Министерство образования и науки Российской Федерации
Сибирский государственный университет науки и технологий
имени академика М. Ф. Решетнева

Е. П. Моргунов
О. Н. Моргунова

Технологии разработки программ в среде операционных систем Linux и FreeBSD

Вводный курс

Утверждено редакционно-издательским советом университета
в качестве учебного пособия для бакалавров,
обучающихся по направлениям подготовки
09.03.01 – «Информатика и вычислительная техника»,
09.03.02 – «Информационные системы и технологии»,
09.03.04 – «Программная инженерия» всех форм обучения

Красноярск 2018

УДК 004.4
ББК 32.973.26-018
М 79

Рецензенты:
кандидат технических наук, главный специалист-эксперт В. А. Морозов
(Управление Федерального казначейства по Красноярскому краю);
кандидат технических наук, доцент А. Н. Горошкин
(Сибирский государственный университет науки и технологий
имени академика М. Ф. Решетнева)

Моргунов, Е. П.
М 79 Технологии разработки программ в среде операционных систем Linux и FreeBSD. Вводный курс : учеб. пособие / Е. П. Моргунов, О. Н. Моргунова. – Красноярск, 2018. – 207 с.

В учебном пособии описаны основные технологии разработки программ на основе инструментария с открытым исходным кодом в среде операционных систем Linux и FreeBSD. Освещаются такие вопросы, как установка этого инструментария, его настройка и практическое использование. Приведены примеры программ, иллюстрирующие применение рассматриваемых технологий.

Пособие может быть полезно широкому кругу пользователей, знакомых с основами программирования в операционной системе Windows, желающих самостоятельно познакомиться с технологиями разработки программ в среде операционных систем Linux и FreeBSD.

УДК 004.4
ББК 32.973.26-018

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 язык PHP, но, изучив принципы и основные приемы, вы сможете самостоятельно освоить технологию разработки Интернет-приложений и на других языках.

Использование инструментов и технологий иллюстрируется в среде операционных систем Linux и FreeBSD. В качестве представителя семейства Linux была взята ОС Debian. Для ОС Debian на момент написания пособия актуальной была версия 9.1, а для ОС FreeBSD – версия 11.1.

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

```
psql –d test –U postgres
```

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-gtk=3 --enable-shared --enable-debug
--enable-intl --enable-unicode --enable-utf8
CFLAGS="-g -O0" CXXFLAGS="-g -O0" > conf_log.txt 2>&1 &
```

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

```
ldd ./demo_shared_lib
```

на экран будет выведено следующее:

```
        linux-gate.so.1 (0xb77b5000)
        libgeometry.so.1 => not found
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6
(0xb75d7000)
        /lib/ld-linux.so.2 (0xb77b7000)
```

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

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

Предполагается, что обучающийся должен иметь установленную ОС Debian или FreeBSD. Эта система должна быть русифицирована. Должна быть установлена графическая система X Window и какое-либо окружение рабочего стола (desktop environment), например, KDE, GNOME, LXDE или Xfce. Обучающийся должен знать элементарные команды для работы в среде UNIX-подобной операционной системы.

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

1. Текстовые редакторы

Хотя мы живем в эпоху интегрированных сред разработки (Integrated development environment – IDE), тем не менее, автономные текстовые редакторы не исчезли как класс инструментов и продолжают широко использоваться для написания исходных текстов программ.

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

1.1. Текстовый редактор vi

Этот текстовый редактор является одним из базовых средств ОС Debian и FreeBSD 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, посоветуем использовать электронную справочную систему:

