для создания библиотеки будет очень похож на тот, который вы уже использовали в среде FreeBSD.

Файл **shared_lib_win.sh**

```
#!/bin/sh
# ------------------------------------------------------------
# Создание разделяемой библиотеки
# ------------------------------------------------------------

# Компилируем исходные модули для библиотеки
gcc -c circle.c
gcc -c square.c

# Включаем скомпилированные модули
# в библиотеку libgeometry.dll
gcc -shared -o libgeometry.dll circle.o square.o
```

Полученный файл **libgeometry.dll** переместите в каталог **/usr/local/lib**:

```
mv libgeometry.dll /usr/local/lib
```

Команда для компиляции исполняемого файла будет такой:

```
gcc -o demo_dll_lib demo_lib.c -I/usr/local/include
-L/usr/local/lib –lgeometry
```

Запустить программу можно таким образом:

```
./ demo_dll_lib.exe
```

ПРИМЕЧАНИЕ. Вспомните о том, что в системе MSYS есть два варианта терминала: **rxvt.exe** и **sh.exe**, которые описаны в главе 1.

4.1.3. Динамическая загрузка библиотек

Представим себе программу, предназначенную для обработки файлов нескольких типов, причем, заранее не известно, файл какого типа откроет пользователь. Конечно, можно в данном случае использовать статическую или разделяемую библиотеки, содержащие программный код для работы с файлами всех требуемых типов. Но при этом нужно учитывать, что в оперативную память будет загружаться сразу *весь* набор библиотечных функций, а не только те функции, которые нужны для обработки файла конкретного типа, открытого пользователем в настоящий момент. Конечно, загрузка ненужного программного кода замедлит процесс запуска программы.

В подобных случаях может помочь динамическая загрузка библиотек. При этом подходе библиотека загружается в память только при необходимости использования программного кода, содержащегося в ней.

Для реализации описанной технологии используются следующие стандартные функции: dlopen(), dlclose(), dlsym(), dlerror(). Подробнее познакомиться с ними можно с помощью электронного руководства **man**, например:

```
man dlopen
```

В качестве простого примера рассмотрим динамическую загрузку разделяемой библиотеки **libgeometry.so**, которая была создана нами ранее.

Программа **demo_dl_lib2.c**

```
// ------------------------------------------------------------
// Программа, иллюстрирующая использование
// динамической загрузки разделяемой библиотеки
//
// За основу принята версия 2 программы, демонстрирующей
// использование разделяемых библиотек
// ------------------------------------------------------------

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

```
// включать этот заголовочный файл теперь уже не нужно,
// т. к. при компиляции программы функции из библиотеки
// еще недоступны
// #include "geometry2.h"

#define  PRINT_FORMULA  1

// определим указатель на функцию, принимающую два параметра
// и возвращающую значение типа float (таковы все функции
// из нашей библиотеки)
typedef float ( *library_func )( float, int );

// прототип функции get_func()
library_func get_func( void *lib_handle,
                       const char *func_name );

int main( void )
{
  float rad;
  float side;

  void *lib;              // дескриптор загружаемой библиотеки
  library_func lib_fn;    // указатель на функцию из библиотеки
  const char *error;      // сообщение об ошибке при обращении
                          // к библиотеке

  // открываем разделяемую библиотеку
  // ПРИМЕЧАНИЕ. Можно в качестве имени библиотеки указать
  //             не только libgeometry.so,
  //             но и libgeometry.so.2,
  //             и libgeometry.so.2.0.1,
  //             и libgeometry.so.2.1.1. Можно указать даже
  //             libgeometry.so.1, и это будет работать,
  //             но не совсем корректно, т. к. в библиотеке
  //             libgeometry.so.1 все функции имеют только
  //             по одному параметру).
  //             Рекомендуем поэкспериментировать с именами
  //             библиотеки.
  //             Такая гибкость возможна в том числе потому,
  //             что часть упоминаемых имен библиотеки
  //             являются символическими ссылками.
  lib = dlopen( "libgeometry.so", RTLD_LAZY );
  if ( !lib )
  {
    fprintf( stderr,
            "Не могу открыть библиотеку libgeometry.so: %s\n",
             dlerror() );
    exit( 1 );
  }

  // DEBUG
  /* dlsym( lib, "nonexistent_function" );
     dlsym( lib, "circle_len" );
```

```
    if ( error = dlerror() )
      fprintf( stderr, "Ошибка: %s\n", error );

    fprintf( stderr, "Ошибка: %s\n", dlerror() );
  */

  printf( "Динамическая загрузка разделяемой " \
          "библиотеки\n\n" );

  printf( "Введите радиус окружности: " );
  scanf( "%f", &rad );
  printf( "\n%f\n", rad );

  // если удалось получить указатель на функцию circle_len,
  // то выполним расчет
  // ПРИМЕЧАНИЕ. Вызываем библиотечную функцию, используя
  //             указатель на нее.
  if ( ( lib_fn = get_func( lib, "circle_len" ) ) != NULL )
    printf( "Длина окружности: %f\n",
            ( *lib_fn )( rad, PRINT_FORMULA ) );

  // получаем доступ к остальным функциям из библиотеки
  // аналогично
  if ( ( lib_fn = get_func( lib, "circle_area" ) ) != NULL )
    printf( "Площадь круга: %f\n",
            ( *lib_fn )( rad, PRINT_FORMULA ) );

  printf( "\nВведите длину стороны квадрата: " );
  scanf( "%f", &side );
  printf( "\n%f\n", side );

  if ( ( lib_fn = get_func( lib, "square_perim" ) ) != NULL )
    printf( "Периметр квадрата: %f\n",
            ( *lib_fn )( side, PRINT_FORMULA ) );

  if ( ( lib_fn = get_func( lib, "square_area" ) ) != NULL )
    printf( "Площадь квадрата: %f\n",
            ( *lib_fn )( side, PRINT_FORMULA ) );

  // закрываем библиотеку
  dlclose( lib );

  return 0;
}

library_func get_func( void *lib_handle,
                       const char *func_name )
{
  // lib_handle - дескриптор загружаемой библиотеки
  // func_name - имя функции из библиотеки

  library_func lib_func; // указатель на функцию из библиотеки
  const char *error;
```

```
  // вызываем эту функцию, чтобы на всякий случай
  // гарантированно "сбросить" флаг наличия ошибки
  // ПРИМЕЧАНИЕ. См. закомментированный фрагмент
  //             с пометкой DEBUG в функции main(). Можно
  //             раскомментировать его, откомпилировать
  //             программу и посмотреть, что получится.
  dlerror();

  lib_func = dlsym( lib_handle, func_name );
  if ( ( error = dlerror() ) )
  {
    fprintf( stderr, "Не могу найти %s: %s\n",
             func_name, error );
    // можно при необходимости и вообще завершить программу
    // exit( 1 );
    return NULL;
  }

  return lib_func;
}
```

Поскольку на этапе компиляции обращений к функциям библиотеки **libgeometry.so** не выполняется, то команда компиляции выглядит совсем просто:

```
gcc -o demo_dl_lib2 demo_dl_lib2.c
```

Можно убедиться, что полученный исполняемый файл не зависит от библиотеки **libgeometry.so**:

```
ldd demo_dl_lib2
```

На экран будет выведено следующее:

```
demo_dl_lib2:
     libc.so.6 => /lib/libc.so.6 (0x2807a000)
```

Таким образом, все ошибки во взаимодействии с библиотекой (например, попытка найти в библиотеке несуществующую функцию) выявляются только в процессе работы программы. Конечно, в этом есть определенный риск, поэтому такие программы необходимо тщательно тестировать.

В операционной системе Windows

В операционной системе Windows динамическая загрузка разделяемой библиотеки выполняется аналогично. Функции dlopen() соответствует

функция LoadLibrary(), функции dlclose() – функция FreeLibrary(), функции dlsym() – функция GetProcAddress().

Текст программы, модифицированной в учетом этих изменений, представлен ниже.

Программа **demo_dl_lib2_win.c**

```
// ------------------------------------------------------------
// Программа, иллюстрирующая использование динамической
// загрузки разделяемой библиотеки в среде ОС Windows
// ------------------------------------------------------------

#include <stdio.h>
#include <windows.h>

// включать этот заголовочный файл теперь уже не нужно,
// т. к. при компиляции программы функции из библиотеки
// еще недоступны
// #include "geometry2.h"

#define  PRINT_FORMULA  1

// определим указатель на функцию, принимающую два параметра
// и возвращающую значение типа float (таковы все функции
// из нашей библиотеки)
typedef float ( *library_func )( float, int );

// прототип функции get_func()
library_func get_func( void *dll_lib_handle,
                       const char *func_name );

int main( void )
{
  float rad;
  float side;

  // дескриптор загружаемой библиотеки DLL
  HINSTANCE dll_lib;

  // указатель на функцию из библиотеки DLL
  library_func lib_fn;


  // открываем библиотеку DLL
  dll_lib = LoadLibrary( "libgeometry2.dll" );

  if ( dll_lib == NULL )
  {
    fprintf( stderr, "Не могу открыть библиотеку " \
                     "libgeometry2.dll\n" );
    exit( 1 );
  }
```

```
  printf( "Динамическая загрузка библиотеки DLL\n\n" );

  printf( "Введите радиус окружности: " );
  scanf( "%f", &rad );
  printf( "\n%f\n", rad );

  // если удалось получить указатель на функцию circle_len,
  // то выполним расчет
  // ПРИМЕЧАНИЕ. Вызываем библиотечную функцию, используя
  //             указатель на нее.
  if ( ( lib_fn = get_func( dll_lib, "circle_len" ) ) !=
         NULL )
    printf( "Длина окружности: %f\n",
            ( *lib_fn )( rad, PRINT_FORMULA ) );

  // получаем доступ к остальным функциям из библиотеки
  // аналогично
  if ( ( lib_fn = get_func( dll_lib, "circle_area" ) ) !=
         NULL )
    printf( "Площадь круга: %f\n",
            ( *lib_fn )( rad, PRINT_FORMULA ) );

  printf( "\nВведите длину стороны квадрата: " );
  scanf( "%f", &side );
  printf( "\n%f\n", side );

  if ( ( lib_fn = get_func( dll_lib, "square_perim" ) ) !=
         NULL )
    printf( "Периметр квадрата: %f\n",
            ( *lib_fn )( side, PRINT_FORMULA ) );

  if ( ( lib_fn = get_func( dll_lib, "square_area" ) ) !=
         NULL )
    printf( "Площадь квадрата: %f\n",
            ( *lib_fn )( side, PRINT_FORMULA ) );

  // закрываем библиотеку DLL
  FreeLibrary( dll_lib );

  return 0;
}

library_func get_func( void *lib_handle,
                       const char *func_name )
{
  // lib_handle - дескриптор загружаемой библиотеки DLL
  // func_name - имя функции из библиотеки DLL

  // указатель на функцию из библиотеки DLL
  library_func lib_func;

  // получим адрес библиотечной функции
```

```
  // ПРИМЕЧАНИЕ. Здесь требуется явное приведение типа.
  lib_func = ( library_func )GetProcAddress( lib_handle,
                                             func_name );

  if ( ( lib_func == NULL ) )
  {
    fprintf( stderr, "Не могу найти %s\n", func_name );
    // можно при необходимости и вообще завершить программу
    // exit( 1 );
    return NULL;
  }

  return lib_func;
}
```

Команда компиляции данного программного файла выглядит так:

```
gcc -o demo_dl_lib2 demo_dl_lib2_win.c
```

4.2. Язык Perl

Функцию библиотек в языке Perl выполняют так называемые **модули** (module). В качестве иллюстрации мы приведем текст простого модуля, который построен на основе рекомендаций, изложенных в разделе perlmod документации, поставляемой вместе с Perl. Мы включим в модуль тот же набор функций, который был включен в библиотеку, созданную нами на языке C в предыдущем разделе учебного пособия.

Поскольку схема модуля, позаимствованная из документации, сформирована в расчете на общий случай, то она является несколько избыточной. Поэтому при разработке ваших собственных модулей вы можете принимать эту схему за основу и «отсекать» от нее все лишнее, т. е. то, что не требуется или не используется в конкретном случае.

При создании и использовании модулей важным является понятие **экспортирования** и **импортирования** символов, т. е. процедур и переменных. Экспортирование означает, что, например, процедура, содержащаяся в файле модуля, станет доступной в вызывающей программе при использовании в ней специальной директивы **use**.

В приведенном ниже тексте модуля много комментариев. Рекомендуем их внимательно изучить.

Программа **Geometry.pm**

```
# ------------------------------------------------------------
# Модуль Perl: функции для проведения геометрических
# вычислений
# ------------------------------------------------------------
```

```
package Math::Geometry;  # каталог Math, файл Geometry.pm

use strict;
use warnings;

BEGIN
{
  use Exporter   ();
  our ($VERSION, @ISA, @EXPORT, @EXPORT_OK, %EXPORT_TAGS);

  # установим номер версии
  $VERSION     = 1.00;
  # эта строка может быть полезна при использовании
  # систем управления версиями (RCS/CVS)
  $VERSION = sprintf "%d.%03d", q$Revision: 1.1 $ =~ /(\d+)/g;

  @ISA         = qw( Exporter );

  # экспортируемые функции модуля (пакета),
  # которые экспортируются по умолчанию
  # ПРИМЕЧАНИЕ. Обратите внимание, что мы включили в этот
  #             массив только две наших функции: circle_area и
  #             circle_len. Это сделано в учебно-методических
  #             целях.
  @EXPORT      = qw( &func1 &func2 &func4
                     &circle_area &circle_len );
  %EXPORT_TAGS = ();   # например: TAG => [ qw!name1 name2! ]

  # экспортируемые глобальные переменные модуля (пакета)
  # и функции, которые НЕ экспортируются по умолчанию
  # ПРИМЕЧАНИЕ. Обратите внимание, что мы включили в этот
  #             массив только одну нашу функцию square_perim.
  #             Это сделано в учебно-методических целях.
  @EXPORT_OK   = qw( $Var1 %Hashit &func3
                     &square_perim );
}
our @EXPORT_OK;

# экспортируемые глобальные переменные данного модуля (пакета)
our $Var1;
our %Hashit;

# неэкспортируемые глобальные переменные данного модуля
# (пакета)
our @more;
our $stuff;

# сначала инициализируем экспортируемые глобальные переменные
# модуля (пакета)
$Var1   = '3';
%Hashit = ();

# затем инициализируем остальные глобальные переменные модуля
```

```
# (пакета)
# ПРИМЕЧАНИЕ. Эти переменные доступны извне пакета в форме,
#             например, $Math::Geometry::stuff, а не просто
#             $stuff.
$stuff  = '4';
@more   = ();

# все лексические переменные, область видимости которых
# ограничена данным файлом, должны быть созданы прежде, чем
# созданы функции, использующие эти переменные

# это лексические переменные, видимые в пределах этого файла
my $priv_var    = '2';
my %secret_hash = ();

# здесь размещаются функции, видимые в пределах этого файла
# вызываются такие функции так: &$priv_func;
# такие функции не могут иметь прототипа
my $priv_func = sub {
    # текст функции
};

# здесь располагаются все остальные функции, как
# экспортируемые, так и неэкспортируемые
# ПРИМЕЧАНИЕ. Не забудьте включить какой-нибудь
#             осмысленный текст внутри скобок {}.
sub func1      {}    # функция без прототипа
sub func2()    {}    # прототип указывает на отсутствие
                     # параметров
sub func3($$)  {}    # прототип указывает на наличие двух
                     # скалярных параметров
sub func4(\%)  {}    # прототип указывает на наличие
                     # параметра -- ссылки на хеш-массив

# код, выполняющийся при завершении работы модуля
# (глобальный деструктор)
END { }

# Здесь располагается ваш программный код
# ========================================

# ------------------------------------------------------------
# функции для работы с кругом и окружностью
# ------------------------------------------------------------

use constant   PI => 3.14159265;

# --------------------------
# вычисление площади круга
# --------------------------
sub circle_area( $$ )
{
  # параметры - радиус окружности и признак вывода формулы
```

```
  # на экран
  my $rad = shift;
  my $formula = shift;

  my $area;

  # площадь равна: пи * радиус в квадрате
  $area = PI * $rad * $rad;

  if ( $formula )
  {
    print "Формула для расчета площади круга: \n" .
          "--- пи * радиус в квадрате ---\n";
  }

  # краткая запись: return ( PI * $rad * $rad );
  return ( $area );
}

# ------------------------------
# вычисление длины окружности
# ------------------------------
sub circle_len( $$ )
{
  # параметры - радиус окружности и признак вывода формулы
  # на экран
  my $rad = shift;
  my $formula = shift;

  my $len;

  # длина окружности равна: 2 * пи * радиус
  $len = 2 * PI * $rad;

  if ( $formula )
  {
    print "Формула для расчета длины окружности: \n" .
          "--- 2 * пи * радиус ---\n";
  }

  return ( $len );
}


# ------------------------------------------------------------
# функции для работы с квадратом
# ------------------------------------------------------------

# ----------------------------
# вычисление площади квадрата
# ----------------------------
sub square_area( $$ )
{
```

```
  # параметры - сторона квадрата и признак вывода формулы
  # на экран
  my $side = shift;
  my $formula = shift;

  my $area;

  # площадь равна: длина стороны в квадрате
  $area = $side * $side;

  if ( $formula )
  {
    print "Формула для расчета площади квадрата: \n" .
          "--- длина стороны в квадрате ---\n";
  }

  return ( $area );
}

#---------------------------------
# вычисление периметра квадрата
# ---------------------------------
sub square_perim( $$ )
{
  # параметры - сторона квадрата и признак вывода формулы
  # на экран
  my $side = shift;
  my $formula = shift;

  my $len;

  # периметр равен: 4 * длина стороны
  $len = 4 * $side;

  if ( $formula )
  {
    print "Формула для расчета периметра квадрата: \n" .
          "--- длина стороны * 4 ---\n";
  }

  return ( $len );
}


1;  # это возвращаемое значение "истина"
    # (должно присутствовать обязательно)
```

Теперь мы напишем программу, которая будет использовать процедуры из разработанного модуля. Эта программа аналогична той, что была представлена в предыдущем разделе, посвященном разработке библиотек на языке C.

В предлагаемой вашему вниманию программе также много комментариев, имеющих учебную направленность. Они будет полезны для более глубокого понимания особенностей использования модулей на языке Perl.

Программа **demo_module.pl**

```
#!/usr/bin/perl -w
# ------------------------------------------------------------
# Программа, использующая функции из модуля Perl
# ------------------------------------------------------------

# такая директива позволяет импортировать только те функции,
# которые включены в массив @EXPORT модуля Math::Geometry
# (в данном случае нас интересуют функции circle_len и
# circle_area)
use Math::Geometry;

# такая директива позволяет импортировать те функции,
# которые включены в массив @EXPORT_OK модуля Math::Geometry
# ПРИМЕЧАНИЕ. Для включения в список двух и более функций
#             просто разместите их внутри скобок, разделяя
#             пробелами.
use Math::Geometry qw( square_perim );

# аналог директивы препроцессора #define
use constant  PRINT_FORMULA => 1;

use strict;

my $rad;
my $side;

print "Тестирование модуля (пакета) Math::Geometry\n\n";

print "Введите радиус окружности: ";
$rad = <STDIN>;
printf "\n%f\n", $rad;
printf "Длина окружности: %f\n",
       circle_len( $rad, PRINT_FORMULA );
printf "Площадь круга: %f\n",
       circle_area( $rad, PRINT_FORMULA );

print "\nВведите длину стороны квадрата: ";
$side = <STDIN>;
printf "\n%f\n", $side;
printf "Периметр квадрата: %f\n",
       square_perim( $side, PRINT_FORMULA );

# поскольку функция square_area не включена ни в массив
# @EXPORT, ни в массив @EXPORT_OK модуля (пакета)
# Math::Geometry, то приходится вызывать ее с полным именем,
# включающим имя модуля (пакета)
```