```
man vi
```

Для запуска редактора и создания нового файла введите

```
vi my_file.txt
```

Если вы работаете в среде ОС Debian, то вместо редактора **vi** запустится редактор **vim** (Vi Improved – т. е. улучшенный **vi**). Это более новый и совершенный вариант оригинального редактора **vi**. Однако все приемы использования редактора, описанные ниже, подходят как для **vi**, так и для **vim**.

Сразу после запуска 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* той позиции, в которой стоит курсор;

i ввести текст *перед* той позицией, в которой стоит курсор;

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 нужно 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 ":".

1.2. Текстовый редактор joe

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

ПРИМЕЧАНИЕ. Если вы работаете в ОС FreeBSD, то можете выполнять все команды, приведенные ниже, в среде файлового менеджера **Demos Commander (deco)**. 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. Если же вы работаете в ОС Debian, то можете выполнять все команды в среде файлового менеджера **Midnight Commander (mc)**. За просмотр команд, введенных ранее, отвечают комбинации клавишей Esc-P (просмотр назад) и Esc-N (просмотр вперед). При этом клавишу Esc и клавишу, например, P, следует нажимать *не одновременно*, а поочередно: сначала Esc, а затем – P (или другую клавишу, которая требуется).

Разархивируйте архив с исходными текстами редактора **joe**:

```
tar xzvf joe-4.4.tar.gz
```

В результате будет создан подкаталог **joe-4.4**, в котором разместятся файлы исходных текстов. Перейдите в этот подкаталог и выполните ряд команд (обратите внимание на символы «./» в одной из команд):

```
cd joe-4.4
./configure > conf_log.txt 2>&1
make > make_log.txt 2>&1
make install > make_install_log.txt 2>&1
```

При выполнении этих команд будут создаваться файлы-журналы, в которые будут перенаправляться все выводимые сообщения. В случае возникновения каких-либо проблем эти файлы-журналы могут помочь выяснить их причину.

Если в файлах-журналах не оказалось 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.

Данный редактор имеет следующую особенность: многие 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 требуемого 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 целевую позицию файла и нажмите клавиши Ctrl-K-C, а для перемещения блока – Ctrl-K-M.

Одной из сильных сторон редактора является возможность работы с несколькими файлами одновременно. Чтобы открыть 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). Для того чтобы на экране одновременно присутствовал только один открытый файл, необходимо нажать комбинацию клавишей Ctrl-K-I. Для возврата к режиму, когда на экране одновременно в разных окнах отображается несколько открытых файлов, нажмите Ctrl-K-I повторно.

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 *двукратным* нажатием клавиши Tab вывести на экран список файлов текущего каталога и выбрать файл из списка. Можно ввести первые символы имени файла и также нажать клавишу Tab. В результате будет выведен список файлов текущего каталога, имена которых начинаются с введенных символов. 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 Linux или 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**.

Мы сделали только краткий обзор основных возможностей редактора. Рекомендуем изучить основные приемы работы с ним 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. Они призваны сделать работу с ним более удобной. Конфигурационный 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 (в системе FreeBSD это **deco**, а в системе Debian это **mc)**, либо самим редактором **joe**.

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 62), в начале которой располагается параметр **-asis**. Он отвечает за правильное отображение букв русского алфавита (вопросы русификации операционной системы в данном учебном пособии не рассматриваются). Обратите внимание, что перед этим параметром в начале строки стоит один пробел. Нужно удалить этот пробел, тем самым параметр будет активизирован.

2. Найдите строку (строка номер 96), в начале которой располагается параметр **-pg**. Он отвечает за то, чтобы при перемещении по тексту с помощью клавишей Page Up и Page Down одна или более строк текста служили в качестве «связующих», повторяясь в верхней части окна редактора или в нижней, в зависимости от направления просмотра файла. Нужно

вместо букв «nnn» ввести число 1 и также удалить пробел в начале строки, тем самым активизируя параметр.

3. Найдите строку (строка номер 155), в начале которой располагается параметр **-guess_crlf**. Он отвечает за то, чтобы при просмотре файлов, созданных в операционной системе Windows, символы возврата каретки (ASCII-код 13) не отображались на экране. По умолчанию этот параметр включен, поэтому такие символы не отображаться. Если же требуется визуализировать такие символы, то вставьте пробел в начале строки, чтобы отключить параметр.

4. 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 827; 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 1105, 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 1134, 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 1144 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 1199 есть строки с ключевым словом save. Добавьте еще одну строку, а в качестве кода клавишей введите «.k2».

Для того чтобы клавиши F1 и F2 работали так, как мы предписали, нужно дополнительно деактивировать команды для запуска командной оболочки. Для этого найдите в конце файла, в районе строки номер 1218, строки, в начале которых стоят параметры с именами shell1 и shell2, и вставьте по одному пробелу в начале каждой такой строки.

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. Для выполнения дальнейших настроек редактора рекомендуем самостоятельно изучить конфигурационный файл **joerc**.

1.3. Текстовый редактор emacs

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

Данный редактор имеет очень широкие возможности, однако и порог вхождения – довольно высокий. Поэтому наша цель – только познакомить вас с ним, показать, как его установить и запустить. Предполагается, что дальнейшее изучение возможностей редактора вы будете выполнять самостоятельно с помощью документации.

Если вы работаете в среде ОС FreeBSD, то для установки редактора **emacs** можно воспользоваться коллекцией так называемых портов (Ports Collection). Для установки программ из этой коллекции необходимо наличие доступа к сети Интернет. В каждом подкаталоге этой коллекции содержатся только файлы, необходимые для отыскания исходных текстов конкретной программы в сети Интернет, их загрузки на компьютер и компиляции. Выполните следующие команды:

```
cd /usr/ports/editors/emacs
make install clean
```

Если вы работаете в среде ОС Debian, то для установки редактора **emacs** можно воспользоваться командой

```
apt-get install emacs25
```

Будет установлен редактор **emacs** версии 25. Его можно запустить как из текстового терминала (консоли), так и из графического окружения, например, KDE или Xfce. Например, в среде KDE можно открыть системную консоль и затем выполнить команду

```
emacs
```

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.

Редактор «умеет» очень многое, например, из него можно вызывать отладчик **gdb**. Для расширения возможностей редактора и более тонкой подстройки его под конкретные задачи можно написать макросы.

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**.

4. Ознакомьтесь с конфигурационным файлом редактора **joe** и подумайте, какие еще настройки было бы целесообразно сделать в этом файле. Выполнив их, проверьте на практике, на что они повлияли, стало ли удобнее работать.

5. Откройте в редакторе **joe** несколько файлов одновременно и скопируйте какой-нибудь фрагмент текста из одного файла в другой.

6. Попробуйте ввести текст на русском языке в редакторе **emacs** и сохранить этот текст.

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

2. Языки программирования

Наверное, многие согласятся с тем, что базовыми языками программирования являются C и C++. Но кроме них 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.

2.1. Языки C и C++

В UNIX-подобных операционных системах компилятор языка C устанавливается, как правило, по умолчанию. Также по умолчанию может быть установлен и компилятор языка C++, хотя это бывает не всегда.

Если вы работаете в среде ОС Debian, то для запуска компилятора C используется команда **cc** или **gcc**, а для запуска компилятора C++ служит команда **c++** или **g++**. Если вы работаете в среде ОС FreeBSD, то для запуска компилятора C используется команда **cc**, а для запуска компилятора C++ служит команда **c++**.

Поскольку наше пособие предназначено для тех студентов, которые уже имеют некоторый опыт программирования на языках C или C++, то мы не будем вдаваться в дальнейшие детали, касающиеся этих языков. Если вы хотели бы освежить свои знания языка C или познакомиться с этим языком, уже зная какой-то другой язык программирования, то можете воспользоваться кратким руководством, представленным на сайте авторов пособия: http://www.morgunov.org/teaching.html. Используя это руководство, вы сможете компилировать программы на языке C в среде ОС Linux или FreeBSD. Скажем только, что доступ к описаниям функций этого языка можно получить непосредственно из командной строки с помощью команды **man**, например:

```
man 3 sin
```

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. Для функции sin получим такие сведения:

```
#include <math.h>
```

В командной строке компиляции нужно указать параметр -lm, чтобы была подключена библиотека математических функций.

Рассмотрим процесс создания программы, написанной на языке С и состоящей из нескольких модулей.

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 по две функции. Таким образом, это очень простой случай, который служит только в качестве иллюстрации процесса создания многомодульной программы в среде UNIX-подобной операционной системы.

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

/* 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 );
}
```

Файл **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 этих модулей, должны объявляться без ключевого слова **static**.