```
printf "Площадь квадрата: %f\n",
       Math::Geometry::square_area( $side, PRINT_FORMULA );

print "\nПеременные пакета Math::Geometry\n";
# ПРИМЕЧАНИЕ. Обратите внимание, что значение переменной
#             priv_var пустое, и при этом выводится
#             предупредительное сообщение.
print "priv_var = " . $Math::Geometry::priv_var . "\n";
print "Var1 = " . $Math::Geometry::Var1 . "\n";
print "stuff = " . $Math::Geometry::stuff . "\n";

exit 0;
```

Запись Math::Geometry в директиве use означает, что модуль **Geometry.pm** должен находиться в подкаталоге **Math**.

Как и библиотеки на языке C, модули Perl нужно разместить в одном из каталогов, в которых Perl сможет их найти при необходимости. Чтобы определить перечень этих каталогов, воспользуйтесь следующей небольшой программой.

Программа **INC.pl**

```
#!/usr/bin/perl -w

# @INC – это встроенный массив
foreach ( @INC )
{
  print "$_\n";
}

exit 0;
```

Эта программа выведет примерно следующее (в последней строке выводится символ «.», что означает текущий каталог):

```
/usr/local/lib/perl5/5.8.8/BSDPAN
/usr/local/lib/perl5/site_perl/5.8.8/mach
/usr/local/lib/perl5/site_perl/5.8.8
/usr/local/lib/perl5/site_perl
/usr/local/lib/perl5/5.8.8/mach
/usr/local/lib/perl5/5.8.8
.
```

Создайте подкаталог **Math** в каталоге, например, **/usr/local/lib/perl5/site_perl**, и скопируйте в этот подкаталог файл **Geometry.pl**. Назначьте такие привилегии доступа к модулю **Geometry.pm** и программе **demo_module.pl**, чтобы их можно было выполнять, т. е. **chmod 755**.

ПРИМЕЧАНИЕ. Не забывайте, что в операционной системе FreeBSD регистр символов имеет определяющее значение. Поэтому при создании файлов и каталогов сохраняйте то написание имен, которое используется в учебном пособии.

Теперь можно запустить программу:

```
./demo_module.pl
```

Для получения подробных инструкций по процедуре создания модулей следует обратиться к электронной документации, входящей в комплект Perl:

```
man perlmod
```

В операционной системе Windows

В среде Windows необходимо разместить пакет либо в каталоге **C:\Perl\lib**, либо в каталоге **C:\Perl\site\lib**, либо в текущем каталоге. Для получения списка этих каталогов также можно воспользоваться программой **INC.pl**.

Создайте в каталоге **C:\Perl\site\lib** подкаталог **Math** и скопируйте в него модуль **Geometry.pm**.

В программе **demo_module.pl** замените первую строку, содержащую путь к интерпретатору Perl, на строку

```
#!perl -w
```

Для запуска программы введите

```
perl demo_module.pl
```

Контрольные вопросы и задания

1. С помощью электронного руководства **man** детально изучите работу программы **nm**.

2. Используя команду **file**, исследуйте статическую и разделяемую библиотеки.

3. В программе **demo_dl_lib2.c** раскомментируйте фрагмент кода с пометкой DEBUG. Откомпилируйте программу и внимательно изучите все выводимые ею сообщения.

4. В программе **demo_dl_lib2.c** передайте функции dlopen() в качестве параметра неверное имя библиотеки, например, **libgeometryy.so**. Откомпилируйте и запустите программу. Вы увидите, что при компиляции никаких ошибок не возникает. Ошибка возникает на стадии исполнения программы.

5. Поэкспериментируйте с программой **demo_dl_lib2.c**. Например, последуйте рекомендациям, приведенным в связи с использованием функции dlopen(), и внимательно изучите сообщения, выводимые программой на экран.

6. Сформируйте вторую версию библиотеки **libgeometry.dll** на основе исходных файлов **circle2.c** и **square2.c**. Рекомендуем присвоить этой библиотеке имя **libgeometry2.dll**, чтобы различать вторую и первую версии.

7. Скомпилируйте программу **demo_lib2.c** со второй версией библиотеки **libgeometry.dll**, т. е. **libgeometry2.dll**.

8. Сформируйте вторую версию библиотеки **libgeometry2.dll** на основе исходных файлов **circle2_1.c** и **square2_1.c**. Замените файл **libgeometry2.dll**, который находится в каталоге **/usr/local/lib** системы MSYS, новым вариантом второй версии библиотеки. Попробуйте запустить программу **demo_lib2.exe** без перекомпиляции. Изменился ли формат вывода формул, по которым выполнялись расчеты?

9. Попробуйте включить функцию square_area в директиву use… в программе **demo_module.pl**:

```
use Math::Geometry qw( square_perim square_area );
```

Какая ошибка возникает? Как вы думаете, почему так происходит?

10. Самостоятельно добавьте в модуль **Geometry.pm** ряд других процедур, например, для работы с такими трехмерными геометрическими фигурами, как шар, куб, параллелепипед и др.

11. Сравните технологию применения разделяемых библиотек и технологию применения модулей языка Perl. В чем их сходство и в чем различие?

5. Утилита make

При разработке сложных программ, состоящих из большого числа взаимосвязанных модулей, возникает проблема управления такой системой исходных, объектных и исполняемых файлов. Инструментом, который традиционно применяется в этих случаях, является утилита **make**.

Утилита **make** управляет процессом компиляции с помощью так называемого make-файла. Такой файл имеет довольно жесткую структуру. Основной структурной единицей этого файла является **правило** (rule). Правило указывает, когда и каким образом выполняется переформирование того или иного файла либо выполняется какое-то действие. Под переформированием понимается, например, компиляция, компоновка и т. д. Правило выглядит таким образом:

```
Целевое имя (target): условия (prerequisites)
     команда
```

Команда пишется с обязательным отступом, который формируется **только** с помощью символа табуляции (клавиша Tab).

Правило может быть, например, таким:

```
foo.o : foo.c defs.h
     cc -c -g foo.c
```

Оно расшифровывается следующим образом:

– целевое имя (цель, имя правила) – **foo.o**;

– файлы, от которых зависит целевой файл – **foo.c** и **defs.h**. Если один (или оба) из этих файлов на момент выполнения данного правила оказывается более новым, чем объектный файл **foo.o**, то выполняется команда, приведенная в данном правиле. Команда выполняется также и если файла **foo.o** не существует;

– команда компиляции исходного файла с целью получения объектного модуля, указанного в качестве целевого имени: **cc –c –g foo.c**. При этом параметр **–c** указывает на то, что выполняется только компиляция без создания исполняемого файла, а параметр **–g** указывает на то, что в объектный модуль должна быть включена информация для отладки программы с помощью отладчика.

ПРИМЕЧАНИЕ. В команде не упоминается заголовочный файл **defs.h**, поскольку предполагается, что он включен в исходный файл foo.c с помощью соответствующей директивы препроцессора. Но в список файлов, от которых зависит целевой файл, **defs.h** необходимо включать.

Порядок следования правил в make-файлах не является принципиально важным, за одним исключением: правило, выполняемое по умолча-

нию, должно быть указано первым из всех правил в make-файле. Обычно первое правило предписывает порядок создания основной программы. В качестве целевого имени такого правила чаще всего используется **all**.

Когда вы запускаете утилиту **make**, то по умолчанию она ищет в текущем каталоге файл с именем **Makefile** (обратите внимание на заглавную букву M). Можно давать make-файлу и другое имя, но тогда придется вызывать эту утилиту с параметром **–f имя_make_файла**.

Если утилита **make** вызывается без параметров, то она по умолчанию выполняет первое правило в найденном make-файле. В том, случае, когда для создания исполняемого файла необходимо сначала создать объектные модули, утилита **make** выполняет правила, определяющие порядок создания этих модулей.

В make-файле могут быть правила, согласно которым выполняются действия, не связанные непосредственно с созданием объектных или исполняемых файлов. Одним из примеров подобных действий может быть удаление всех объектных модулей из текущего каталога, т. е. своего рода очистка рабочего каталога. Такое действие можно легко выполнить вручную при небольшом числе исходных файлов, но в больших проектах со сложной структурой каталогов необходимо прибегать к средствам автоматизации рутинных операций. Одним из традиционных целевых имен описанного типа является имя **clean**.

Очень удобным средством повышения наглядности команд в make-файлах являются переменные. Примеры их применения показаны в make-файле, приведенном ниже.

Поскольку у нас уже есть набор исходных текстов программ, которые мы использовали для иллюстрации процедур создания и применения библиотек, то мы можем воспользоваться этими же программами и для иллюстрации работы с make-файлами. Мы покажем, каким образом можно создать исполняемый файл из трех исходных модулей: **demo_lib2.c**, **circle2_1.c**, **square2_1.c**. В приведенном make-файле содержится много комментариев, которые призваны пояснить некоторые важные и полезные приемы.

ПРИМЕЧАНИЕ. При использовании данного make-файла предполагается, что заголовочный файл **geometry2.h** находится в текущем каталоге.

Файл **Makefile**

```
# ------------------------------------------------------------
# Makefile для компиляции программы demo_lib2.c
# ------------------------------------------------------------

# имя исполняемого файла
PRG = demo_lib2

# имена объектных модулей, которые компонуются
# в единый исполняемый файл
```

```
OBJS = demo_lib2.o circle2_1.o square2_1.o

# имя компилятора (может быть, например, g++)
CC = gcc

# параметры компилятора
# параметр -g означает, что в объектные модули будет
# включена информация, необходимая для отладки программы
# с помощью отладчика, например, gdb
CPARAMS = -g

# такая переменная может быть удобна, если необходимо
# указать компилятору, в каких каталогах производить
# поиск заголовочных файлов
# INC = -I/usr/local/include -I/home/my_dir/include

# такая переменная может быть удобна, если необходимо
# указать компоновщику, в каких каталогах производить поиск
# библиотек, а также имена этих библиотек
# LIBS = -L/usr/local/lib -lwx_mswud_adv-2.6 -lmy_lib

# это правило -- all -- выполняется в том случае, когда
# команда make вызывается без параметра, указывающего
# целевое правило (target)
# ОЧЕНЬ ВАЖНО. Все команды в make-файле начинаются
#              с символа табуляции.
#              Не заменяйте символ табуляции пробелами!
all: $(OBJS)
# в следующей строке первый символ - символ табуляции
     $(CC) -o $(PRG) $(OBJS)
# это пример команды, в которой используются дополнительные
# библиотеки
#    $(CC) -o $(PRG) $(OBJS) $(LIBS)

# этот объектный модуль зависит от двух файлов:
# demo_lib2.c и geometry2.h
# переменная $@ означает имя правила (target),
# в данном случае это demo_lib2.o
demo_lib2.o: demo_lib2.c geometry2.h
     $(CC) $(CPARAMS) -c demo_lib2.c -o $@
# это пример команды, в которой используются дополнительные
# каталоги для поиска заголовочных файлов
#    $(CC) $(CPARAMS) -c demo_lib2.c $(INC) -o $@

# эти правила аналогичны предыдущему правилу, с той лишь
# разницей, что объектный модуль зависит не от двух файлов,
# а от одного
circle2_1.o: circle2_1.c
     $(CC) $(CPARAMS) -c circle2_1.c -o $@

square2_1.o: square2_1.c
     $(CC) $(CPARAMS) -c square2_1.c -o $@
```

```
# это правило позволяет сохранить результаты работы
# препроцессора
pre:
     $(CC) -E $(CPARAMS) -c demo_lib2.c -o demo_lib2.pre
     $(CC) -E $(CPARAMS) -c circle2_1.c -o circle2_1.pre
     $(CC) -E $(CPARAMS) -c square2_1.c -o square2_1.pre

# правило, имя которого является названием действия
# (очистить), а не именем файла
clean:
     rm $(OBJS) *.pre $(PRG)
```

Порядок использования приведенного make-файла очень прост. Для того чтобы создать программу **demo_lib2**, выполните команду

```
make
```

Для удаления всех файлов, кроме исходных текстов программ, выполните команду

```
make clean
```

В операционной системе Windows

В среде Windows необходимо лишь изменить имя исполняемого файла, которое присваивается переменной PRG, в стиле Windows, т. е. добавить расширение .exe. Также желательно провести перекодирование исходных текстов из кодировки KOI-R в кодировку CP866 с помощью одной из утилит, предназначенных для этой цели (например, **iconv**). Все остальные приемы, описанные в настоящей главе для операционной системы FreeBSD, должны оставаться справедливыми и для среды MSYS в операционной системе Windows.

Контрольные вопросы и задания

1. Удалите один из объектных модулей, от которых зависит создание исполняемого модуля, например, **circle2_1.o**. Запустите компиляцию командой **make** и посмотрите, какие команды выполняются (при выполнении команд утилита make выводит команды на стандартный вывод, их можно перенаправить в файл-журнал для последующего изучения).

2. Откройте файл **geometry2.h** в текстовом редакторе и, не внося изменений, сохраните его, чтобы изменилось *время* последнего редактирования файла. Затем выполните команду **make**. Какой исходный файл перекомпилируется? Почему?

3. Вместо командного файла, который использовался в предыдущей главе для создания библиотеки **libgeometry.dll**, напишите make-файл. В нем предусмотрите отдельные правила для компиляции и для установки библиотеки.

4. Ознакомьтесь с возможностями утилиты **make** более детально, используя документацию.

6. Отладка программ

В наши дни много говорится о прогрессе в области качества и надежности программного обеспечения, но, тем не менее, время от времени появляются сообщения об ошибках, обнаруженных в программном обеспечении самых известных компаний-разработчиков. Если синтаксические ошибки выявляются еще на этапе компиляции исходных текстов программ, то логические ошибки представляют собой гораздо более серьезную проблему. Как показывает практика, выявить на стадии тестирования все логические ошибки в больших и сложных программах бывает очень трудно. Но, к счастью, все же существует инструмент, который помогает в нелегкой борьбе с логическими ошибками. Этот инструмент называется отладчиком (debugger).

Мы рассмотрим только отладчики, применяемые для отладки программ, написанных на языках C/C++ и Perl.

6.1. Язык C/C++

Отладчик **gdb** используется уже много лет. Его первым автором является легендарный программист Ричард Столлмен (Richard Stallman). Это он отстаивает и продвигает идею свободного программного обеспечения в противовес коммерческому ПО.

Что может отладчик? Вот перечень основных возможностей:

– запустить программу;

– остановить процесс ее выполнения при наступлении некоторого события (условия) или при достижении определенной точки в ее исходном тексте;

– исследовать состояние памяти при остановке программы, а также внести изменения в это состояние (например, изменить значения тех или иных переменных) и продолжить выполнение программы с точки останова.

Таким образом, отладчик позволяет провести эксперименты с программой с целью лучше понять проблемную ситуацию, в которой проявляется логическая ошибка.

Для того чтобы использовать отладчик, необходимо предусмотреть определенные меры на этапе компиляции исходных текстов программ. Такой мерой является включение параметра **–g** в команду компиляции, например:

```
gcc –g –c circle2_1.c
```

Поскольку этот параметр мы уже использовали в make-файле, рассмотренном в предыдущей главе, то программа **demo_lib2** может быть запущена под управлением отладчика **gdb**.

Получить положительный эффект от применения отладчика можно, даже если вы знаете лишь самые простые его возможности и команды. Мы

проведем сеанс работы в отладчике на примере нашей программы **demo_lib2**.

Запускаем отладчик и программу с помощью команды

```
gdb ./demo_lib2
```

Программа на данном этапе еще не запущена. Перед ее запуском мы должны задать какое-то условие останова программы, иначе она будет выполняться практически так же, как и вне отладчика. Таким условием может быть точка останова (breakpoint). Команда, которая управляет точками останова, имеет такое же название – **breakpoint**, но его можно сокращать до одной буквы **b**. В качестве позиции в исходном тексте программы может служить имя функции или номер строки. В том случае, когда необходимо назначить точку останова не в текущем исходном файле, тогда в команде breakpoint следует указать имя этого программного файла.

Мы назначим точку останова на функции main() (при этом круглые скобки вводить не нужно):

```
b main
```

ПРИМЕЧАНИЕ. Эта и последующие команды отладчика вводятся в среде отладчика, а не в среде операционной системы.

Отладчик «ответит» нам примерно так:

```
Breakpoint 1 at 0x401315: file demo_lib2.c, line 17.
```

Теперь можно запустить программу с помощью команды **run**, которая также может быть сокращена до одной буквы **r**:

```
r
```

Вот что выводит отладчик:

```
Starting program: /root/PT_BOOK/LIB_EXAMPLE/demo_lib2

Breakpoint 1, main () at demo_lib2.c:17
17        printf( "Hello, World!\n\n" );
```

Выполнение программы останавливается на строке под номером 17, т. е. на той строке, которая *подлежит выполнению* (а не является *уже выполненной*). Чтобы двигаться далее в пошаговом режиме, введите команду **next**, которая может быть сокращена до одной буквы **n**, и нажмите клавишу Enter. Отладчик выведет:

```
Hello, World!
```

Обратите внимание, что отладчик выводит не только команды, но также и результаты их работы.

Для повторения предыдущей команды можно просто нажимать клавишу Enter. Нажмите ее дважды – и в работу включится функция scanf(), которая предложит вам ввести число. Нужно ввести это число так, как если бы вы работали с программой вне отладчика, и нажать клавишу Enter.

Действуя таким образом, необходимо достичь строки под номером 22:

```
22        printf( "Длина окружности: %f\n", circle_len( rad,
PRINT_FORMULA ) );
```

Но теперь следует ввести другую команду – **step** (сокращенно – просто **s**). Она позволяет войти «внутрь» функции, вызов которой содержится в данной строке исходного текста. Введите

```
s
```

На экран будет выведено следующее:

```
circle_len (rad=3, formula=1) at circle2_1.c:31
31        len = 2 * PI * rad;
```

Вы видите, что указан номер строки файла **circle2_1.c**, с которой была вызвана функция circle_len(). При этом показаны и значения параметров этой функции, переданные ей при вызове. Если вам нужно посмотреть исходный текст выше и ниже текущей строки, то введите команду **list** без параметров (сокращение – **l**) и нажмите клавишу Enter.

В нашей простой программе нет определений сложных структур данных, но иногда бывает необходимо получить справку о типе данных той или иной переменной. Для решения этой задачи служит команда **ptype**. Выясним, например, тип данных переменной len:

```
ptype len
```

Отладчик ответит:

```
type = float
```

Если бы эта переменная была объектом какого-либо класса, то все определение данного класса было бы выведено на экран.

При вызове функций информация о последовательности этих вызовов сохраняется в стеке программы. Поэтому можно увидеть всю цепочку

вызовов функций от запуска программы до текущей функции. В этом может помочь команда **backtrace** (сокращенно – **bt**):

```
bt
```

Стек вызовов функций в данном случае очень небольшой, но нам важно проиллюстрировать принципиальную возможность получения доступа к этой информации. Вот что покажет отладчик (обратите внимание на обратную нумерацию функций):

```
#0  circle_len (rad=3, formula=1) at circle2_1.c:31
#1  0x0040136b in main () at demo_lib2.c:22
```

Выведенные строки показывают, что функция circle_len() была вызвана со строки номер 22 программного файла **demo_lib2.c**, а в настоящий момент времени управление находится на строке 31 программного файла **circle2_1.c**.

Теперь разрешите отладчику выполнить текущую строку программного файла, введя команду **n** (**next**). Программа вычислит значение длины окружности, которое записывается в переменную len. Чтобы узнать значение переменной, используйте команду **print** (сокращенно – **p**):

```
p len
```

Отладчик «знает» и это:

```
$1 = 18.849556
```

В начале настоящей главы мы говорили, что отладчик позволяет изменять значения переменных. Давайте это сейчас проделаем: назначим переменной len значение, например, 15 (правда, в нарушение законов геометрии). Это делается так:

```
p len=15
```

Вы можете проверить, изменилось ли значение переменной, с помощью опять-таки команды **p**.

В том случае, когда необходимо выполнить оставшуюся часть функции (не программы, а функции) не в пошаговом режиме, а в обычном, используется команда **finish**. Введите ее и нажмите клавишу Enter:

```
finish
```

Картина на экране будет такой:

```
Run till exit from #0  circle_len (rad=3, formula=1) at cir-
cle2_1.c:33
Формула для расчета длины окружности:
--- 2 * пи * радиус ---
0x0040136b in main () at demo_lib2.c:22
22        printf( " Длина окружности: %f\n", circle_len( rad,
PRINT_FORMULA ) );
Value returned is $4 = 15
```

Программа вывела на экран текст, вывод которого предусмотрен функцией circle_len(). Выведено также и значение, возвращенное функцией circle_len() в вызывающую функцию, т. е. printf().

Теперь создадим еще одну точку останова, но уже с использованием номера строки и имени программного модуля. Пусть это будет так:

```
b square2_1.c:16
```

Отладчик выполнит команду и сообщит об этом:

```
Breakpoint 2 at 0x40149f: file square2_1.c, line 16.
```

Для продолжения работы нужно использовать команду **continue** (сокращенно – **c**):

```
c
```

Отладчик сообщит о том, что работа продолжена:

```
Continuing.
```

Когда процесс управления программой дойдет до очередной точки останова, отладчик выведет сообщение

```
Breakpoint 2, square_area (side=3, formula=1) at
square2_1.c:16
16        if ( formula )
```

У нас появилась очередная пауза, во время которой можно познакомиться с системой подсказок отладчика. Предлагаем вам поэкспериментировать с командой **help**. Первый шаг – выполнить эту команду без параметров:

```
help
```

Отладчик выведет список классов команд. Выберем для более детального ознакомления, например, класс running:

```
help running
```

Получаем более детальную информацию именно об этом классе команд. Вот ее небольшой фрагмент:

```
...
disconnect -- Disconnect from a target
finish -- Execute until selected stack frame returns
handle -- Specify how to handle a signal
...
```

Можно получить еще более детальные сведения о конкретной команде, например:

```
help until
```

Вот эти сведения:

```
Execute until the program reaches a source line greater than
the current
or a specified location (same args as break command) within
the current frame.
```

Поскольку имена некоторых команд довольно длинные, то допускаются сокращения этих имен при условии, что они однозначно определяют имя команды. Ряд сокращенных имен команд мы уже использовали во время нашего первого сеанса работы с отладчиком.

Чтобы завершить выполнение программы, введите команду **continue** (или просто **c**). При успешном завершении работы программы отладчик выведет сообщение

```
Program exited normally.
```

Для выхода из отладчика используйте команду **quit** (или просто **q**):

```
q
```

Более подробную информацию об использовании отладчика **gdb** можно почерпнуть из электронного руководства **man**. Детальнейшее описание **gdb** можно найти на сайте http://www.gnu.org.

В операционной системе Windows

В среде Windows используются те же приемы работы в отладчике **gdb**, что и в среде ОС FreeBSD. Необходимо лишь учитывать особенности

терминалов **rxvt.exe** и **sh.exe**, о которых речь шла в первой главе при описании среды MSYS.

6.2. Язык Perl

В отличие от языков C и C++, программа на языке Perl не требует какой-либо специальной подготовки для того, чтобы ее можно было запустить под управлением отладчика.

Запуск отладчика производится следующим образом:

```
perl -d ./demo_module.pl
```

Сразу отметим, что интерфейс и набор команд отладчика языка Perl и отладчика **gdb** очень похожи. Поэтому мы ограничимся лишь кратким описанием основных команд отладчика Perl.

Получить краткую подсказку по командам отладчика можно с помощью команды **h**

```
h
```

Для получения более подробной подсказки введите

```
h h
```

Поскольку вывод этой команды не умещается на одном экране, то можно воспользоваться клавишей Scroll Lock, которая позволяет организовать «прокрутку» буфера экрана вверх и вниз.

Начать работу можно путем ввода команды **n** (**next**) или **s** (**step**). Назначение этих команд такое же, что и в отладчике gdb. Как и в отладчике **gdb**, для повторения предыдущей команды достаточно просто нажать клавишу Enter.

ПРИМЕЧАНИЕ. Введя команду, не забывайте нажимать клавишу Enter.

Назначить точку останова можно с помощью команды **b**. Например, команда

```
b 33
```

назначит точку останова на строке под номером 33 текущего программного файла.

Можно назначить в качестве точки останова какую-либо процедуру, находящуюся, например, в модуле **Math::Geometry**, подключаемом с помощью директивы use. Команда будет такой (не забывайте о регистре символов):

```
b Math::Geometry::square_area
```

Получить подробную подсказку по использованию команды **b** можно с помощью следующей команды

```
h b
```

Просмотреть значение переменной позволяет команда **p**, например:

```
p $len
```

Эта же команда предоставляет возможность назначить переменной новое значение, например:

```
p $len=33
```

Продолжить выполнение программы в обычном режиме до следующей точки останова можно, введя команду **c** (**continue**):

```
c
```

Выход из отладчика при завершении работы программы осуществляется по команде **q** (**quit**):

```
q
```

Получить подробные инструкции по использованию отладчика Perl можно с помощью электронного руководства:

```
man perldebug
```

В операционной системе Windows

В среде Windows используются те же приемы работы в отладчике Perl, что и в среде ОС FreeBSD. При этом наличие среды MSYS не требуется. Конечно, можно использовать Perl и в этой среде, но тогда необходимо поместить модуль **Math::Geometry** в один из каталогов, список которых можно получить, запустив в среде MSYS программу **INC.pl**, которая приведена в главе, посвященной библиотекам.

Контрольные вопросы и задания

1. При использовании параметров **–O1**, **–O2**, **–O3** (O – заглавная латинская буква) компилятор языка C/C++ генерирует *оптимизированный*

объектный код, отличающийся от того кода, который получается без использования данных параметров. Симптомы различий, имеющихся в сгенерированном коде, проявляются при запуске программы под управлением отладчика. Для того чтобы получить представление об этом интересном и, на первый взгляд, необъяснимом феномене, рекомендуем вам проделать следующий несложный эксперимент.

Создайте копию make-файла, который вы использовали в предыдущей главе для создания программы **demo_lib2**. Пусть имя нового make-файла будет **Makefile_opt** (opt означает – optimized). В новом make-файле измените значение переменной CPARAMS:

```
CPARAMS = -g -O3
```

Параметр **–O3** указывает компилятору, что необходимо генерировать оптимизированный код.

Создайте оптимизированный вариант программы **demo_lib2**:

```
make -f Makefile_opt clean
make -f Makefile_opt
```

Запустите этот вариант программы под управлением отладчика:

```
gdb ./demo_lib2.exe
```

Как и прежде, назначьте первую точку останова на функции main():

```
b main
```

Затем с помощью команды **r** (**run**) запустите программу и пройдите ее всю с помощью команды **s** (step). Эта команда позволит вам зайти внутрь каждой функции, вызываемой из функции main().

Выполняя программу в пошаговом режиме, обратите внимание на следующие моменты:

– операторы в функциях clircle_len(), circle_area(), square_perim(), square_area() выполняются не в логическом порядке, который предписан исходным текстом программы, а в ином порядке. При этом некоторые операторы повторяются более одного раза, хотя в этих функциях нет циклов;

– операторы return отладчик игнорирует вовсе.

Фрагмент сеанса работы с программой в отладчике показан ниже. Как можно предположить на основе этого фрагмента, проверка условия if() производится трижды.

```
31        len = 2 * PI * rad;
(gdb) s
33        if ( formula )
(gdb) s
```

```
31          len = 2 * PI * rad;
(gdb) s
33          if ( formula )
(gdb) s
31          len = 2 * PI * rad;
(gdb) s
33          if ( formula )
```

Таким образом, вы видите, что отладка оптимизированного кода представляет собой несколько необычный процесс и потому требует определенного опыта. Поэтому можно рекомендовать вам включать параметры оптимизации для компилятора только после завершения отладки программы в неоптимизированном варианте. Здесь под словом «оптимизация» понимается работа компилятора, а не ваша творческая деятельность по улучшению алгоритмов, используемых вами в программе.

2. Проведите подобный эксперимент с более сложной программой. В качестве таковой можно использовать, например, одну из программ, представленных на сайте http://www.gnu.org или на другом сайте, содержащем бесплатные исходные тексты программных продуктов.

3. В отладчике языка Perl запустите более сложную программу, чем та, которую мы рассмотрели в этой главе. Если это ваша программа, в которой есть логическая ошибка, то попытайтесь найти ошибку с помощью отладчика.

4. Подумайте, каким образом в отладчике языка Perl можно избежать, если потребуется, ручного прохождения цикла, содержащего, например, 1000 итераций. Под ручным прохождением цикла мы понимаем следующую процедуру: вы вводите команду **n** (**next**) или **s** (**step**) на первом операторе цикла, а затем просто нажимаете клавишу Enter для повторения этой команды. Но в данном случае таких «простых» нажатий будет очень много, что не всегда целесообразно.

7. Интернационализация и локализация программного обеспечения

В наши дни правилом хорошего тона считается организация диалога программы с пользователем на родном для него языке. Один из подходов к решению этой задачи представлен в настоящей главе.

7.1. Терминология и стандарты

Термины *интернационализация* (internationalization) и *локализация* (localization) широко используются в среде разработчиков программного обеспечения. В англоязычной литературе и в сети Internet эти термины зачастую заменяются короткими аббревиатурами – I18n и L10n соответственно.

Для успешного распространения программного продукта в различных странах желательно, чтобы пользовательский интерфейс был представлен на родном для пользователей языке. Кроме того, в разных странах приняты различные форматы написания числовых величин, отличаются способы представления дат и т. д. Было время, когда производители программного обеспечения поддерживали ряд идентичных в функциональном отношении версий одного и того же программного продукта, предназначенных для распространения в различных географических регионах мира. Сейчас уже очевидно, что это не самое лучшее решение.

Под локализацией понимается перевод на конкретный язык всех текстовых сообщений, инструкций и документации программного продукта, изменение форматов вывода дат и времени, числовых и денежных величин, изменение порядка сортировки алфавитных символов в соответствии с традициями той страны, в которую поставляется данный программный продукт. Набор вышеперечисленных параметров по-английски называется locale, т. е. параметры локализации. В среде операционной системы FreeBSD эти параметры можно просмотреть с помощью команды **locale**.

Под интернационализацией понимается такой способ проектирования программного продукта, при котором в исходные тексты программ включаются некоторые избыточные конструкции. Однако наличие таких конструкций позволяет значительно упростить процесс проведения локализации. Можно упрощенно сказать так: интернационализация – это некое обобщение, универсализации, а локализация – это конкретизация, подгонка программного продукта под нужны пользователей в определенной стране. Поскольку разработка программы с учетом требований интернационализации выполняется однократно, а процедуры локализации – многократно, то подобный подход оказывается оправданным в экономическом и техническом отношении. Как вы увидите, процесс разработки исходных текстов может быть отделен от процедуры локализации, т. е. программисты и

переводчики могут работать независимо друг от друга. Это повышает надежность локализованных программ.

Кроме проблемы наличия различных естественных языков существует и проблема наличия различных кодовых страниц (code page, code set, character set), или кодировок символов, даже для одного и того же языка. Например, для русского языка в операционной системе UNIX традиционно используется кодировка KOI8-R, а в среде операционной системы Windows – кодировки CP1251 и CP866 (последняя известна также под именем «альтернативной» кодировки). Кодовая страница – это таблица, ставящая в соответствие каждому символу определенный числовой код. Например, заглавная латинская буква A имеет числовой код 65 (причем, во многих кодовых страницах).

Предположим, что мы используем какую-либо программу, имеющую русскоязычный интерфейс, в среде UNIX и в среде Windows. В этом случае нам придется выполнить перекодирование символьных строк, содержащихся в программе, в кодировку KOI8-R для работы в среде UNIX и в кодировку CP1251 для работы в среде Windows. Обратите внимание, что речь идет не о переводе с одного языка на другой, а лишь об изменении способа кодирования русскоязычных символов.

Если ваша программа требует наличия поддержки различных кодировок, то вы можете использовать библиотеку **iconv** (http://www.gnu.org/software/libiconv). В состав дистрибутивного набора библиотеки iconv входит также и утилита iconv, которая позволяет выполнять перекодирование текстовых файлов (например, исходных текстов программ) из командной строки. В среде операционной системы FreeBSD вы можете установить эту библиотеку, пользуясь CD-дисками из установочного комплекта операционной системы. Проще всего выполнить эту процедуру с помощью утилиты **sysinstall**. Если вы уже установили графическую среду KDE, то тогда эта библиотека у вас также установлена.

Загрузить скомпилированные библиотеки и утилиты **iconv** для системы Windows можно с сайта http://www.mingw.org. Вам нужно найти архивный файл **libiconv-x.x-bin.tar.bz2** на этом сайте. Для установки программного продукта нужно сначала разархивировать полученный файл. В нем представлена иерархия каталогов, начиная с каталога **/usr**. Содержимое архива нужно скопировать в структуру каталогов среды MSYS, но при этом следует учесть, что каталог /usr монтируется на Windows-каталог **C:\msys\1.0**. Таким образом, путь **/usr/local** соответствует Windows-пути **C:\msys\1.0\local**.

В качестве примера использования утилиты **iconv** в операционной системе FreeBSD рассмотрим перекодирование текстового файла, например, **hello.c**, представленного в кодировке KOI8-R, в кодировку CP1251:

```
iconv –f KOI8-R –t CP1251 hello.c > hello.c.win
```

Обратите внимание на то, что новый, перекодированный, файл направляется на стандартный вывод и поэтому его можно перенаправить в файл. В среде Windows (MSYS) эта утилита работает аналогично, но имеет имя **iconv.exe**.

В заключение скажем несколько слов о стандарте UNICODE. Этот стандарт был предложен в качестве решения проблемы наличия многих языков и множества различных кодовых страниц (кодировок). В традиционных кодовых страницах для кодирования одного символа служит один байт. Таким образом, с помощью одной такой таблицы можно закодировать только 256 различных символов. Поэтому возникают трудности с использованием в программах одновременно нескольких естественных языков (например, русского и датского). Если же для кодирования одного символа допустить использование более чем одного байта, то тогда эта проблема была бы решена. В стандарте UNICODE (http://www.unicode.org) предложены различные формы (схемы) кодирования символов. Форма кодирования UTF-8 предполагает использование от одного до четырех байтов для кодирования одного символа. При этом для представления самых распространенных символов используются однобайтовые коды. Форма кодирования UTF-16 предписывает использование одного или двух двухбайтовых блоков для кодирования каждого символа, а в UTF-32 используются только четырехбайтовые блоки.

7.2. Использование утилиты gettext

Одним из способов интернационализации программ является использование библиотеки **gettext**. Если она отсутствует в вашей системе FreeBSD, то установите ее с помощью утилиты **sysinstall** с установочного набора CD-дисков.

Мы покажем использование библиотеки gettext на простом примере и ограничимся только переводом текстовых сообщений, выводимых программой. За рамками нашего рассмотрения остаются другие вопросы локализации, такие, как настройка форматов ввода и вывода числовых величин, дат, времени, вопросы сортировки символьных строк в алфавитном порядке и т. д. Более подробно изучить эти вопросы можно, например, обратившись к библиотеке **ICU** (http://icu-project.org).

Первый шаг на пути к получению локализованной программы, это включение в нее вызовов специальной функции gettext(). Применение этой функции позволит получать тексты сообщений, выводимых программой, на родном языке пользователя, а не на английском языке.

Программа **hello.c**

```
// ------------------------------------------------------------
// Программа для изучения способа локализации приложений
```

```
// ------------------------------------------------------------

#include <libintl.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

#define _( STRING ) gettext( STRING )

int main( void )
{
  setlocale( LC_ALL, "" );
  // bindtextdomain( "hello", "/usr/local/share/locale" );
  bindtextdomain( "hello", "./" );
  textdomain( "hello" );

  // printf( gettext("Hello, world!\n") );
  printf( _( "Hello, world!\n" ) );
  printf( _( "How are you, world?\n" ) );
  printf( _( "Thank you, I'm fine!\n" ) );

  exit( 0 );
}
```

Сделаем пояснения к приведенной программе.

1. Обратите внимание на включение заголовочных файлов **libintl.h** и **locale.h**.

2. Функции setlocale(), bindtextdomain() и textdomain() делают всю подготовительную работу в программе. Первая из них – setlocale() – имеет два параметра. Первый из них указывает, что необходимо назначить *все* настройки локализации, т. е. LC_COLLATE, LC_MESSAGES, LC_NUMERIC, LC_CTYPE, LC_MONETARY и LC_TIME. Второй параметр – пустая строка, что означает использование настроек локализации, заданных в операционной системе (вспомните о переменной LANG, которая назначается в файле **/etc/profile** в системе FreeBSD). Функция textdomain() назначает имя домену текстовых сообщений, из которого будут выбираться сообщения, выводимые нашей программой. Имя домена совпадает с именем программы. Функция bindtextdomain() указывает базовый каталог для домена текстовых сообщений. В нашем примере это текущий каталог, в котором находится программа.

3. Использование макроподстановки позволяет не включать вызовы функции gettext() в текст программы, т. к. эту работу сделает теперь препроцессор.

4. При использовании библиотеки **gettext** предполагается, что в тексте программы строки представлены на *английском* языке, а не на русском или каком-то другом.

Теперь откомпилируйте программу (эта команда должна вводиться на одной строке):

```
gcc -o hello hello.c -I/usr/local/include -L/usr/local/lib
-lintl
```

Следующий шаг – извлечение текстовых строк, подлежащих переводу на другой язык:

```
xgettext -d hello -k_ -o hello.pot hello.c
```

В этой команде параметр **–d** указывает имя домена текстовых сообщений. Это имя должно совпадать с именем, переданным в качестве параметра функции textdomain(), но может не совпадать с именем программы. Параметр **–k** нужен для того, чтобы утилита **xgettext** «признавала» макроподстановку _() в качестве ключевого слова. По умолчанию эта утилита извлекает только символьные строки, являющиеся параметрами функции gettext(). Обратите внимание на отсутствие пробела между **–k** и символом подчеркивания. Параметр **–o** указывает имя результирующего файла. Расширение .pot означает portable object template. Этот файл используется в качестве шаблона для перевода текстовых сообщений на другой язык.

Следующий шаг – создание файла, содержащего переведенные текстовые строки. Для начала нужно из файла-шаблона получить файл-заготовку для выполнения перевода:

```
msginit -l ru_RU.KOI8-R -o hello_ru.po -i hello.pot
```

Первый параметр **–l** (**locale**) указывает утилите **msginit**, какие параметры локализации использовать при генерировании файла-заготовки **hello_ru.po**. В записи вида ru_RU.KOI8-R символы ru означают язык (русский), символы RU – страну (Россия), KOI8-R – вид кодировки символов.

В процессе генерирования файла **hello_ru.po** будет выведен запрос насчет вашего адреса электронной почты.

Теперь необходимо перевести все символьные строки, извлеченные из файла **hello.c** и помещенные в файл **hello_ru.po**. Для работы с PO-файлом можно использовать текстовый редактор общего назначения, например, **joe** или **emacs**, хотя существуют и специальные редакторы PO-файлов. При использовании редактора общего назначения важно соблюдать формат PO-файла.

Начать следует с перевода некоторых фрагментов полей заголовка, в частности, тех, которые написаны заглавными буквами. Например, в строке

```
"Project-Id-Version: PACKAGE VERSION\n"
```

укажите имя и версию вашей программы, например:

```
"Project-Id-Version: Hello 1.0\n"
```

При переводе символьных строк необходимо соблюдать формат записей PO-файла. Он таков:

```
WHITE-SPACE (пустая строка)
#  TRANSLATOR-COMMENTS (комментарии переводчика)
#. AUTOMATIC-COMMENTS (автоматические комментарии)
#: REFERENCE... (ссылка)
#, FLAG... (флаг)
msgid UNTRANSLATED-STRING (оригинальная строка)
msgstr TRANSLATED-STRING (переведенная строка)
```

В качестве примера приведем одну запись из файла **hello_ru.po**:

```
#: hello.c:20
#, c-format
msgid "Hello, world!\n"
msgstr "Привет, мир!\n"
```

Завершив перевод символьных строк в файле **hello_ru.po**, нужно скомпилировать его в файл специального двоичного формата, который позволяет ускорить поиск переведенных символьных строк в процессе работы программы. Команда для выполнения этой процедуры такова:

```
msgfmt -c -v -o hello.mo hello_ru.po
```

Если вы допустили нарушения формата PO-файла в процессе его редактирования, то утилита **msgfmt** выведет сообщения об ошибках. Если же все сделано корректно, то сообщение будет таким:

```
3 переведенных сообщения.
```

Остался последний шаг – размещение скомпилированного MO-файла (machine object) с текстами сообщений в том каталоге операционной системы, который был указан в качестве параметра функции bindtextdomain(). В нашем примере это текущий каталог. Выполните команды:

```
mkdir -p ru_RU/LC_MESSAGES
cp hello.mo ./ru_RU/LC_MESSAGES
```

В результате в текущем каталоге будет создан подкаталог **ru_RU**, а в нем подкаталог **LC_MESSAGES**, в который и будет скопирован файл **hello.mo**.

Теперь настал момент проверки функционирования локализованной программы. Введите команду

```
./hello
```

Она должна вывести на экран такие строки:

```
Привет, мир!
Как дела, мир?
Спасибо, прекрасно!
```

Конечно, текст этих сообщений может быть и немножко другим, если вы использовали более вольный перевод на русский язык.

В операционной системе Windows

При использовании среды MSYS в операционной системе Windows вы не должны испытать каких-то особенных затруднений. Небольшие различия следующие.

При выполнении команды

```
msginit -l ru_RU.KOI8-R -o hello_ru.po -i hello.pot
```

могут быть выведены четыре сообщения об ошибках, но все равно выводится и сообщение об успехе:

```
Создано hello_ru.po.
```

При этом имя кодовой страницы KOI8-R игнорируется – получается CP1251, как вы можете увидеть в заголовке полученного файла **hello_ru.po**:

```
"Content-Type: text/plain; charset=CP1251\n"
```

Таким образом, можно эту команду упростить:

```
msginit -l ru_RU -o hello_ru.po -i hello.pot
```

Если попытаться изменить базовый каталог для сообщений с текущего каталога, т. е. «**./**», на системный каталог **/usr/local/share/locale**, то это перевод может не работать. Тогда вместо строки

```
bindtextdomain( "hello", "/usr/local/share/locale" );
```

введите

```
bindtextdomain( "hello",
                "c:\\msys\\1.0\\local\\share\\locale" );
```

Тем самым вы вместо имени каталога, формируемого средой MSYS, укажете программе имя каталога непосредственно в стиле Windows.

В каталоге **C:\msys\1.0\local\share\locale** уже есть подкаталог **ru\LC_MESSAGES**. Можно скопировать файл **hello.mo** в него. Но можно создать подкаталог **ru_RU\LC_MESSAGES** и скопировать файл **hello.mo** в него. Внеся изменения в текст программы **hello.c**, не забудьте ее перекомпилировать.

Контрольные вопросы и задания

1. Как вы думаете, почему термины internationalization и localization заменяются именно такими аббревиатурами – I18n и L10n соответственно?

2. С помощью электронных руководств man выясните назначение каждого из параметров (категорий) локализации LC_COLLATE, LC_MESSAGES, LC_NUMERIC, LC_CTYPE, LC_MONETARY и LC_TIME:

```
man locale
man setlocale
```

3. Посмотрите содержимое каталога **/usr/share/locale**. Вы увидите, что для русского языка там представлено несколько вариантов локализации: ru_RU.CP1251, ru_RU.CP866, ru_RU.ISO8859-5, ru_RU.KOI8-R, ru_RU.UTF-8. Посмотрите содержимое каждого из этих подкаталогов.

4. В среде FreeBSD выполните команду создания POT-файла в такой форме:

```
msginit -l ru_RU -o hello_ru.po -i hello.pot
```

(т. е. без параметра KOI8-R) и сравните результирующий файл с тем, который был получен при наличии этого параметра.

5. Выполните локализацию для немецкого языка в среде FreeBSD.

ПРИМЕЧАНИЕ. Если вы не знаете этого языка, то сделайте упрощенные переводы английских сообщений на немецкий язык, главное, чтобы они отличались от русских переводов, выполненных вами при изучении материала этой главы.

При этом учтите, что имя результирующего MO-файла будет таким же, как и при «русификации» этой программы, т. е. **hello.mo**. Поэтому важно не перепутать русский и немецкий варианты переводов.

```
msginit -l de_DE.ISO8859-1 -o hello_de.po -i hello.pot
msgfmt -c -v -o hello.mo hello_de.po
mkdir -p de_DE/LC_MESSAGES
cp hello.mo ./ru_RU/LC_MESSAGES
```

Если вы используете файловый менеджер, например, **deco**, то выйдите из него.

Теперь выполните команду, которая предпишет операционной системе использовать параметры локализации для немецкого языка:

```
LANG=de_DE.ISO8859-1; export LANG
```

Если вы используете не **Bourne shell**, а **C shell**, то команда будет немного отличаться:

```
setenv LANG de_DE.ISO8859-1
```

Сначала с помощью команды **locale** проверьте, удалось ли вам создать среду немецкого языка в вашей операционной системе. Должно быть выведено примерно следующее:

```
LANG=de_DE.ISO8859-1
LC_CTYPE="de_DE.ISO8859-1"
LC_COLLATE="de_DE.ISO8859-1"
LC_TIME="de_DE.ISO8859-1"
LC_NUMERIC="de_DE.ISO8859-1"
LC_MONETARY="de_DE.ISO8859-1"
LC_MESSAGES="de_DE.ISO8859-1"
LC_ALL=
```

Если среда создана, то запустите программу:

```
./hello
```

Если сообщения не стали немецкими (в той степени, в какой вы владеете этим языком), то проверьте, все ли вы сделали правильно.

6. В операционной системе FreeBSD существует специальный каталог **/usr/local/share/locale**. Можно использовать его в качестве базового каталога для вашего файла **hello.mo**. Попробуйте модифицировать программу **hello.c** соответствующим образом.

8. Базы данных

Пожалуй, практически каждый программист прямо или косвенно соприкасается с базами данных. В настоящей главе мы представим систему управления базами данных (СУБД) PostgreSQL. Это одна из наиболее известных свободно распространяемых СУБД. Мы не только опишем процедуру ее установки, но также продемонстрируем способы получения доступа к базе данных из прикладных программ, написанных на языках C и Perl.

8.1. Основные понятия

Это элементарное введение содержит лишь сведения, необходимые для осмысленного выполнения процедуры установки и настройки СУБД PostgreSQL, а также для выполнения примеров программ. Для получения основательной подготовки в области баз данных нужно обратиться к соответствующей литературе.

Система баз данных – это компьютеризированная система, предназначенная для хранения, переработки и выдачи информации по запросу пользователей. Такая система включает в себя программное и аппаратное обеспечение, сами данные, а также пользователей.

Современные системы баз данных являются, как правило, многопользовательскими. В таких системах одновременный доступ к базе данных могут получить сразу несколько пользователей.

Основным программным обеспечением является система управления базами данных. По-английски она называется database management system (DBMS). Кроме СУБД в систему баз данных могут входить утилиты, средства для разработки приложений (программ), средства проектирования базы данных, генераторы отчетов и др.

Пользователи систем с базами данных подразделяются на ряд категорий. Первая категория – это прикладные программисты. Вторая категория – это конечные пользователи, ради которых и выполняется вся работа. Они могут получить доступ к базе данных, используя прикладные программы или универсальные приложения, которые входят в программное обеспечение самой СУБД. В большинстве СУБД есть так называемый **процессор языка запросов**, который позволяет пользователю вводить команды языка высокого уровня (например, языка SQL). Третья категория пользователей – это администраторы базы данных. В их обязанности входят: создание базы данных, выбор оптимальных режимов доступа к ней, разграничение полномочий различных пользователей на доступ к той или иной информации в базе данных, выполнение резервного копирования базы данных и т. д.

Систему баз данных можно разделить на два главных компонента: сервер и набор клиентов (или внешних интерфейсов). Сервер – это и есть СУБД. Клиентами являются различные приложения, написанные прикла-

ными программистами, или встроенные приложения, поставляемые вместе с СУБД. Один сервер может обслуживать много клиентов.

Современные СУБД включают в себя словарь данных. Это часть базы данных, которая описывает сами данные, хранящиеся в ней. Словарь данных помогает СУБД выполнять свои функции.

В настоящее время преобладают базы данных реляционного типа. Их характерной чертой является тот факт, что данные воспринимаются пользователем как таблицы. В распоряжении пользователя имеются операторы для выборки данных из таблиц, а также для вставки новых данных, обновления и удаления имеющихся данных.

Рассмотрим простую систему, в которой всего две таблицы: «Студенты» и «Успеваемость».

Таблица «Студенты»

Номер зачетной книжки	Ф. И. О.	Серия паспорта	Номер паспорта
55500	Иванов Иван Петрович	0402	645327
55800	Климов Андрей Иванович	0402	673211
55865	Новиков Николай Юрьевич	0202	554390

Таблица «Успеваемость»

Номер зачетной книжки	Предмет	Учебный год	Семестр	Оценка
55500	Физика	2000/2001	1	5
55500	Математика	2000/2001	1	4
55800	Физика	2000/2001	1	4
55800	Физика	2000/2001	2	5

Строки таких таблиц называются **записями,** а столбцы – **полями**. На пересечении строк и столбцов должны находиться «атомарные» значения, которые нельзя разбить на какие-либо элементы без потери смысла.

В теории баз данных эти таблицы называют **отношениями (relation)** – поэтому и базы данных называются реляционными. Отношение – это математический термин. При определении свойств таких отношений используется теория множеств. В терминах данной теории строки таблицы будут называться **кортежами**, а колонки – **атрибутами**. Отношение имеет заголовок, который состоит из атрибутов, и тело, состоящее из кортежей. Количество атрибутов называется **степенью отношения**, а количество кортежей – **кардинальным числом**.

При работе с базой данных часто приходится следовать различным ограничениям. В нашем случае ограничения следующие:

– номер зачетной книжки состоит из пяти цифр и не может быть отрицательным;

– номер семестра может принимать только два значения – 1 и 2;

– оценка может принимать только три значения – 3, 4 и 5.

Для идентификации строк в таблицах и для связи таблиц между собой используются так называемые ключи. **Потенциальный ключ** – это уникальный идентификатор строки в таблице базы данных. Он состоит из одного или нескольких полей этой строки. Например, в таблице «Студенты» таким идентификатором может быть поле «Номер зачетной книжки», а могут быть и два поля, взятые вместе – «Серия паспорта» и «Номер паспорта». В последнем случае ключ будет составным. При этом важным является то, что потенциальный ключ должен быть *не избыточным*, т. е. никакое подмножество полей, входящих в него, не должно обладать свойством уникальности. В нашем примере ни поле «Серия паспорта», ни поле «Номер паспорта» в отдельности не могут использоваться в качестве уникального идентификатора.

Ключи нужны для адресации на уровне строк (записей). При наличии в таблице более одного потенциального ключа один из них выбирается в качестве так называемого **первичного ключа**, а остальные будут являться **альтернативными ключами**.

Рассмотрим таблицы «Студенты» и «Успеваемость». Предположим, что в таблице «Студенты» нет строки с номером зачетной книжки 55900, тогда включать строку с таким номером зачетной книжки в таблицу «Успеваемость» не имеет смысла. Таким образом, значения поля «Номер зачетной книжки» в таблице «Успеваемость» должны быть согласованы со значениями такого же поля в таблице «Студенты». Поле «Номер зачетной книжки» в таблице «Успеваемость» является примером того, что называется **внешним ключом**. Говорят, что «внешний ключ ссылается на потенциальный ключ в ссылочной таблице». Внешний ключ может быть составным, т. е. может включать более одного поля. Внешний ключ не обязан быть уникальным. Проблема обеспечения того, чтобы база данных не содержала неверных значений внешних ключей, известна как проблема **ссылочной целостности**. Ограничение, согласно которому значения внешних ключей должны соответствовать значениям потенциальных ключей, называется **ограничением ссылочной целостности (ссылочным ограничением)**. Таблица, содержащая внешний ключ, называется **ссылающейся** таблицей, а таблица, содержащая соответствующий потенциальный ключ, – **ссылочной (целевой)** таблицей.

Для обеспечения ограничений ссылочной целостности применяются специальные способы проектирования базы данных. Предусматривается, что при удалении записи из ссылочной таблицы соответствующие записи из ссылающейся таблицы должны быть также удалены, а при изменении значения поля, на которое ссылается внешний ключ, должны быть изменены значения внешнего ключа в ссылающейся таблице. Этот подход называется **каскадным удалением (обновлением)**.

Иногда применяются и другие подходы. Например, вместо удаления записей из ссылающейся таблицы в этих записях просто заменяют значения полей, входящих во внешний ключ, так называемыми NULL-значениями. Это специальные значения, означающие «ничто», или отсут-

ствие значения, они не совпадают со значением «нуль» или «пустая строка». NULL-значение применяется в базах данных и в качестве значения по умолчанию, когда пользователь не ввел никакого конкретного значения. Первичные ключи не могут содержать NULL-значений.

Транзакция – одно из важнейших понятий теории баз данных. Она означает набор операций над базой данных, рассматриваемых как единая и неделимая единица работы, выполняемая полностью или не выполняемая вовсе, если произошел какой-то сбой в процессе выполнения транзакции. В нашей базе данных транзакцией могут быть, например, две операции: удаление записи из таблицы «Студенты» и удаление связанных по внешнему ключу записей из таблицы «Успеваемость».

8.2. Установка СУБД PostgreSQL

Загрузить исходные тексты этой СУБД можно с сайта http://www.postgresql.org. Выберите для загрузки последнюю **стабильную** версию архивного файла с именем вида **postgresql-8.x.x.tar.gz** или **postgresql-8.x.x.tar.bz2**. Здесь символами x обозначены младшие номера версий, которые подвержены более частым изменениям, чем старший номер версии, который на данном «историческом» отрезке времени имеет номер 8.

Подготовка к установке СУБД заключается в следующем. Зарегистрируйтесь в системе под именем root. Скопируйте архивный файл в каталог **/usr/src** (можно, конечно, выбрать и другой каталог), извлеките файлы из архива, а затем перейдите во вновь созданный подкаталог **postgresql-8.x.x**:

```
cp postgresql-8.x.x.tar.gz /usr/src
cd /usr/src
tar xzvf postgresql-8.x.x.tar.gz
cd /usr/src/postgresql-8.x.x
```

Процедура установки состоит из ряда шагов, которые мы будем нумеровать для придания всей процедуре большей четкости.

1. Первым шагом является конфигурирование иерархии исходных текстов СУБД. Эта задача решается традиционным для свободного (поставляемого в исходных текстах) программного обеспечения способом, а именно, путем запуска программы **configure**. Она может получать различные параметры в зависимости от требований, предъявляемых к устанавливаемому программному продукту, и с учетом настроек операционной системы. Программа (скрипт) **configure** написана на языке shell. Однако подобные скрипты не пишутся вручную, а создаются с помощью специальных программ автоматического конфигурирования.

Мы рекомендуем вам скомпилировать СУБД PostgreSQL с поддержкой отладчика и отключенной оптимизацией объектного кода. Первая задача решается путем использования параметра **--enable-debug** (обратите внимание на два дефиса перед этим параметром), а вторая – путем назначения переменной CFLAGS значения **–O0**, которое предписывает компилятору генерировать неоптимизированный код. Список всех параметров, принимаемых программой configure, можно получить с помощью команды

```
./configure –-help
```

Для выполнения поставленной задачи введите команду

```
./configure --enable-debug CFLAGS=-O0 > conf_log.txt 2>&1 &
```

Все сообщения, выводимые в процессе конфигурирования, можно перенаправить в файл-журнал **conf_log.txt**. В него будут поступать как сообщения, направляемые на стандартный вывод, так и сообщения об ошибках (этим управляет компонент 2>&1). Кроме того, программа configure сама создает файлы-журналы в текущем каталоге. Однако они не являются точной копией файла **conf_log.txt**. Последний знак & в командной строке означает, что команда будет выполняться в фоновом режиме. Поэтому вы можете наблюдать процесс с помощью команды

```
tail –f conf_log.txt
```

В любой момент вы можете прервать этот просмотр нажатием клавишей Ctrl-C, при этом процесс будет продолжаться. Конечно, вы можете запустить эту команду и в обычном режиме, а не как фоновый процесс.

По окончании процедуры конфигурирования загляните в файл **conf_log.txt**. Если в конце файла нет сообщения об ошибке, то все в порядке.

2. Следующий этап – компиляция программ. Для этой цели нужно использовать только утилиту GNU **make** (ее имя **gmake**). Если она не установлена в вашей операционной системе, то установите ее, загрузив исходные тексты с сайта http://www.gnu.org.

```
gmake > gmake_log.txt 2>&1 &
tail -f gmake_log.txt
```

Процесс может занять от двух–трех минут до получаса, в зависимости от производительности вашего компьютера. Последняя строка, выведенная в файл-журнал, должна быть такой:

```
All of PostgreSQL is successfully made. Ready to install.
```

3. Теперь нужно установить скомпилированные программы в системные каталоги операционной системы:

```
gmake install > gmake_install_log.txt 2>&1 &
tail -f gmake_install_log.txt
```

4. Для удобства использования утилит, входящих в комплект СУБД, рекомендуется добавить путь к этим утилитам в переменную среды PATH. Если вы используете **Bourne shell** (**/bin/sh**), то корректировать нужно файл **~/.profile** (**~** означает домашний каталог пользователя):

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

Если вы используете **csh** or **tcsh**, тогда команда будет такой (изменения нужно внести в файл **~/.cshrc**):

```
set path = ( /usr/local/pgsql/bin $path )
```

ПРИМЕЧАНИЕ. При установке операционной системы FreeBSD устанавливаются и утилиты СУБД PostgreSQL, но они имеют более старую версию. Поэтому, если вы не выполнили рекомендуемой корректировки переменной PATH, то при запуске утилит без указания полного пути к ним вы получите доступ к их более старым версиям. Так что будьте внимательны.

---

5. Для запуска сервера СУБД PostgreSQL необходимо наличие специальной учетной записи в операционной системе. Как правило, такой пользователь имеет имя postgres. Рекомендуем вам также использовать это имя. Создайте учетную запись с помощью команды **adduser**. Основную часть параметров можно принимать по умолчанию. Если у вас нет особых причин сделать иначе, то выберите в качестве командного интерпретатора **Bourne shell** (**sh**) из списка, предложенного утилитой **adduser**.

6. Теперь необходимо инициализировать кластер баз данных. Создайте каталог, в котором будет располагаться база данных. Затем измените владельца этого каталога на пользователя postgres:

```
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
```

Войдите в систему под именем postgres или, используя команду **su**, «притворитесь» этим пользователем (обратите внимание на пробелы слева и справа от дефиса):

```
su - postgres
```

Инициализация кластера баз данных выполняется так:

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

На экран выводится целый ряд сообщений. Обратите внимание на информацию о параметрах локализации:

```
The database cluster will be initialized with locale
ru_RU.KOI8-R.
The default database encoding has accordingly been set to
KOI8.
```

Последние сообщения такие:

```
Success. You can now start the database server using:

    /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
or
    /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l
logfile start
```

7. СУБД успешно установлена, значит, можно попытаться запустить серверный процесс.

ПРИМЕЧАНИЕ. Это выполняется только от имени пользователя postgres (т. е. суперпользователя СУБД), а не от имени пользователя root.

Войдите в систему под именем пользователя postgres и выполните команду (она вводится на одной строке)

```
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data
-l logfile start
```

На экран будет выведено сообщение

```
server starting
```

Проверьте, запущен ли серверный процесс, с помощью команды

```
ps -axw
```

На экране вы найдете примерно такую строку:

```
37561  v0  I      0:00,06 /usr/local/pgsql/bin/postgres -D
/usr/local/pgsql/data
```

Если вы хотите, чтобы сервер СУБД PostgreSQL автоматически запускался при загрузке операционной системы и выгружался при ее выключении, то используйте файлы **/etc/rc.local** и **/etc/rc.shutdown.local** (если та-

ких файлов еще нет, создайте их). Для выполнения операций с этими файлами необходимо зарегистрироваться в системе под именем пользователя root.

В файл **/etc/rc.local** добавьте команду

```
su - postgres -c '/usr/local/pgsql/bin/pg_ctl -D
/usr/local/pgsql/data -l logfile start'
```

Эту команду можно ввести в одну строку или использовать символ «\» для продолжения команды на следующей строке, аналогично тому, как в программах на языке С «склеиваются» строковые константы.

В файл **/etc/rc.shutdown.local** добавьте команду

```
su - postgres -c '/usr/local/pgsql/bin/pg_ctl -D
/usr/local/pgsql/data stop'
```

Обе команды содержат следующий компонент:

```
su – postgres
```

Он указывает операционной системе, что выполнять команду, заключенную в одинарные кавычки, необходимо от имени пользователя postgres. При этом обратите внимание на наличие пробелов слева и справа от дефиса.

При первом запуске серверного процесса создается файл-журнал **logfile** в каталоге **/home/postgres**.

Если вы все сделали правильно, то при помощи команды