Теперь создадим программу, которая использует функции из двух модулей.

Файл **demo.c**

```
/* ------------------------------------------------------------
   Программа, использующая функции из модулей
------------------------------------------------------------*/

#include <stdio.h>

#include "geometry.h"

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

  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 используемых функций. Его необходимо включать в каждый модуль, содержащий обращения (вызовы) к этим функциям. Приведем текст этого файла.

Файл **geometry.h**

```
/* ------------------------------------------------------------
   Прототипы функций из модулей
------------------------------------------------------------ */

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

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

На следующем шаге необходимо скомпилировать исходные тексты в объектные модули:

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

ПРИМЕЧАНИЕ. В качестве имени компилятора мы использовали gcc. В дальнейших командах мы также будем использовать это же имя. Если в вашей системе используется другой компилятор, например, clang или cc, то все команды вы должны корректировать соответственно.

Получив три объектных файла – circle.o, square.o и demo.o, – можно создать исполняемый файл:

```
gcc -o demo demo.o circle.o square.o
```

Параметр -o demo означает имя исполняемого файла. Для запуска полученной программы введите

```
./demo
```

Символы «./» означают текущий каталог. Они необходимы потому, что в UNIX-подобных операционных системах, как правило, поиск исполняемых программ в текущем каталоге по умолчанию не производится.

2.2. Язык Perl

2.2.1. Вводные сведения

Perl – это язык программирования, который первоначально предназначался для обработки текстовых файлов: поиска и замены строк символов, 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**.

Хотя этот язык является интерпретируемым, но работают программы на языке Perl очень быстро, т. к. сначала производится преобразование исходного текста во внутренний формат, а затем уже выполнение программы. Поэтому многократно выполняемые фрагменты программного кода преобразуются во внутренний код только один раз.

В настоящее время Perl является одним из языков, применяемых для программирования Интернет-приложений. Он входит в состав операционных систем Debian и FreeBSD «по умолчанию». Ряд системных программ в этих 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.

2.2.2. Первая программа на языке Perl

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

```
print "Первая программа на языке Perl\n";
exit( 0 );
```

Сохраните его под именем, например, **first.pl**. Расширение «.**pl**» не является обязательным, но, как правило, программы на языке Perl имеют

именно такое расширение. Для выполнения этой программы можно выполнить такую команду:

```
perl first.pl
```

Существует и другой способ выполнения программ, написанных на этом языке. Сначала добавьте в текст программы первую строку такого содержания:

```
#!/usr/bin/env perl
```

У вас получится

```
#!/usr/bin/env perl

print "Первая программа на языке Perl\n";
exit( 0 );
```

Теперь сделайте вашу программу исполняемым файлом. Для этого нужно назначить соответствующие права доступа к файлу программы. Это можно сделать таким образом:

```
chmod 755 first.pl
```

Теперь можно запустить эту программу более простым способом:

```
./first.pl
```

Обратите внимание на символы «./». Поскольку в UNIX-подобных ОС по умолчанию поиск исполняемых файлов (программ) в текущем каталоге не производится, то необходимо в команде запуска программы указывать символы «./», означающие текущий каталог. Если эти символы в команде не указать, то будет выведено сообщение об ошибке.

Теперь объясним назначение каждой строки этой маленькой программы. Первая строка позволяет операционной системе найти путь к интерпретатору языка Perl. Обратите внимание на символы «**#!**» в начале этой строки. Их наличие обязательно.

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, что означает успешное завершение.

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

Теперь вернемся к первой строке скрипта. В этой строке можно указать и полный путь к интерпретатору Perl. При этом нужно учитывать, что он может устанавливаться в различные каталоги (например, при наличии разных его версий). Как правило, для его размещения используются два места: **/usr/bin/perl** или **/usr/local/bin/perl**. 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
```

Чтобы точно узнать, где находится интерпретатор в вашей системе, воспользуйтесь такой командой (обратите внимание на отсутствие пробела между словами where и is):

```
whereis perl
```

Предположим, что в вашей системе интерпретатор Perl размещен в каталоге **/usr/bin**, тогда первая строка скрипта будет такой:

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

Если же Perl находится в каталоге **/usr/local/bin**, то соответственно откорректируйте первую строку скрипта.

Если операционная система не найдет интерпретатор Perl в каталоге, указанном в первой строке скрипта, например, **first.pl**, то в ответ на попытку выполнить скрипт

```
./first.pl
```

будет выведено сообщение

```
./first.pl: Command not found.
```

Символы **-w** в первой строке служат предписанием интерпретатору для проверки выполнения объявлений всех переменных в программе. В нашей программе пока нет переменных.

Создать исходный текст программы можно в редакторе joe или в любом другом текстовом редакторе, в том числе, одном из тех, которые вхо-

дят в состав графической среды (desktop environment), например, KDE или GNOME.

Узнать версию Perl можно с помощью команды

```
perl -v
```

2.2.3. 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). Переменные можно объявлять с помощью ключевого слова **my**. Оно означает, что переменная будет лексической, т. е. ее область видимости будет ограничена тем блоком операторов, в котором она объявлена. Блоком является группа операторов, ограниченная левой и правой фигурными скобками (как и в языке C). В частности, такая переменная может быть объявлена внутри процедуры. 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. Если же переменная создается без использования ключевого слова **my**, тогда она будет глобальной. Приведем несколько примеров объявлений переменных. Обратите внимание на символы **$**, **@**, **%** перед именами скалярных переменных, массивов и хеш-массивов соответственно.

Скаляры

```
my $author;              # неинициализированная переменная
my $author = "Пушкин";   # инициализированная переменная -
                         # строка
my $birth_year = 1799;   # инициализированная переменная -
                         # целое значение
my $price = 525.8;       # инициализированная переменная -
                         # числовое значение с десятичной
                         # частью (цена книги)
```

Скажем сразу, что Perl является чувствительным к регистру символов. Поэтому переменные, например, $author и $Author – это различные переменные.

Обратите внимание, что для создания литерального значения символьной строки мы использовали двойные кавычки. Но в таких случаях можно использовать и одинарные кавычки: они дадут тот же самый результат, т. е. строковую константу. Но так можно поступать не всегда.

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 $author = "Пушкин";
```

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

```
my $last_name = "Пушкин";
my $first_name = "Александр";
my $patronymic_name = "Сергеевич";

my $full_name;
$full_name = $first_name . ' ' . $patronymic_name . ' ' .
             $last_name;
```

Обратите внимание, что при формировании значения переменной $full_name, содержащей полное имя автора, мы добавляли пробелы между составными частями полного имени. Эти пробелы были представлены в виде символьной строки. При этом для формирования литерального значения символьной строки мы использовали не двойные кавычки, а одинарные. В данном случае можно было бы использовать и двойные кавычки. Но двойные кавычки имею принципиальное отличие от одинарных. Если в составе строки, заключенной в двойные кавычки, будет имя переменной, тогда вместо ее имени будет подставлено значение переменной. Поэтому полное имя автора в рассмотренном примере можно было сформировать и таким способом:

```
my $full_name;
$full_name = "$first_name $patronymic_name $last_name";
```

В этом случае можно вставить в строку не только пробелы, но и другие символы, если это необходимо. А вместо имен переменных будут подставлены их значения. В результате переменная $full_name получит такое значение:

```
Александр Сергеевич Пушкин
```

Если бы в данном примере вместо двойных кавычек мы использовали одинарные, тогда значением переменной $full_name была бы такая строка:

```
$first_name $patronymic_name $last_name
```

Символы «$» и имена переменных были бы сохранены буквально.

При выполнении операций над скалярами Perl определяет по *содержимому* переменной, как с ней поступить, т. е. считать ли переменную строковой или числовой.