```
ps –axw
```

вы можете увидеть, что сервер успешно запущен:

```
779  ??  Ss     0:00,03 postgres: writer process    (postgres)
780  ??  Ss     0:00,00 postgres: stats collector process
(postgres)
651 con- I      0:00,23 /usr/local/pgsql/bin/postgres -D
/usr/local/pgsql/data
```

8. Если сервер СУБД запущен, можно создать базу данных. В качестве ее имени выберем **test**. От имени пользователя postgres выполните команду

```
/usr/local/pgsql/bin/createdb test
```

На экран должно быть выведено сообщение:

```
CREATE DATABASE
```

9. Пробуем подключиться к только что созданной базе данных **test** (опять как пользователь postgres):

```
/usr/local/pgsql/bin/psql test
```

На экране вы увидите:

```
Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit
```

Символ «\» в этих командах необходимо вводить буквально. Например, для выхода введите \q.

На этом установка СУБД PostgreSQL может считаться в основном завершенной. Добавим лишь несколько замечаний. Важно помнить, что учетная запись пользователя в операционной системе и учетная запись пользователя в СУБД – это разные вещи, хотя их имена могут совпадать, как это имеет место для пользователя postgres. Пользователь root, имеющий особое значение с точки зрения операционной системы, с точки зрения СУБД PostgreSQL такого значения не имеет. Поэтому, если вы хотите, чтобы в рамках СУБД была учетная запись с именем root, то ее нужно создать с помощью SQL-команды **CREATE USER**. Для этого войдите в операционную систему как пользователь postgres и подключитесь к базе данных **test** с помощью команды

```
/usr/local/pgsql/bin/psql test
```

Если вы добавите путь **/usr/local/pgsql/bin** в переменную PATH для пользователя postgres, как это было рекомендовано выше, то команда будет более короткой:

```
psql test
```

Получив доступ к базе данных, в ответ на приглашение

```
test=#
```

выполните команду (не забудьте поставить точку с запятой в конце команды)

```
create user root;
```

При успешном выполнении этой команды СУБД «ответит» вам:

```
CREATE ROLE
```

Выйдите из программы **psql** и попробуйте снова подключиться к базе данных **test**, но уже как пользователь root:

```
psql –d test –U root
```

Если вы не указали имя пользователя с помощью параметра **–U**, то по умолчанию используется то же имя, которое имеет текущий пользователь операционной системы. Однако если в СУБД нет пользователя с таким именем, вы получите сообщение об ошибке.

Рекомендуем вам поэкспериментировать с утилитой **psql**, подключаясь к базе данных **test** от имени разных пользователей СУБД, регистрируясь в операционной системе также под разными именами.

Конечно, настоящее краткое руководство не может и не должно заменять документацию, входящую в комплект поставки СУБД. Традиционно мы рекомендуем обратиться к электронным руководствам **man**. Только не забывайте, что при установке операционной системы FreeBSD устанавливаются и утилиты PostgreSQL более ранних версий, а с ними устанавливаются и man-страницы. Поэтому для получения доступа к документации, установленной в каталог **/usr/local/pgsql/man**, вы можете поступить так:

```
cd /usr/local/pgsql/man
man -M . psql
```

или даже так:

```
man -M /usr/local/pgsql/man psql
```

Вместо **psql** указывайте имя одной из интересующей вас утилит, которые собраны в каталоге **/usr/local/pgsql/bin**.

В комплект поставки входит также и очень подробная документация в формате HTML. Она находится в каталоге **/usr/local/pgsql/doc/html**. Стартовый файл, содержащий оглавление, – **index.html**.

В операционной системе Windows

Инсталляционный архивный файл, который также можно загрузить с сайта http://www.postgresql.org, содержащий уже скомпилированные программы, имеет, как правило, расширение .zip. В этом архиве содержится установочный файл, представленный в формате MSI. Необходимо извлечь файлы из архива и запустить программу **postgresql-8.x.msi**.

Установите СУБД в каталог, путь к которому не содержит пробелов, например, **C:\Pg** или **C:\pgsql**. Не устанавливайте СУБД в традиционный каталог **C:\Program Files**, т. к. путь к нему содержит пробельный символ. Это требование важно потому, что многие утилиты, входящие в комплект MSYS и MinGW, были «рождены» в среде UNIX и не «любят» пробелы в именах каталогов.

При запуске утилиты **psql** в среде Windows нужно сделать следующее. Выберите строку CommandPrompt из списка доступных опций в группе PostgreSQL 8.x, находящейся в списке «Все программы» (кнопка «Пуск»). Откроется окно с командной строкой. Сначала необходимо изменить свойства этого окна, а именно: выбрать шрифт Lucida Console. Затем уже в окне введите команду

```
chcp 1251
```

Эта команда назначает текущую кодовую страницу (вспомните о параметрах локализации, о которых мы говорили в предыдущей главе).

Теперь запустите утилиту **psql**:

```
psql –d postgres –U postgres
```

Если при установке СУБД вы назначили пароль для пользователя postgres, то вас попросят ввести его при запуске **psql**. Теперь вы может ввести команду

```
\?
```

и получить подсказку на русском языке.

Кроме скромной утилиты **psql** в комплект СУБД входит мощная утилита **pgAdmin III**, имеющая графический интерфейс пользователя. В частности, она имеет специальный модуль (интерфейс) для ввода SQL-команд (меню Tools–>Query tool). Рекомендуем вам самостоятельно изучить возможности утилиты **pgAdmin III** с помощью документации и метода проб и ошибок.

8.3. Язык SQL

Язык SQL – это непроцедурный язык, который является стандартным средством работы с данными во всех реляционных СУБД. Операторы (команды), написанные на этом языке, лишь указывают СУБД, какой результат должен быть получен, но не описывают процедуру получения этого результата. СУБД сама определяет способ выполнения команды пользователя.

Предлагаем вам кратко ознакомиться с языком SQL на практике. Для выполнения этой задачи сначала запустите утилиту **psql**:

```
psql –d test –U postgres
```

Для создания таблиц в языке SQL служит команда CREATE TABLE. Ее упрощенный синтаксис таков:

```
CREATE TABLE имя_таблицы
( имя_поля тип_данных [ограничения_целостности],
  имя_поля тип_данных [ограничения_целостности],
  ...
  имя_поля тип_данных [ограничения],
  [первичный_ключ]
);
```

В квадратных скобках показаны необязательные элементы команды. После команды нужно обязательно поставить символ «;».

Для получения полной информации о команде языка SQL введите команду

```
\h CREATE TABLE
```

Создайте таблицу «Студенты», описанную в начале этой главы. Таблица имеет следующую структуру (т. е. набор полей и их типы данных):

Имя поля	Тип данных	Тип данных PostgreSQL
Номер зачетной книжки	Числовой	numeric(5)
Ф. И. О.	Символьный	text
Серия паспорта	Числовой	numeric(4)
Номер паспорта	Числовой	numeric(6)

Число в описании типа данных numeric означает количество цифр, которые можно ввести в это поле. Тип данных text позволяет ввести значение, содержащее любые символы. Для поля «Серия паспорта» мы выбрали числовой тип, хотя более дальновидным был бы выбор символьного типа (см. задание номер 1 в конце главы).

```
CREATE TABLE students
( mark_book numeric( 5 ) NOT NULL,
  name text NOT NULL,
  psp_ser numeric( 4 ),
  psp_num numeric( 6 ),
  PRIMARY KEY ( mark_book )
);
```

Для СУБД регистр символов (прописные или строчные буквы) значения не имеет. Однако традиционно ключевые слова языка SQL вводят в верхнем регистре, что повышает наглядность SQL-операторов.

ПРИМЕЧАНИЕ. Эту команду (как и все SQL-команды в программе psql) можно вводить двумя способами. Первый способ заключается в построчном вводе команды точно так же, как она напечатана в тексте главы. При этом после ввода каждой строки нужно нажимать клавишу Enter. До тех пор, пока команда не введена полностью, вид приглашения к вводу команд, выводимого утилитой **psql**, будет отличаться от первоначального. В конце команды необходимо поставить точку с запятой.

Второй способ заключается в том, что команда вводится полностью на одной строке, при этом строка сворачивается «змейкой». Нажимать клавишу Enter после ввода каждого фрагмента команды не нужно, но можно для повышения наглядности вводить пробел. Второй способ удобнее тем, что он позволяет впоследствии с помощью клавиши «стрелка вверх» вызвать на экран всю команду полностью и при необходимости отредактировать ее либо выполнить еще раз без редактирования. Мы рекомендуем вам использовать именно этот способ.

В приведенной команде выражение NOT NULL означает, что не допускается ввод таких записей в таблицу students, у которых значение поля mark_book или name не указано. Выражение PRIMARY KEY (mark_book) означает, что поле mark_book будет служить первичным ключом таблицы students, т. е. значения этого поля будут уникальными идентификаторами записей в таблице.

Если вы ввели эту команду правильно, то система выдаст сообщение об успешном создании таблицы:

```
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index
"students_pkey" for table "students"
CREATE TABLE
```

Теперь введите несколько записей в созданную таблицу. Упрощенный формат команды таков:

```
INSERT INTO имя_таблицы
( имя_поля, имя_поля ... )
VALUES ( значение_поля, значение_поля, ... );
```

В нашем случае это будет выглядеть так:

```
INSERT INTO students ( mark_book, name, psp_ser, psp_num )
VALUES ( 12300, 'Иванов Иван Иванович', 0402, 543281 );
```

Обратите внимание на одинарные кавычки, в которые заключено значение поля name. Для символьных полей кавычки обязательны, а для числовых – нет. Введите 2–3 записи, изменяя значения полей. Вспомните о том, что можно редактировать ранее введенную команду, вызвав ее на экран при помощи клавиша «стрелка вверх».

Для выборки информации из таблиц базы данных служит такая команда:

```
SELECT имя_поля, имя_поля ...
FROM имя_таблицы;
```

Выберем всю информацию из таблицы students:

```
SELECT * FROM students;
```

Символ «*» означает выбор ВСЕХ полей каждой записи.

```
test=# SELECT * FROM students;
 mark_book |          name           | psp_ser | psp_num
-----------+-------------------------+---------+---------
     12300 | Иванов Иван Иванович    |     402 |  543281
     12302 | Сидоров Сидор Сидорович |     402 |  543381
     12301 | Петров Петр Петрович    |     202 |  543285
(3 rows)
```

Можно выбрать только фамилии, имена и отчества студентов:

```
SELECT name FROM students;
```

```
test=# SELECT name FROM students;
          name
------------------------
 Иванов Иван Иванович
 Сидоров Сидор Сидорович
 Петров Петр Петрович
(3 rows)
```

В том случае, когда вы вводите значения ВСЕХ полей, можно не перечислять их имена в первой части команды INSERT. Однако в этом случае нужно строго соблюдать порядок полей, т. е. тот порядок, в котором они были описаны при создании таблицы.

Введите еще 2–3 записи, изменяя значения полей, используя сокращенную форму команды INSERT:

```
INSERT INTO students
VALUES ( 12700, 'Климов Андрей Николаевич', 0204, 123281 );
```

Теперь введите еще 2–3 записи, изменяя значения полей, используя сокращенную форму команды INSERT, при которой вводятся значения НЕ ВСЕХ полей:

```
INSERT INTO students ( mark_book, name )
VALUES ( 13200, 'Павлов Павел Павлович' );
```

В этой команде используются лишь те поля, для которых указано ограничение NOT NULL в команде создания таблицы. Поля, для которых такое ограничение указано, должны ОБЯЗАТЕЛЬНО присутствовать в команде INSERT.

Попробуйте ввести такую запись:

```
INSERT INTO students ( mark_book, psp_ser, psp_num )
VALUES ( 13700, 0402, 432781 );
```

Вы получите сообщение об ошибке (внимательно изучите его):

```
ERROR:  null value in column "name" violates not-null con-
straint
```

Попробуйте ввести запись с таким значением поля «Номер зачетной книжки» (mark_book), которое вы уже вводили. Вы также получите сообщение об ошибке. Подумайте, почему оно появилось.

Для того чтобы выбрать из таблицы не все записи, а только те, которые удовлетворяют какому-либо условию, используется более сложная форма команды SELECT:

```
SELECT имя_поля, имя_поля ...
FROM имя_таблицы
WHERE условие;
```

Например, чтобы выбрать запись для студента с номером зачетной книжки 12300, выполните команду:

```
SELECT *
FROM students
WHERE mark_book = 12300;
```

Если Вы помните, то в таблице «Студенты» (students) есть несколько записей, у которых заполнены значения не всех полей. Для того чтобы эти значения заполнить, сначала нужно выявить такие записи. Для этого можно поступить следующим образом:

```
SELECT * FROM students ORDER BY name;
```

Предложение ORDER BY служит для сортировки записей. С помощью данной команды вы не только выберете записи из таблицы, но также упорядочите их по значениям поля «Ф. И. О.» (name). Вы увидите все записи из таблицы students и среди них – искомые записи с пустыми значениями полей psp_ser и psp_num. Однако, в том случае, когда записей в таблице много, такой способ не очень удобен. Лучше поступить так:

```
SELECT * FROM students WHERE psp_ser is NULL;
```

Обратите внимание на выражение: psp_ser is NULL. Оно принципиально отличается от выражения: psp_ser = '' (здесь введены две одинарные кавычки). Вспомните, что NULL (null) означает отсутствие какого-либо значения вообще, в том числе и пустой строки.

Теперь обновите записи, в которых не указаны паспортные данные. Для этого воспользуйтесь командой:

```
UPDATE имя_таблицы
SET имя_поля=значение, имя_поля=значение, ...
WHERE условие;
```

Условие, указываемое в команде, должно ограничить диапазон обновляемых записей. Для того чтобы ввести паспортные данные для студента с номером зачетной книжки, например, 12300, нужно выполнить такую команду:

```
UPDATE students
SET psp_ser=0402, psp_num=123456
WHERE mark_book=12300;
```

Команда удаления записей похожа на команду выборки:

```
DELETE FROM имя_таблицы
WHERE условие
```

Удалите какую-нибудь одну запись из таблицы «Студенты» (students):

```
DELETE FROM students
WHERE mark_book = 12300;
```

Вы можете указать и какое-нибудь более сложное условие, например:

```
DELETE FROM students
WHERE mark_book >= 12300 and mark_book <= 12500;
```

или

```
DELETE FROM students
WHERE name <> 'Иванов Иван Иванович' and mark_book <= 12500;
```

В этой команде символы «<>» означают, что значения поля name в удаляемых записях должны быть не равны указанному значению.

ПРИМЕЧАНИЕ. Если вы удалили слишком много записей, восстановите их, используя возможность редактировать ранее введенные команды (клавиша «стрелка вверх»).

Теперь создайте ту таблицу «Успеваемость», которая была описана в начале главы. Структура ее такова:

Имя поля	Тип данных	Тип данных PostgreSQL
Номер зачетной книжки	Числовой	numeric(5)
Предмет	Символьный	text
Учебный год	Символьный	text
Семестр	Числовой	numeric(1)
Оценка	Числовой	numeric(1)

```
CREATE TABLE progress
( mark_book numeric( 5 ) NOT NULL,
  subject text NOT NULL,
  acad_year text NOT NULL,
  term numeric( 1 ) NOT NULL CHECK ( term = 1 or term = 2 ),
  mark numeric( 1 ) NOT NULL CHECK ( mark >= 3 and mark <= 5 )
DEFAULT 5,
  FOREIGN KEY ( mark_book )
  REFERENCES students ( mark_book )
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
```

Команда для создания этой таблицы более сложная, чем предыдущая. Дополнительные ограничения CHECK позволяют ограничить допустимые значения полей term и mark. Предложение DEFAULT позволяет указать значение по умолчанию для поля mark. Предложение FOREIGN KEY создает ограничение ссылочной целостности и указывает в качестве ссылочного ключа поле mark_book. Это означает, что в таблицу «Успеваемость» (progress) нельзя ввести запись, значение поля mark_book которой отсутствует в таблице «Студенты» (students). Говоря простым языком, нельзя ввести запись об оценке того студента, информация о котором еще не введена в таблицу «Студенты». При удалении записи из таблицы «Студенты» (students) будут также удалены записи из таблицы «Успеваемость» (progress), у которых значение поля mark_book совпадает со значением этого поля в удаляемой записи в таблице «Студенты». Таким образом, при удалении информации о студенте удаляется и информация обо всех его оценках. Если же мы изменим номер зачетной книжки в таблице «Студенты», то СУБД сама изменит этот номер во всех записях об оценках для данного студента.

Для того чтобы посмотреть, какая получилась таблица, введите команду

```
\d progress
```

Это не команда языка SQL, а команда утилиты **psql**. Она служит для вывода информации о структурах таблиц.

Введите несколько записей в таблицу «Успеваемость» с помощью команды INSERT. Можно воспользоваться такой формой команды, в которой не указываются имена полей, но в этом случае нужно задать значения всех полей в том порядке, в котором они перечислены в команде создания таблицы CREATE TABLE.

```
INSERT INTO progress VALUES ( 12500, 'Физика', '2000/2001', 1,
4 );
```

Введите несколько записей (по 2–3 для каждого из студентов, перечисленных в таблице «Студенты»), изменяя соответственно номер зачетной книжки, предмет, семестр и оценку.

Попробуйте ввести предшествующую команду, сократив ее немного:

```
INSERT INTO progress VALUES ( 12700, 'Математика',
'2000/2001', 1 );
```

Отличие в том, что для поля «Оценка» (mark) значение не указывается. Это допустимо, т. к. данное поле идет в списке последним. А поскольку для него предусмотрено значение по умолчанию (5), то именно это значение и будет записываться в базу данных, если вы не укажете значение явно. Если бы такое поле (для которого задано значение по умолчанию) было не последним в списке, то нам пришлось бы тогда использовать полную форму команды INSERT:

```
INSERT INTO имя_таблицы
( имя_поля, имя_поля ... )
VALUES ( значение_поля, значение_поля, ... );
```

Но в списке имен полей и в списке значений полей мы могли бы пропустить то поле, которое имеет значение по умолчанию.

Попробуйте ввести в таблицу «Успеваемость» (progress) запись, у которой значение поля «Номер зачетной книжки» (mark_book) такое, которого нет в таблице «Студенты» (students). Вы получите сообщение об ошибке, которое демонстрирует работу правил ссылочной целостности:

```
ERROR:  insert or update on table "progress" violates foreign
key constraint "progress_mark_book_fkey"
DETAIL:  Key (mark_book)=(99900) is not present in table "stu-
dents".
```

Теперь решим более сложную задачу. Предположим, что нам нужно получить экзаменационную ведомость по физике (или другому предмету)

за первый семестр 2000/2001 учебного года. Для этого мы выполняем такую команду:

```
SELECT * FROM students, progress
WHERE progress.acad_year = '2000/2001' and
      progress.term = 1 and
      progress.subject = 'Физика' and
      students.mark_book = progress.mark_book;
```

Обратите внимание на наличие имен двух таблиц в предложении FROM, а также на строку students.mark_book = progress.mark_book в предложении WHERE. Данная строка (условие) позволяет выполнить так называемое **соединение** таблиц. При этом формируются объединенные записи из тех записей двух таблиц, у которых значения поля mark_book одинаковы.

В экзаменационных ведомостях не требуется наличия паспортных данных. В них можно также не указывать для каждого студента повторяющееся наименование предмета, поэтому мы можем указать в команде только часть полей. Отсортируем нашу ведомость по фамилии, имени и отчеству студентов.

```
SELECT progress.mark_book, students.name, progress.mark
FROM students, progress
WHERE progress.acad_year = '2000/2001' and
      progress.term = 1 and
      progress.subject = 'Физика' and
      students.mark_book = progress.mark_book
ORDER BY students.name;
```

А теперь мы можем выбрать все оценки для одного студента за весь период обучения:

```
SELECT *
FROM progress
WHERE mark_book = ( SELECT mark_book
                    FROM students
                    WHERE name = 'Иванов Иван Иванович')
ORDER BY acad_year, term, subject;
```

Обратите внимание, что в приведенной команде не указываются имена таблиц перед именами полей. Это допустимо, когда СУБД сможет однозначно определить, из какой таблицы выбирать конкретное поле. Обратите также внимание на наличие двух предложений SELECT в одной команде. Второе предложение SELECT называется **подзапросом**. Этот подзапрос дает значение поля mark_book для того студента, значение поля «Ф. И. О.» (name) которого равно 'Иванов Иван Иванович'.

Для проверки работы правил ссылочной целостности выполните обновление поля mark_book в одной из записей таблицы students:

```
UPDATE students
SET mark_book = 12900
WHERE mark_book = 12300;
```

Сделайте выборку записей из таблицы students и убедитесь в том, что обновление произошло. Затем сделайте выборку записей из обеих таблиц, как вы делали это ранее, и убедитесь, что и в таблице progress обновление также произошло. Это было выполнено благодаря работе правил ссылочной целостности. Причем, данные операции были произведены в рамках одной **транзакции**. Если бы в процессе их выполнения произошел сбой электропитания, то в базе данных все записи остались бы в том состоянии, в котором они находились до начала выполнения вашей команды UPDATE.

Выполните команду удаления какой-нибудь записи из таблицы students:

```
DELETE FROM students
WHERE mark_book = 12300;
```

Сделайте выборку записей из обеих таблиц, как вы делали это ранее, и убедитесь, что и из таблицы progress записи были также удалены.

Можно сделать и общую выборку из таблицы progress и визуально убедиться в том, что удаление записей действительно имело место:

```
SELECT * FROM progress;
```

Для выхода из программы **psql** наберите команду **\q** и нажмите Enter.

В операционной системе Windows

В среде Windows все представленные SQL-команды будут работать аналогично. При этом вы можете выполнять их не только с помощью утилиты **psql**, но также и с помощью утилиты **pgAdmin III**.

8.4. Библиотека libpq

Прикладные программы, предназначенные для работы с базами данных, должны иметь возможность каким-то образом обращаться к функциям СУБД. Для реализации такой возможности существуют различные технологии, предусматривающие использование различных языков программирования. В состав установочного комплекта СУБД PostgreSQL входит

специальная библиотека **libpq**, которая предназначена для организации взаимодействия прикладной программы, написанной на языке C, с базой данных под управлением СУБД PostgreSQL.

Мы покажем только самые простые примеры использования этой библиотеки. Если вы следовали нашим рекомендациям по установке СУБД, то в каталоге **/usr/src/postgresql-8.x.x/src/test/examples** должны находиться примеры использования библиотеки **libpq**.

Чтобы скомпилировать примеры программ, находящихся в этом каталоге, вы можете поступить одним из двух способов. Первый способ является более простым в исполнении, но менее информативным для вас. Для его реализации проделайте следующее:

```
cd /usr/src/postgresql-8.x.x/src/test/examples
gmake
```

В результате вы получите четыре скомпилированные программы.

Второй метод более трудоемок, но он, на наш взгляд, даст вам более полное представление о том, как подключить библиотеку **libpq** на этапе компиляции и компоновки прикладной программы. Мы продемонстрируем его на примере одной из программ. Скопируйте файл **testlibpq.c** в какой-нибудь из каталогов, в котором можно поэкспериментировать с программой. Скомпилируйте эту программу (команду вводите в одну строку):

```
gcc -o testlibpq testlibpq.c -I/usr/local/pgsql/include
-L/usr/local/pgsql/lib -lpq
```

Проверьте, от каких разделяемых библиотек зависит полученный исполняемый модуль:

```
ldd testlibpq

testlibpq:
        libpq.so.5 => not found (0x0)
        libc.so.6 => /lib/libc.so.6 (0x2807a000)
```

Оказывается, при запуске этой программы разделяемая библиотека **libpq.so.5** не будет найдена и загружена (вы можете это проверить). Чтобы исправить ситуацию, модифицируйте команду компиляции:

```
gcc -o testlibpq testlibpq.c -I/usr/local/pgsql/include
-L/usr/local/pgsql/lib -lpq -Wl,-rpath=/usr/local/pgsql/lib
```

Обратите внимание, что в параметре **–Wl,–rpath=/usr/local/pgsql/lib** не должно быть пробелов.

Команда **ldd** подтвердит, что теперь все в порядке:

```
ldd testlibpq

testlibpq:
        libpq.so.5 => /usr/local/pgsql/lib/libpq.so.5
(0x2807a000)
        libc.so.6 => /lib/libc.so.6 (0x28098000)
        libcrypt.so.3 => /lib/libcrypt.so.3 (0x2817d000)
```

При запуске программы **testlibpq** необходимо передать ей в качестве параметра имя базы данных и имя пользователя (обратите внимание, что они передаются в виде одной строки):

```
./testlibpq "dbname=test user=postgres"
```

На экран будет выведен список баз данных, присутствующих в кластере, которым управляет СУБД PostgreSQL. Для получения этих сведений программа обращается к системной таблице pg_database, входящей в состав словаря данных СУБД.

Обратите внимание на очень маленький размер файла **testlibpq**. Как вы уже знаете, это достигается за счет того, что на этапе компоновки программы объектный код из библиотек не включается в исполняемый модуль. Иногда бывает необходимо пойти на увеличение размера исполняемого модуля и включить в него весь необходимый ему объектный код. Такой способ называется статической компоновкой и реализуется путем включения параметра **–static** в команду компиляции программы:

```
gcc -o testlibpq_st testlibpq.c -I/usr/local/pgsql/include
-L/usr/local/pgsql/lib -lpq -lcrypt -static
```

Если с помощью программы **ldd** проверить зависимость полученного исполняемого модуля от динамических библиотек, то он сообщит следующее:

```
ldd testlibpq_st

ldd: testlibpq_st: not a dynamic executable
```

Но при этом размер исполняемого модуля стал на два порядка больше.

Давайте модифицируем эту программу-пример и «поручим» ей выбирать информацию из таблицы students, которую вы создали при ознакомлении с языком SQL.

Программа **test.c**

```
/*
 * Программа: test.c
```

```
 * Проверка возможности доступа к базе данных с помощью
 * библиотеки libpq
 */

#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void exit_nicely( PGconn *conn )
{
  PQfinish( conn );
  exit( 1 );
}

int main( int argc, char **argv )
{
  const char *conninfo; /* параметры для соединения с базой
                           данных */
  PGconn     *conn;     /* дескриптор соединения с базой
                           данных */
  PGresult   *res;      /* результат выполнения SQL-запроса */
  int        nFields;   /* число полей в выборке */
  int        i, j;

  /*
   * если пользователь передает параметр в командной строке,
   * то нужно использовать его в качестве строки conninfo;
   * если параметр в командной строке не передан,
   * то по умолчанию для подключения к базе данных
   * используется строка "dbname=test user=postgres"
  */
  if ( argc > 1 )
    conninfo = argv[ 1 ];
  else
    conninfo = "dbname=test user=postgres";

  /* подключаемся к базе данных */
  conn = PQconnectdb( conninfo );

  /* проверяем успешность подключения */
  if ( PQstatus( conn ) != CONNECTION_OK )
  {
    fprintf( stderr,
             "Подключиться к базе данных не удалось: %s",
             PQerrorMessage( conn ) );
    exit_nicely( conn );
  }

  /* функция PQexec исполняет SQL-запрос, переданный ей
     в виде символьной строки */
  res = PQexec( conn, "SELECT * FROM Students" );
  if (PQresultStatus(res) != PGRES_TUPLES_OK)
  {
```

```
    fprintf( stderr, "Запрос SELECT неудачен: %s",
             PQerrorMessage( conn ) );
    PQclear( res );
    exit_nicely( conn );
  }

  nFields = PQnfields( res ); /* количество полей в выборке */

  /* сначала выведем имена полей для полученной выборки
     (фактически это имена полей таблицы Students) */
  for ( j = 0; j < nFields; j++ )
  {
    /* для поля "Фамилия, имя, отчество" выделим больше места
       (функция PQfname возвращает имя поля по его порядковому
       номеру в выборке) */
    if ( strcmp( PQfname( res, j ), "name" ) == 0 )
      printf( "%-30s", PQfname( res, j ) );
    else
      printf( "%-10s", PQfname( res, j ) );
  }

  printf( "\n\n" );

  /* выведем данные из полученной выборки
     (функция PQntuples возвращает число строк в выборке) */
  for ( i = 0; i < PQntuples( res ); i++ )
  {
    for ( j = 0; j < nFields; j++ )
    {
      if ( strcmp( PQfname( res, j ), "name" ) == 0 )
        /* функция PQgetvalue возвращает значение поля номер i
           из строки номер j */
        printf( "%-30s", PQgetvalue( res, i, j ) );
      else
        printf( "%-10s", PQgetvalue( res, i, j ) );
    }
    printf( "\n" );
  }

  /* освободим память, которую занимала структура res,
     как только эта структура больше не нужна */
  PQclear( res );

  /* закроем соединение с базой данных */
  PQfinish( conn );

  return 0;
}
```

Обратите внимание на директиву #include "libpq-fe.h". Этот файл необходим для использования функций из библиотеки **libpq**.

Откомпилируйте эту программу:

```
gcc -o test test.c -I/usr/local/pgsql/include
-L/usr/local/pgsql/lib -lpq -Wl,-rpath=/usr/local/pgsql/lib
```

При запуске программы test необходимо передать ей в качестве параметра имя базы данных и имя пользователя (обратите внимание, что они передаются в виде одной строки):

```
./test "dbname=test user=postgres"
```

На экран будет выведен список студентов из таблицы students.

Получить подробные инструкции по использованию библиотеки **libpq** можно в документации, поставляемой в комплекте с СУБД.

В операционной системе Windows

Поскольку в среде Windows вы устанавливали СУБД не из исходных текстов, то каталога с примерами в этом установочном комплекте нет. Возьмите эти примеры из того комплекта, которым вы пользовались в среде FreeBSD. Работа в среде Windows (MSYS) не имеет принципиальных отличий, однако некоторые команды необходимо модифицировать. Не забывайте также запускать сервер баз данных PostgreSQL. В системе Windows это можно сделать двумя способами, о чем шла речь при установке СУБД.

Начнем с примера **testlibpq.c**. Команда компиляции выглядит так:

```
gcc -o testlibpq testlibpq.c -I/c/PostgreSQL/include
-L/c/PostgreSQL/lib -lpq
```

Здесь вместо традиционной формы обозначения идентификатора логического диска **C:** используется **/c**, как это принято в среде MSYS. Вместо пути **/c/PostgreSQL** укажите тот путь, где установлена СУБД на вашем компьютере. Для запуска программы введите

```
./testlibpq.exe "dbname=test user=postgres password=*****"
```

Вместо звездочек введите пароль, который вы назначили пользователю postgres при установке СУБД.

Теперь скомпилируйте вторую программу – test.c:

```
gcc -o test test.c -I/c/PostgreSQL/include
-L/c/PostgreSQL/lib -lpq
```

Запустите ее также с параметром:

```
./test.exe "dbname=test user=postgres password=*****"
```

8.5. Язык Perl и СУБД PostgreSQL

Организовать взаимодействие с базой данных под управлением СУБД PostgreSQL можно и из программы на языке Perl. Самый простой способ выполнения этой задачи – установить модуль (пакет) **Pg.pm**. Получить его можно по адресу http://gborg.postgresql.org/project/pgperl/projdisplay.php.

Процедура установки пакета такова.

1. Войдите в систему под именем postgres. Поскольку все операции по установке пакета **Pg.pm** удобнее выполнять непосредственно из среды командного процессора, то не запускайте файловый менеджер (такой, как, например, **deco**).

2. Скопируйте архивный файл **Pg-2.1.1.tar.gz** в домашний каталог пользователя postgres и извлеките файлы из архива:

```
cp Pg-2.1.1.tar.gz /home/postgres
cd /home/postgres
tar xzvf Pg-2.1.1.tar.gz
cd Pg-2.1.1
```

3. Назначьте переменную среды POSTGRES_HOME:

```
POSTGRES_HOME=/usr/local/pgsql; export POSTGRES_HOME
```

Эти две команды нужно ввести в одной строке и нажать клавишу Enter. Не забудьте, что эту операцию нужно выполнять не из среды файлового менеджера, а непосредственно из среды операционной системы.

4. Убедитесь, что текущим каталогом является тот, в котором находятся установочные файлы пакета **Pg.pm**, и выполните команду (не забудьте, что в системе UNIX регистр символов в именах файлов имеет значение)

```
perl Makefile.PL
```

На экран будет выведено следующее:

```
Configuring Pg
Remember to actually read the README file !
OS: freebsd
Checking if your kit is complete...
Looks good
Writing Makefile for Pg
```

5. Выполните команду

```
make
```

6. Назначьте следующие переменные среды:

```
PGDATABASE=test;  export PGDATABASE
PGUSER=postgres;  export PGUSER
```

7. Протестируйте сформированный пакет:

```
make test
```

На экран будет выведен ряд сообщений:

```
PERL_DL_NONLAZY=1 /usr/bin/perl "-Iblib/lib" "-Iblib/arch"
test.pl
Pg::conndefaults ........ ok
Pg::connectdb ........... ok
$conn->exec ............. ok
.........
$result->fnumber ........ ok
$result->fetchrow ....... ok
test sequence finished.
```

8. Если на предыдущем шаге не было выведено сообщений об ошибках, то войдите в систему под именем root, перейдите в каталог, где находятся установочные файлы пакета **Pg.pm**, и выполните установку пакета в системные каталоги:

```
cd /home/postgres/Pg-2.1.1
make install
```

Для получения подробных инструкций по использованию пакета обратитесь к электронной документации:

```
man Pg
```

Рекомендуем вам также внимательно просмотреть исходные тексты программы **test.pl** и программ в подкаталоге **examples**.

Рассмотрим программу, которая выбирает все строки из таблицы students. Эта программа аналогична программе **test.c**, рассмотренной в предыдущем параграфе.

Программа **test_pq.pl**

```
#!/usr/bin/perl -w
# ------------------------------------------------------------
# Программа test_pq.pl
# Проверка возможности доступа к базе данных с помощью
# пакета Pg.pm
# ------------------------------------------------------------
```

```
use strict;
use Pg;                  # пакет (модуль) для работы
                         # с PostgreSQL
use constant DBNAME => 'test';   # имя базы данных

my $conn;                # дескриптор соединения с базой данных
my $result;              # результат выполнения SQL-запроса
my @row;                 # массив для хранения одной записи
                         # из выборки
my $command;             # строка SQL-запроса для работы с БД

# подключение к базе данных
$conn = Pg::connectdb( "dbname=" . DBNAME . " user=postgres"
                     );

# проверка успешности подключения
die $conn->errorMessage
  unless PGRES_CONNECTION_OK eq $conn->status;

# формирование SQL-запроса в виде символьной строки:
# выбрать все записи из таблицы Students
# (упорядочим записи по значениям поля name)
$command = "SELECT * FROM students " .
           "ORDER BY name";

# выполнение запроса к базе данных
$result = $conn->exec( $command );

# если произошла ошибка при выполнении запроса к базе данных
if ( PGRES_TUPLES_OK ne $result->resultStatus )
{
  die $conn->errorMessage;  # завершить работу
}

# если в выборке нет ни одной строки
if ( $result->ntuples == 0 )
{
  print "В таблице Students нет записей\n";
  return ( 0 );
}

# выведем заголовки
printf "%-12s", "Зач. книжка";
printf "%-30s", "Ф. И. О.";
printf "%-12s", "Серия пасп.";
printf "%-12s", "Номер пасп.";
print "\n\n";

# выводим все строки из полученной выборки
# метод fetchrow считывает в массив очередную строку
# из выборки
while ( @row = $result->fetchrow )
```

```
{
  # метод fnumber возвращает номер поля в выборке,
  # т. е. в массиве, содержащем текущую строку выборки
  # (поэтому при необходимости можно вывести значения
  # полей на экран не в том порядке, в котором эти поля
  # представлены в структуре реляционной таблицы
  # в базе данных)
  printf "%-12s", $row[ $result->fnumber( 'mark_book' ) ];
  printf "%-30s", $row[ $result->fnumber( 'name' ) ];
  printf "%-12s", $row[ $result->fnumber( 'psp_ser' ) ];
  printf "%-12s", $row[ $result->fnumber( 'psp_num' ) ];
  print "\n";
}
exit ( 0 );
```

Выполнить эту программу можно таким образом:

```
perl ./test_pq.pl
```

Возможен и такой подход:

```
chmod 755 test_pq.pl
./test_pq.pl
```

В операционной системе Windows

В среде Windows (MSYS) установить пакет **Pg.pm**, используя указания автора пакета, не удается.

Контрольные вопросы и задания

1. Подумайте над возможной заменой числового типа данных, выбранного нами для поля «Серия паспорта» в таблице «Студенты», на символьный тип. Обратите внимание на то, что при вводе, например, серии «0402» первый ноль не хранится в базе данных.

2. Модифицируйте базу данных test. Создайте еще одну таблицу «Учебные дисциплины». Эта таблица должна содержать два поля: «Код учебной дисциплины» (числовой тип данных) и «Наименование учебной дисциплины» (символьный тип данных). В таблице «Успеваемость» замените поле «Предмет» полем «Код учебной дисциплины». При выводе информации из таблицы «Успеваемость» используйте соединение этой таблицы с таблицей «Учебные дисциплины» в команде SELECT, чтобы вместо числовых кодов на экран выводились наименования учебных дисциплин (предметов).

3. Модифицируйте программу **test.c**, например, реализуйте операции ввода новых записей в таблицу «Студенты» (students) и удаления существующих записей из этой таблицы.

4. Изучите остальные программы из набора тестовых программ, входящих в комплект поставки СУБД PostgreSQL.

5. Модифицируйте программу **test_pq.pl**, например, реализуйте возможность ввода записей в таблицу «Успеваемость» (progress). При этом сделайте так, чтобы все сведения об экзаменационных оценках программа «умела» читать из списка, хранящегося в текстовом файле.

6. Загрузите модули DBI и DBD с сайта http://www.cpan.org и самостоятельно разберитесь с правилами их использования.

9. Интернет-технологии

Интернет-технологии изменили жизнь многих людей в последние годы. Электронная почта значительно потеснила почту традиционную. Люди все чаще обращаются к услугам всемирной сети с самыми разными вопросами: профессиональными, бытовыми и даже личными. Поэтому нам представляется важным включить в пособие хотя бы простые примеры создания Интернет-приложений.

9.1. Установка web-сервера Apache

Web-сервер – это программно-аппаратный комплекс, который управляет процессом предоставления информации пользователям, обращающимся к услугам web-сервера с помощью браузеров. Такой сервер предоставляет не только статические HTML-документы, хранящиеся на нем в виде текстовых файлов в формате HTML, но и документы (файлы) других форматов. Он может также запускать программы, которые взаимодействуют с базой данных и формируют HTML-документы динамически, т. е. на основе содержимого базы данных.

Часто термин web-сервер понимают только как программный комплекс. В мире используются различные программные продукты этого класса, но одним из самых популярных является Apache.

Загрузить исходные тексты или скомпилированные модули этого web-сервера можно по адресу http://www.apache.org. Найдите последнюю стабильную версию и загрузите ее на свой компьютер. Так как номера версий постоянно изменяются, то мы, как и прежде, ту часть номера, которая подвержена наиболее частым изменениям, заменим символами «x».

Поскольку процесс установки программных продуктов в операционной системе FreeBSD вам уже знаком, ограничимся лишь краткой инструкцией.

1. Войдите в систему под именем пользователя root, извлеките файлы из архива и перейдите в созданный подкаталог **httpd-2.x.x**:

```
tar xzvf httpd-2.x.x.tar.gz
cd httpd-2.x.x
```

2. Запустите утилиту конфигурирования исходных текстов:

```
./configure > conf_log.txt 2>&1 &
```

3. Скомпилируйте программы:

```
make > make_log.txt 2>&1 &
```

4. Установите программы (по умолчанию они будут установлены в каталог **/usr/local/apache2**):

```
make install > make_install_log.txt 2>&1 &
```

5. Запустите web-сервер:

```
/usr/local/apache2/bin/apachectl start
```

Можно это сделать и таким образом:

```
cd /usr/local/apache2/bin
./apachectl start
```

ПРИМЕЧАНИЕ. Будут выведены предупреждающие сообщения, но web-сервер должен, тем не менее, успешно запуститься.

Чтобы web-сервер запускался при загрузке компьютера, можно включить эту команду в файл **/etc/rc.local**, аналогично тому, как вы поступали при установке СУБД PostgreSQL.

Для проверки выполнения запуска программы введите:

```
ps –ax
```

Если на экране вы увидите примерно следующие сообщения, то все в порядке:

```
26556  ??  Ss   0:00,00 /usr/local/apache2/bin/httpd -k start
26557  ??  I    0:00,00 /usr/local/apache2/bin/httpd -k start
26558  ??  I    0:00,00 /usr/local/apache2/bin/httpd -k start
26559  ??  I    0:00,00 /usr/local/apache2/bin/httpd -k start
26560  ??  I    0:00,00 /usr/local/apache2/bin/httpd -k start
26561  ??  I    0:00,00 /usr/local/apache2/bin/httpd -k start
```

6. Протестируйте работу web-сервера:

```
lynx http://localhost
```

На экран должно быть выведено лишь короткое сообщение:

```
"It works!"
```

ПРИМЕЧАНИЕ. Если вы еще не установили браузер **lynx**, то можете установить его с помощью утилиты **sysinstall** с установочных дисков FreeBSD.

Чтобы остановить работу web-сервера, выполните команду (как пользователь root)

```
/usr/local/apache2/bin/apachectl stop
```

или

```
cd /usr/local/apache2/bin
./apachectl stop
```

Для получения помощи используйте электронные руководства:

```
man -M /usr/local/apache2/man httpd
```

При установке программного продукта будет установлена подробная документация в каталог **/usr/local/apache2/manual**. Чтобы сделать ее доступной, необходимо модифицировать конфигурационный файл **httpd.conf**. Выполните команды

```
cd /usr/local/apache2/conf
cp httpd.conf httpd.conf.orig
```

Удалите символ # в начале строки (т. е. раскомментируйте строку):

```
"Include conf/extra/httpd-manual.conf"
```

После внесения изменений в конфигурационный файл необходимо «сказать» web-серверу, чтобы он заново перечитал файл **httpd.conf**. Это можно сделать, послав web-серверу сигнал с помощью команды **kill** следующим образом (обратите внимание на обратные кавычки):

```
kill -1 `cat /usr/local/apache2/logs/httpd.pid`
```

Теперь проверьте, доступна ли документация:

```
lynx http://localhost/manual
```

В операционной системе Windows

Загрузите с сайта http://www.apache.org инсталляционный файл в формате MSI. При установке web-сервера введите в предложенном окне диалога адрес электронной почты администратора и имя сервера. Если предполагается использование web-сервера только в локальном режиме, тогда можно ввести вымышленный адрес электронной почты, а в качестве имени сервера введите IP-адрес 127.0.0.1.

После установки web-сервера и его запуска можно протестировать его работу так же, как и в системе FreeBSD. Таким же образом можно сделать доступной и документацию, находящуюся в подкаталоге manual. После внесения изменений в конфигурационный файл **httpd.conf** следует перезапустить web-сервер. Утилиту для перезапуска Restart можно найти в

группе Apache HTTP Server в меню Windows, вызываемом нажатием кнопки «Пуск».

9.2. Создание простой CGI-программы

Common Gateway Interface (CGI) – это протокол, определяющий правила взаимодействия web-сервера и программы, при выполнении которой генерируется HTML-документ, передаваемый браузеру пользователя для визуализации.

Предлагаем вам пример простой CGI-программы. Но сначала создадим HTML-документ, из которого эта программа вызывается. Мы не поясняем назначение основных тегов языка разметки HTML, т. к. их применение понятно из контекста. К тому же, язык HTML не очень сложен в изучении, поэтому каждый может ознакомиться с ним самостоятельно.

Файл **test.html**

```
<HTML>
<HEAD>
<TITLE>
Запрос информации с помощью формы
</TITLE>

<BODY BGCOLOR = "#FFFF00">
<H1>Это простая форма для запроса информации
 у пользователя</H1>

<FORM ACTION = "/cgi-bin/cgi.pl" METHOD = "POST">
Введите ваше имя: <INPUT TYPE = "text" NAME = "fname"
                    SIZE = 20> <BR>
Введите вашу фамилию: <INPUT TYPE = "text" NAME = "lname"
                         SIZE = 30> <BR>
<INPUT TYPE = "submit" NAME = "add_student" VALUE =
"Записать">
<INPUT TYPE = "submit" NAME = "show_db" VALUE = "Показать базу
данных">
</FORM>

</BODY>
</HTML>
```

Обратите внимание на то, как указан путь к CGI-программе в теге FORM ACTION. Он выглядит, как абсолютный путь, который начинается с корневого каталога. Однако в качестве корневого каталога в данном случае выступает тот каталог, в который установлен web-сервер Apache.

А теперь создадим текст CGI-программы на языке Perl. В этой программе много комментариев, поясняющих различные принципиально важ-

ные вопросы, поэтому рекомендуем вам не пренебрегать вдумчивым чтением комментариев.

Программа **cgi.pl**

```
#!/usr/bin/perl -w
# ------------------------------------------------------------
# Программа для обработки данных, введенных в поля формы
# ------------------------------------------------------------

# переменная для временного хранения всей информации,
# введенной в поля формы, в виде единой строки
my $form_info;

# общее количество информации, переданной от формы к программе
# ПРИМЕЧАНИЕ. Здесь используется хеш-массив %ENV, который
#             создается самим Perl'ом для сбора информации
#             о так называемых переменных окружения
#             (по-английски -- environment). В элементе
#             с ключом 'CONTENT_LENGTH' записано количество
#             байтов информации, переданной этой программе
#             при ее запуске по команде браузера
#             (см. тег FORM ACTION). В этой переданной
#             информации и содержатся имена полей формы
#             и значения, введенные пользователем в поля
#             формы.
my $info_len = $ENV{ 'CONTENT_LENGTH' };

# массив для хранения параметров, переданных этой программе
my @params = ();

my $i;

# хеш-массив для размещения в нем имен параметров формы
# и значений параметров; ключом является имя параметра,
# а хранимым элементом - значение, введенное в окно формы
my %prms = ();

# имя параметра и его значение
my( $param_name, $param_value );

# это так называемый HTTP-заголовок
# ПРИМЕЧАНИЕ. HTTP - это протокол, т. е. набор правил,
#             по которым "общаются" браузер пользователя
#             и web-сервер. Наш заголовок очень простой:
#             он указывает браузеру, что тот текст, который
#             браузер получит от web-сервера, браузер должен
#             истолковать как HTML-документ.
# ВАЖНО! Обратите внимание на то, что в конце строки указаны
#        ДВА символа новой строки. Это сделано потому, что
#        согласно стандарту HTTP, заголовок должен отделяться
#        от тела документа пустой строкой.
print "Content-type: text/html\n\n";
```

```
# эти команды формируют обычные для HTML-документа теги
print "<HTML>\n";
print "<HEAD>\n";
print "<TITLE>CGI-program</TITLE>\n";
print "</HEAD>\n";

# выберем белый цвет фона
# ПРИМЕЧАНИЕ. Из шести шестнадцатеричных цифр первые две
#             соответствуют яркости красного цвета, следующие
#             две - зеленого, последние - синего. Значения
#             цифр - от 0 до F. Можно поэкспериментировать.
# ПРИМЕЧАНИЕ. Обратите внимание на наличие символов "\"
#             перед кавычками. Это называется
#             "экранированием". Экранирование в данном случае
#             выполняется в соответствии с требованиями языка
#             Perl, а браузер получит документ уже без этих
#             символов "\".
print "<BODY BGCOLOR=\"#FFFFFF\">\n";

# читаем со стандартного ввода столько байтов, сколько их
# передано этой программе, и записываем все считанные байты
# в строку $form_info
read( STDIN, $form_info, $info_len );

# проводим необходимые преобразования считанной информации,
# которые нужны потому, что при передаче данных от формы
# к программе происходит замена всех символов, кроме букв
# латинского алфавита, на их шестнадцатеричное представление
# с добавлением символа "%" перед каждой парой
# шестнадцатеричных цифр
# ПРИМЕЧАНИЕ. Если бы мы вводили в поля формы только
#             английские имена, то это преобразование не было
#             бы нужно.
# ПРИМЕЧАНИЕ. Это регулярное выражение не нужно стараться
#             понять полностью, пока достаточно только знать
#             его назначение. Это выражение можно найти
#             в любом учебнике по CGI-программированию и там
#             прочесть подробное объяснение того, что делает
#             функция pack().
$form_info =~
  s/%([\dA-Fa-f][\dA-Fa-f])/pack( "C", hex( $1 ) )/eg;

# при передаче информации в программу формируется строка
# такого вида: fname=Иван&lname=Петров&add_student=.......
# т. е. она состоит из пар: имя_параметра=значение_параметра,
# разделенных символом & (амперсанд). Наша задача - разделить
# строку сначала на такие пары, используя в качестве
# разделителя символы &, и записать их в массив, что и делает
# функция split(). Выражение /&/ указывает символ-разделитель.
@params = split( /&/, $form_info );

# а теперь мы сформируем хеш-массив, в который поместим
```

```
# значения, введенные в поля формы; ключами же этих значений
# будут имена полей-параметров
foreach ( @params )
{
  # опять используем функцию split(), но разделителем уже
  # будет символ "=", поскольку имя и значение параметра
  # разделены именно этим символом
  # ПРИМЕЧАНИЕ. В левой части выражения помещены сразу две
  #             переменные в скобках: это замечательное
  #             свойство языка Perl, которое позволяет
  #             присвоить значения сразу двум и более
  #             переменным. В данном случае функция split()
  #             дает два значения, которые и заносятся
  #             в переменные. Переменная $_, как вы уже
  #             знаете, хранит значение текущего элемента
  #             массива @params.
  ( $param_name, $param_value ) = split( /=/, $_ );

  # запишем значение $param_value в хеш-массив по ключу
  # $param_name
  # ПРИМЕЧАНИЕ. Хеш-массивы особенно удобны, когда параметров
  #             много.
  $prms{ $param_name } = $param_value;
}

# поскольку наша форма позволяет не только ввести новую запись
# в базу данных, но и просмотреть базу данных, нужно выполнять
# проверку того, какая из кнопок была нажата. В зависимости от
# того, какую кнопку нажал пользователь, программе будет
# передан тот или иной "кнопочный" параметр: add_student или
# show_db
# ПРИМЕЧАНИЕ. Функция exists определяет, есть ли в хеш-массиве
#             элемент с указанным ключом.

# нажали кнопку "Записать"
if ( exists $prms{ 'add_student' } )
{
  # откроем файл в режиме дополнения (для этого -
  # символы ">>")
  # ПРИМЕЧАНИЕ. Ваша программа должна иметь права на запись
  #             в каталог /home/stud/db. Поскольку web-сервер
  #             запускает все программы от имени пользователя
  #             daemon, то вы должны позаботиться, чтобы этот
  #             пользователь имел вышеуказанные права.
  #             Для назначения прав служит команда chmod.
  open( ADD, ">> /home/stud/db/students.db" ) ||
   die "Не могу открыть файл /home/stud/db/students.db: $!\n";

  # добавим в конец файла (нашей простой базы данных) новую
  # информацию: фамилию и имя, введенные в поля формы
  # ПРИМЕЧАНИЕ. Обратите внимание на отсутствие запятой
  #             после ADD.
  print( ADD "$prms{ 'lname' } $prms{ 'fname' }\n" );
```

```
  # а это сообщение будет направлено в окно браузера
  # для пользователя
  print "Информация записана в базу данных\n";

  # закроем нашу базу данных
  close( ADD ) ||
   die "Не могу закрыть файл /home/stud/db/students.db: $!\n";
}
# нажали кнопку "Показать базу данных"
elsif ( exists $prms{ 'show_db' } )
{
  # откроем файл в режиме чтения
  open( READ, "< /home/stud/db/students.db" ) ||
   die "Не могу открыть файл /home/stud/db/students.db: $!\n";

  # сделаем заголовок (теги <B>...</B> означают выделение
  # полужирным шрифтом), а после него пропустим пустую строку
  # (поэтому два тега <BR>)
  print "<B>База данных</B><BR><BR>\n";

  # считываем файл построчно и построчно выводим его,
  # нумеруя строки
  while ( <READ> )
  {
    # вспомните, что означают символы "."
    # в данном случае переменная $_ "впитывает" в себя целую
    # строку файла; тег <BR> служит указанием браузеру перейти
    # на новую строку, а символ "\n" нужен для того, чтобы
    # сформированный HTML-документ имел читаемый вид
    # (если мы захотим его посмотреть в исходном виде,
    # то браузеры предоставляют такую возможность)
    # ПРИМЕЧАНИЕ. Попробуйте убрать поочередно тег <BR>
    #             и символ "\n" и посмотрите, что получается.
    print ++$i . " " . $_ . "<BR>\n";
  }

  # закроем файл
  close( READ ) ||
   die "Не могу закрыть файл /home/stud/db/students.db: $!\n";
}

# сформируем заключительные теги
print "</BODY>\n";
print "</HTML>\n";

# завершаем программу
exit( 0 );
```

Схема работы CGI-программ такова. Когда вы вводите какие-то данные в поля формы и нажимаете кнопку типа «submit», то ваш браузер собирает все введенные данные и отправляет их по сети Интернет (или по

локальной сети) web-серверу, например, Apache. Web-сервер получает эти данные и запускает программу, которая указана в теге FORM с помощью выражения ACTION. Полученные данные web-сервер передает этой программе на ее стандартный ввод. Программа должна уметь принять такие данные. Затем эта программа, в нашем случае – **cgi.pl**, формирует HTML-документ путем простой печати на стандартный вывод символьных строк, содержащих полезную информацию, а также HTML-тегов, необходимых для формирования документа. Все содержимое стандартного вывода программы web-сервер отправляет браузеру пользователя на тот компьютер, с которого была запущена форма для ввода данных. Браузер, получив такой набор тегов и полезной информации, формирует HTML-документ «на лету» и отображает его в своем окне.

Для того чтобы описанная схема работала, нужно обеспечить web-серверу доступ к тем HTML-документам и CGI-программам, которые должны взаимодействовать между собой. Конфигурация web-сервера Apache по умолчанию такова, что HTML-документы должны находиться в его подкаталоге **htdocs**, а CGI-программы – в подкаталоге **cgi-bin**. Скопируйте ваши файлы в соответствующие каталоги, предварительно назначив CGI-программе такие привилегии доступа, чтобы ее могли открывать в режиме чтения и исполнять все пользователи (по умолчанию web-сервер Apache запускается с правами пользователя daemon и, следовательно, все запускаемые им программы будут иметь привилегии этого пользователя):

```
chmod 755 cgi.pl
cp test.html /usr/local/apache2/htdocs
cp cgi.pl /usr/local/apache2/cgi-bin
```

Если в вашей операционной системе FreeBSD нет учетной записи пользователя stud, то создайте ее. Это позволит упростить запуск программы **cgi.pl**.

ПРИМЕЧАНИЕ. Конечно, вы можете обойтись и без создания этой учетной записи, но тогда необходимо немного подкорректировать программу, а именно: изменить путь к файлу данных, который используется в программе **cgi.pl**.

Войдите в систему под именем пользователя stud и создайте каталог **/home/stud/db**, а в этом каталоге создайте пустой файл **students.db**. Назначьте этому файлу такие права доступа, чтобы все пользователи могли открывать его в режиме чтения/записи. Это необходимо по той же причине, по которой ранее вы назначили права доступа к CGI-программе.

```
mkdir /home/stud/db
cd /home/stud/db
> students.db
chmod 666 students.db
```

Теперь можно проверить работу нашей CGI-программы (это можно делать, регистрируясь в системе с правами любого пользователя). Проверьте, запущен ли web-сервер Apache (как это сделать, описано выше). Если он запущен, то введите такую команду:

```
lynx http://localhost/test.html
```

Введите несколько записей, а затем просмотрите содержимое «базы данных».

Обратите внимание, что если для открытия HTML-документа использовать команду

```
lynx test.html
```

то взаимодействия HTML-документа и CGI-программы не будет. Очень важно это понимать.

Если вы установили графическую среду KDE, то вместо браузера **lynx** можно использовать браузер, входящий в ее состав.

Дадим еще два полезных совета. Первый совет: обязательно проверяйте синтаксис CGI-программы перед ее использованием:

```
perl –c cgi.pl
```

Совет второй: в процессе своей работы web-сервер Apache ведет файлы-журналы **access_log** и **error_log** (на жаргоне они называются «логи»), которые находятся в подкаталоге **logs**. В случае возникновения ошибок при запуске CGI-программ рекомендуем вам обращаться к этим файлам-журналам.

В операционной системе Windows

Для запуска программы **cgi.pl** в среде Windows необходимо внести в программу небольшие изменения. Во-первых, нужно изменить первую строку, которая содержит путь к интерпретатору Perl:

```
#!perl –w
```

Во-вторых, необходимо изменить путь к файлу «базы данных» **students.db** с учетом принятых в среде Windows правил. Возможен такой вариант:

```
c:\\my_dir\\my_subdir\\students.db
```

Обратите внимание на удвоение символа «\», разделяющего подкаталоги. Можно, правда, использовать и разделитель подкаталогов в стиле UNIX:

```
c:/my_dir/my_subdir/students.db
```

Скопировать HTML-документ и CGI-программу следует в такие же подкаталоги web-сервера Apache, как и в системе FreeBSD. Назначать привилегии доступа не придется.

9.3. Взаимодействие CGI-программы и СУБД PostgreSQL

В настоящее время для создания динамических web-страниц и web-сайтов используются различные технологии и языки программирования. Мы выбрали для демонстрационного примера язык Perl и СУБД PostgreSQL. Однако этот выбор вовсе не означает какой-либо критики в адрес других инструментов. Наша задача – показать базовые приемы и объяснить основные принципы, а не заниматься детальным анализом конкурирующих технологий.

Давайте немного модифицируем HTML-документ, который был представлен в предыдущем параграфе.

Файл **test_db.html**

```
<HTML>
<HEAD>
<TITLE>
Запрос информации с помощью формы
</TITLE>

<BODY BGCOLOR = "#FFFF00">
<H1>Список студентов</H1>

<FORM ACTION = "/cgi-bin/cgi_db.pl" METHOD = "POST">
Номер зачетной книжки: <INPUT TYPE = "text" NAME = "mark_book"
                        SIZE = 6> <BR>
Фамилия, имя, отчество: <INPUT TYPE = "text" NAME = "name"
                         SIZE = 50> <BR>
Серия паспорта: <INPUT TYPE = "text" NAME = "psp_ser"
                 SIZE = 10> <BR>
Номер паспорта: <INPUT TYPE = "text" NAME = "psp_num"
                 SIZE = 7> <BR>
<INPUT TYPE = "submit" NAME = "add_student"
VALUE = "Записать">
<INPUT TYPE = "submit" NAME = "show_db"
VALUE = "Показать весь список">
</FORM>
```

```
</BODY>
</HTML>
```

Мы напишем и новую CGI-программу. Ее главными чертами будут использование модуля (пакета) **CGI.pm** и организация взаимодействия с базой данных под управлением СУБД PostgreSQL. Использование пакета **CGI.pm** позволит значительно упростить и облегчить программирование процедур обмена данными между web-сервером и программой.

Если вы помните, то в заданиях к предыдущей главе мы рекомендовали вам подумать над изменением числового типа данных для поля «Серия паспорта» на символьный. При разработке программы, текст которой приведен ниже, эта модификация была учтена.

Для того чтобы удалить таблицу students, выполните SQL-команду:

```
DROP TABLE students CASCADE;
```

А затем создайте эту же таблицу вновь, но с учетом модификации. Можно обойтись и без удаления таблицы – использовать SQL-команду ALTER TABLE. Однако это более сложный вариант. Конечно, если бы в вашей базе данных хранились реальные данные, то вот так запросто удалять таблицы было бы нельзя.

Программа **cgi_db.pl**

```
#!/usr/bin/perl -w
# ------------------------------------------------------------
# Программа: организация взаимодействия CGI-скрипта
# с СУБД PostgreSQL
# ------------------------------------------------------------

use strict;
use CGI;

# сообщения об ошибках будут выводиться в окно браузера
use CGI::Carp qw( fatalsToBrowser );

use Pg;                          # для работы с PostgreSQL
use constant DBNAME => 'test';   # имя базы данных

# будем выводить стек вызовов функций при возникновении ошибок
$SIG{ __DIE__ } = \&CGI::Carp::confess;

my $conn;               # дескриптор соединения с базой данных
my $query = CGI->new(); # новый CGI-объект

# отмена буферизации на устройствах
select( STDERR );
$| = 1;
select( STDOUT );
$| = 1;
```

```
print $query->header();       # HTTP-заголовок
print $query->start_html();   # HTML-заголовок

# подключение к базе данных
$conn = Pg::connectdb( "dbname=" . DBNAME .
                       " user=postgres" );

# проверка успешности подключения
die $conn->errorMessage
  unless PGRES_CONNECTION_OK eq $conn->status;

# если нажата кнопка "Записать"
if ( $query->param( 'add_student' ) )
{
  # ввод записи в таблицу Students
  add_student( $conn, $query );
}

# если нажата кнопка "Показать весь список"
if ( $query->param( 'show_db' ) )
{
  # вывод всего списка студентов
  show_db( $conn, $query );
}

# вывод заключительных тегов HTML
print $query->end_html();

exit ( 0 );

sub show_db
{
  my $conn = shift;   # дескриптор соединения с базой данных
  my $q = shift;      # CGI-объект

  my $result;         # результат выполнения SQL-запроса
  my @row;            # массив для хранения одной записи
                      # из выборки
  my $command;        # строка SQL-запроса для работы с БД

  # формирование SQL-запроса в виде символьной строки
  $command = "SELECT * FROM students " .
             "ORDER BY name";

  # выполнение запроса к базе данных
  $result = $conn->exec( $command );

  # если произошла ошибка при выполнении запроса к базе данных
  if ( PGRES_TUPLES_OK ne $result->resultStatus )
  {
    die $conn->errorMessage;  # завершить работу
  }
```

```
  # если в выборке нет ни одной строки
  if ( $result->ntuples == 0 )
  {
    print "В таблице Students нет записей\n";
    return ( 0 );
  }

  # выведем информацию в виде таблицы
  print "<TABLE BORDER WIDTH = 100%>\n";
  print "<TR>\n";
  print "<TD>Номер зачетной книжки</TD>\n";
  print "<TD>Фамилия, имя, отчество</TD>\n";
  print "<TD>Серия паспорта</TD>\n";
  print "<TD>Номер паспорта</TD>\n";
  print "</TR>\n";

  # выводим все строки из полученной выборки
  # метод fetchrow считывает в массив очередную строку
  # из выборки
  while ( @row = $result->fetchrow )
  {
    print "<TR>\n";

    # выводим значения всех полей текущей записи
    print "<TD>$row[ $result->fnumber( 'mark_book' )
                  ]</TD>\n";
    print "<TD>$row[ $result->fnumber( 'name' ) ]</TD>\n";
    print "<TD>$row[ $result->fnumber( 'psp_ser' ) ]</TD>\n";
    print "<TD>$row[ $result->fnumber( 'psp_num' ) ]</TD>\n";
    print "</TR>\n";
  }
  print "</TABLE>\n";

  return 0;
}

sub add_student
{
  my $conn = shift;   # дескриптор соединения с базой данных
  my $q = shift;      # CGI-объект

  my $result;         # результат выполнения SQL-запроса
  my $command;        # строка SQL-запроса для работы с БД

  # формирование SQL-запроса в виде символьной строки
  # (будьте внимательны при расстановке многочисленных
  # кавычек)
  $command = "INSERT INTO students " .
                  "( mark_book, name, psp_ser, psp_num ) " .
             "VALUES( " . $q->param( 'mark_book' ) . ", " .
                    "'" . $q->param( 'name' ) . "', " .
                    "'" . $q->param( 'psp_ser' ) . "', " .
                    $q->param( 'psp_num' ) . " )";
```

```
  # можно вывести команду в окно браузера для отладочных целей
  # print "$command<BR><BR>\n";

  # выполнение запроса к базе данных
  $result = $conn->exec( $command );

  # если произошла ошибка при вставке записей в базу данных
  if ( PGRES_COMMAND_OK ne $result->resultStatus )
  {
    die $conn->errorMessage;
  }
  else
  {
    print "Запись добавлена в базу данных<BR>\n";
  }

  return 0;
}
```

Как и в предыдущем параграфе, назначьте необходимые привилегии доступа к программе **cgi_db.pl** и скопируйте ее и HTML-документ в соответствующие подкаталоги web-сервера Apache:

```
chmod 755 cgi_db.pl
cp test_db.html /usr/local/apache2/htdocs
cp cgi_db.pl /usr/local/apache2/cgi-bin
```

Проверьте, запущены ли web-сервер и СУБД PostgreSQL. Если все в порядке, то протестируйте программу **cgi_db.pl**:

```
lynx http://localhost/test_db.html
```

В операционной системе Windows

В среде Windows (MSYS) установить модуль (пакет) **Pg.pm**, используя указания автора пакета, не удается, поэтому необходимо использовать модуль (пакет) **DBI.pm** с драйвером **DBD::Pg**. Получить последние версии этих пакетов можно на сайте http://www.cpan.org.

Контрольные вопросы и задания

1. Модифицируйте программу **cgi.pl** следующим образом: добавьте в HTML-документ еще одно поле, которое будет называться «Оценка». Модифицированная программа должна обрабатывать это поле, записывать его значение в файл данных **students.db**. Но прежде чем выполнить ввод записи в файл данных, программа должна сделать проверку правильности за-

полнения этого поля. Допустимые значения: 3, 4 и 5. Если введено недопустимое значение, программа должна вывести сообщение об этом. При нажатии на кнопку «Показать базу данных» программа должна после вывода всех записей из файла **students.db** вывести и средний балл для этой группы студентов.

2. Сделайте процедуру show_db, выполняющую выборку записей (строк) из таблицы «Студенты» в программе **cgi_db.pl**, более интеллектуальной. Возможная схема ее работы такая: если в поля формы введены какие-либо данные, то они должны служить в качестве критерия выборки строк из таблицы. Если же поля формы пустые, то производится выборка всех строк таблицы.

При выполнении этого задания рекомендуем вам обратиться к краткому введению в язык SQL, изученному вами ранее. Можно использовать способность СУБД PostgreSQL производить выборку строк при помощи регулярного выражения, например:

```
SELECT * FROM Students WHERE name ~ '^Ив';
```

Такая команда позволит выбрать все строки, в которых значение поля name начинается с символов «Ив».

3. Модифицируйте структуру таблицы «Студенты» (students) с помощью SQL-команды ALTER TABLE, а именно: добавьте колонку (поле) «Дата зачисления». Эта колонка (поле) будет иметь тип данных date. Соответствующим образом модифицируйте HTML-документ **test_db.html** и программу **cgi_db.pl**. Сделайте проверку правильности даты, введенной в окно HTML-документа.

4. Модифицируйте программу **cgi_db.pl** для использования модуля (пакета) **DBI.pm** с драйвером **DBD::Pg** в среде Windows.

10. Средства создания интерфейса пользователя

Заключительная глава пособия посвящена средствам разработки интерфейса пользователя. В настоящее время доминирует графический тип интерфейса. Поэтому мы представим одну из наиболее известных свободно распространяемых библиотек, предназначенных для создания кроссплатформенных приложений с графическим интерфейсом, – wxWidgets. С другой стороны, консольные приложения зачастую также нуждаются в удобном, пусть и простом, интерфейсе пользователя. Для разработки интерфейса этого типа в операционной системе UNIX традиционно используется библиотека curses (или ncurses).

10.1. Библиотека ncurses

При установке ОС FreeBSD эта библиотека также устанавливается, если вы инсталлируете исходные тексты программ и утилит операционной системы. Располагается она в каталоге **/usr/src/contrib/ncurses**. В состав пакета **ncurses** входит большое число программ-примеров, которые находятся в подкаталоге **test**. Мы рекомендуем вам скомпилировать эти примеры и использовать их в качестве учебного пособия. Для обеспечения возможности использования отладчика **gdb** служит параметр **–g** компилятора языка С. При этом рекомендуем вам отключить оптимизацию объектного кода.

Поскольку вы уже приобрели некоторый опыт установки программных продуктов в среде FreeBSD, то мы опишем процедуру компиляции примеров программ ncurses очень кратко.

Если вы не хотите отключать оптимизацию объектного кода, то можете запускать утилиту **configure** из файлового менеджера (например, **deco**). Если же вы решили последовать нашему совету и отключить оптимизацию, то выйдите из файлового менеджера непосредственно в среду командного процессора операционной системы FreeBSD.

Выполните следующие команды:

```
cd /usr/src/contrib/ncurses/test
CFLAGS="-g -O0" ./configure
```

Если оптимизацию отключать не требуется, то команда упрощается:

```
./configure
```

Можно отключить оптимизацию и путем непосредственного редактирования файла **Makefile** в текущем каталоге: просто удалите параметр **–O2** из строки

```
CFLAGS = -g -O2
```

Теперь можно выполнить компиляцию программ:

```
make
```

Эти программы активно используют практически всю площадь экрана для создания различных элементов интерфейса пользователя. Поэтому для изучения их работы в среде отладчика **gdb** можно сделать так, чтобы свои операции он выполнял на том виртуальном терминале, с которого вы его запустили, а программа выполнялась на другом терминале. Например, вы планируете запустить отладчик на терминале ttyv1 и хотите, чтобы программа выполнялась на терминале ttyv2. Команда будет такой:

```
gdb -tty=/dev/ttyv2 ./xmas
```

Для переключения между виртуальными терминалами вы используете в данном случае клавиши Alt-F2 и Alt-F3. При этом программа будет работать на терминале ttyv2 независимо от того, зарегистрировался на нем какой-либо пользователь или нет.

В комплект библиотеки **ncurses** входят также дополнительные библиотеки **panel**, **form** и **menu**. В электронных руководствах **man** содержится исчерпывающая информация о системе **ncurses**:

```
man ncurses
man panel
man form
man menu
```

Кроме того, в каталоге **/usr/src/contrib/ncurses/doc/html** есть очень хорошее введение в технологию программирования с использованием этой библиотеки – файл **ncurses-intro.html**.

Подключение библиотеки **ncurses** можно проиллюстрировать на примере программы **xmas**, находящейся в подкаталоге **test** (здесь этапы компиляции и компоновки разделены, хотя это не обязательно делать для программ, состоящих из одного исходного модуля):

```
gcc -c -I.  -DHAVE_CONFIG_H -g xmas.c
gcc -g -o xmas xmas.o -lpanel -lmenu -lform -lncurses
```

В операционной системе Windows

В среде Windows библиотека **ncurses** не работает.

10.2. Библиотека wxWidgets

Чтобы установить эту библиотеку в среде ОС FreeBSD, вам необходимо предварительно установить графическую подсистему X11.

Загрузите последнюю стабильную версию библиотеки с сайта http://www.wxwidgets.org.

1. Войдите в систему под именем root. Распакуйте архивный файл в каталог, например, **/usr/src** или **/home/my_dir/my_subdir**.

ПРИМЕЧАНИЕ. Как и прежде, вместо конкретных цифр, отражающих номер версии в имени архивного файла, мы будем использовать символы «x».

```
tar xzvf wxWidgets-2.x.x.tar.gz
```

2. Перейдите в каталог **wxWidgets-2.x.x** и найдите в нем файл **install-x11.txt**, в котором процедура установки описана подробно.

```
cd wxWidgets-2.x.x
```

3. Создайте подкаталог, в котором будет выполняться построение библиотеки (вы можете создать несколько таких подкаталогов и построить в них различные варианты библиотеки на основе разных параметров конфигурации):

```
mkdir build_lib
```

4. Перейдите в этот подкаталог и запустите из него утилиту **configure** со следующими параметрами (команду необходимо вводить в одну строку):

```
cd build_lib
../configure --with-X11 --enable-shared --enable-log
--enable-debug --enable-debug_gdb --enable-debug_cntxt
--enable-debug_info --enable-debug_flag --without-odbc
--enable-monolithic --with-libiconv-prefix=/usr/local
--disable-optimise --enable-intl --enable-unicode
--disable-threads > conf_log.txt 2>&1 &
```

Обратите внимание на две точки перед именем утилиты – это означает, что она находится в родительском каталоге.

Этот набор параметров предназначен для создания разделяемой библиотеки с поддержкой отладчика **gdb**, с отключенной оптимизацией объектного кода (это удобно при изучении исходных кодов в отладчике), с отключенной поддержкой потоков операционной системы, с поддержкой средств интернационализации. Поскольку используется параметр **--enable-monolithic**, то эта библиотека будет построена в виде одного модуля **.so**,

хотя в принципе она может быть построена и в виде набора библиотек меньшего размера. Каждая из таких библиотек отвечает за выполнение определенного набора функций.

Приобретя некоторый опыт, вы можете выбрать и другую конфигурацию. Для получения списка всех возможных параметров запускайте утилиту **configure** с параметром **--help**:

```
configure --help
```

5. Теперь скомпилируйте библиотеку (этот процесс может занять от пяти минут до одного часа в зависимости от мощности вашего компьютера):

```
make > make.log 2>&1 &
```

6. Установите созданную библиотеку и другие файлы, необходимые для ее использования:

```
make install > make_install.log 2>&1 &
```

7. В состав дистрибутива входит большое число программ-примеров, которые находятся в подкаталоге **samples**. Скомпилируйте самую простую из этих программ, затем запустите графическую подсистему (если она еще не запущена) и выполните скомпилированную программу:

```
cd samples/minimal
make
startx
./minimal
```

Библиотека wxWidgets сопровождается хорошей документацией, которую можно загрузить с того же сайта. Причем, документация представлена в различных форматах, в том числе, pdf и html.

В операционной системе Windows

В среде Windows (MSYS) библиотека устанавливается аналогично.

1. Распакуйте файл **wxWidgets-2.x.x.tar.gz** в каталог **C:\wx**. Очень важно выполнить следующее требование: не устанавливайте продукт в каталог, путь к которому содержит пробелы. Например, традиционный каталог **C:\Program Files** не подойдет.

2. Найдите в файле **INSTALL-MSW.txt** раздел «Cygwin/MinGW compilation», а в этом разделе найдите подраздел «Using configure».

3. Запустите среду MSYS и перейдите в каталог, содержащий исходные тексты библиотеки wxWidgets (в среде MSYS путь **C:\wx** выглядит как

/c/wx). Создайте в нем подкаталог, в котором вы будете строить библиотеку, и перейдите в этот подкаталог:

```
cd /c/wx
mkdir build_lib
cd build_lib
```

4. Запустите утилиту configure с параметрами, большинство которых таки же, как и в среде FreeBSD (команду вводите в одну строку)

```
../configure --with-msw --enable-shared --enable-log
--enable-debug --enable-debug_gdb --enable-debug_cntxt
--enable-debug_info --enable-debug_flag --with-odbc
--enable-monolithic --disable-optimise --enable-intl
--enable-unicode --enable-dnd
```

5. Дальнейшие шаги такие же, как в системе FreeBSD:

```
make > make.log 2>&1
make install > make_instal.log 2>&1
cd samples/minimal
make
./minimal.exe
```

При запуске программы **minimal.exe** будет выведено сообщение об ошибке (невозможно найти библиотеку). Для устранения этой проблемы дополните переменную PATH среды MSYS:

```
PATH=$PATH:/usr/local/lib
```

Повторите запуск программы **minimal.exe**.

Для того чтобы программу **minimal.exe** (и других программы, созданные в среде MSYS с использованием библиотеки wxWidgets) можно было запускать непосредственно из среды Windows, нужно обеспечить, чтобы библиотека **mingwm10.dll** и библиотеки wxWidgets были доступны при запуске программ. Для этого нужно добавить путь **C:\MinGW\bin** и путь **C:\msys\1.0\local\lib** в переменную PATH (последовательность действий следующая: кнопка «Пуск» – Панель управления – Система – Дополнительно – Переменные среды – Системные переменные – PATH).

Для изучения возможностей библиотеки wxWidgets используйте документацию, которую можно получить с того же сайта http://www.wxwidgets.org.

Контрольные вопросы и задания

1. На основе библиотеки **ncurses** модифицируйте интерфейс программы, выполняющей выборку информации из базы данных под управлением СУБД PostgreSQL.

2. Скомпилируйте все примеры программ, содержащиеся в подкаталоге **samples** установочного комплекта библиотеки wxWidgets, и ознакомьтесь с их работой.

3. С помощью утилиты **ldd** посмотрите список разделяемых библиотек, от которых зависит программа **minimal** в среде ОС FreeBSD.

4. На основе программы **minimal** напишите свою программу.

Заключение

Итак, вы познакомились с рядом инструментальных средств и технологий, используемых в среде ОС UNIX и находящих все более широкое применение в среде ОС Windows. В одном пособии, независимо от его объема, невозможно осветить все вопросы, которые могут представлять интерес для современного программиста. Поэтому рекомендуем вам не останавливаться на достигнутом, а продолжать изучение как инструментария, описанного в этом пособии, так и аналогичного, возможно, конкурирующего, инструментария, не упомянутого здесь. Большую помощь может оказать документация, входящая в комплект многих современных программных средств, распространяемых на условиях свободного ПО. Например, объем документации, сопровождающей библиотеку wxWidgets, составляет около двух тысяч страниц.

Очень полезным для начинающего программиста было бы изучение исходных текстов серьезных программных продуктов, разработанных лучшими программистами мира. Программное обеспечение с открытым исходным кодом предоставляет такую возможность.

Важный вопрос, не затронутый в пособии, – это лицензирование программных продуктов. Даже свободное ПО использует зачастую различные лицензионные соглашения. Рекомендуем вам ознакомиться с текстами этих соглашений, например, для СУБД PostgreSQL, web-сервера Apache, утилиты GNU make, языка программирования Perl, операционной системы FreeBSD и др. Лицензионные соглашения всегда включаются в дистрибутивный комплект исходных текстов и документации программного продукта.

В приведенном списке рекомендуемой литературы представлена лишь малая часть полезных книг. Не стоит ограничиваться только этим списком: ведь постоянно выходят новые книги. Мы надеемся, что после изучения материала нашего пособия вам будет легче ориентироваться в потоке специальной литературы.

Желаем вам, уважаемый читатель, не только профессионального роста, но и получения удовольствия от самого процесса программирования!

Рекомендуемая литература

1. Брайант, Р. Компьютерные системы: архитектура и программирование [Текст] : пер. с англ. / Р. Брайант, Д. О'Халларон. – СПб. : БХВ-Петербург, 2005. – 1104 с.

2. Браун, М. Perl. Архив программ [Текст]: пер. с англ. / М. Браун. – М. : БИНОМ, 2001. – 720 с.

3. Кристиансен, Т. Perl: библиотека программиста [Текст]: пер. с англ. / Т. Кристиансен, Н. Торкингтон. – СПб. : Питер, 2001. – 736 с.

4. Митчелл, М. Программирование для Linux. Профессиональный подход [Текст] : пер. с англ. / М. Митчелл, Дж. Оулдем, А. Самьюэл. – М. : Вильямс, 2002. – 288 с.

5. Реймонд, Э. Искусство программирования для Unix [Текст] : пер. с англ. / Э. Реймонд. – М. : Вильямс, 2005. – 544 с.

6. Рочкинд, М. Программирование для UNIX [Текст] : пер. с англ. / М. Рочкинд. – М. Русская редакция ; СПб. : БХВ-Петербург, 2005. – 704 с.

7. Харт, Дж. Системное программирование в среде Windows [Текст] : пер. с англ. / Дж. Харт. – М. : Вильямс, 2005. – 592 с.

8. Эбен, М. FreeBSD. Энциклопедия пользователя [Текст] : пер. с англ. / Майкл Эбен, Брайан Таймэн. – К. : ТИД «ДС», 2002. – 736 с.

Учебное издание

МОРГУНОВ Евгений Павлович
МОРГУНОВА Ольга Николаевна
ТЫНЧЕНКО Валерия Валериевна

Технологии разработки программ на основе инструментария с открытым исходным кодом

Вводный курс

Учебное пособие

Компьютерная верстка *Е. П. Моргунова*

Подписано в печать 25.12.2006. Формат 60×84/16. Бумага офсетная.
Печать плоская. Усл. печ. л. 8,60. Уч.-изд. л. 5,20. Тираж 100 экз.
Заказ № ___.

Отпечатано в типографии «Город».
660014 г. Красноярск, ул. Юности, 24 «а».