Массивы

Обратите внимание на символ «@» перед именем массива).

```
my @authors = ( 'Пушкин', 'Чехов', 'Гумилев' );
my @empty_array = ();      # пустой массив
```

Для обращения к элементу массива используется тот же подход, что и в языке C. Например, для получения значения элемента с индексом 2 нужно написать так:

```
$authors[ 2 ]
```

Это значение – «Гумилев». Обратите внимание, что при обращении к элементу массива символ @ изменяется на $. Счет элементов в массиве начинается с нуля (но это правило можно изменить при необходимости). Для записи значения в массив можно просто присвоить значение элементу:

```
$authors[ 5 ] = 'Северянин';
```

Если в массиве нет элемента с номером 5 (а наш массив @authors 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

Так можно создать пустой хеш-массив (обратите внимание на символ % перед именем хеш-массива):

```
my %empty_hash = ();
```

В следующем хеш-массиве записаны сведения об авторах и их произведениях.

```
my %poems = ( 'Пушкин' => 'Евгений Онегин',
              'Гумилев' => 'Жираф',
              'Северянин' => 'Классические розы'
            );
```

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 «Пушкин», «Гумилев», «Северянин» являются ключами, а соответствующие им строки – значениями. Чтобы извлечь требуемое значение из хеш-массива, нужно знать ключ для этого значения, но не нужно знать место расположения (например, индекс в обычном массиве указывает на место расположения элемента данных). Поэтому, чтобы получить название произведения, написанного Гумилевым, мы пишем:

```
$poems{ 'Гумилев' }
```

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:

```
$poems{ 'Анненский' } = 'Среди миров';
```

Кроме трех описанных типов данных существуют и так называемые **ссылки** (references). Можно с некоторой долей условности считать их аналогом указателей в языке С.

Для создания ссылки на переменную нужно использовать символ обратной косой черты «\». Создав ссылку на переменную, можно обращаться к одной и той же структуре данных, используя разные имена. Рассмотрим пример:

```
my $author = "Пушкин";
my $author_ref = \$author;   # ref -- reference
```

Теперь выведем значение оригинальной переменной и значение, которое можно получить с помощью обращения по ссылке. Обратите внимание, что для получения значения области памяти, на которую указывает ссылка, нужно добавить еще один символ «$». И еще одна важная деталь: символы «\n», означающие перевод строки (как и в языке С), нужно записывать в двойных кавычках, а не в одинарных. В противном случае они

будут восприняты буквально и выведены как два отдельных символа «\» и «n».

```
print $author . "\n";        # будет выведено "Пушкин"
print $$author_ref . "\n";   # будет выведено "Пушкин"
```

С использованием ссылок можно создавать сложные структуры данных, например, многомерные массивы.

```
my @authors = ( ['Пушкин', 'Тургенев', 'Толстой' ],
                ['Дюма', 'Гюго' ],
                ['Байрон', 'Шекспир', 'Шоу', 'Уайльд' ]
              );
```

В этих конструкциях используются ссылки на так называемые анонимные массивы. Массив @authors состоит из ссылок на массивы. Ссылки на анонимные массивы оформляются с помощью квадратных скобок, но внешние скобки будут круглыми, поскольку @authors является массивом, а не ссылкой на массив.

Вложенные массивы могут иметь нерегулярную структуру, т. е. число элементов в массивах одного уровня иерархии может быть различным. При этом могут использоваться вложенные ссылки на анонимные массивы, Например.

```
my @authors2 = ( [ 'Россия',
                   [ 'Пушкин', 'Тургенев', 'Толстой' ]
                 ],
                 [ 'Франция',
                   [ 'Дюма', 'Гюго' ]
                 ],
                 [ 'Англия',
                   [ 'Байрон', 'Шекспир', 'Шоу', 'Уайльд' ] ]
               );

print $authors[ 1 ][ 0 ] . "\n";          # Дюма
print $authors2[ 2 ][ 1 ][ 3 ] . "\n";    # Уайльд
```

В массивах могут содержаться в качестве элементов не только другие массивы, но также хеш-массивы. В этом случае для конструирования хеш-массивов используются фигурные скобки, а в массиве будут содержаться *ссылки* на эти хеш-массивы. Например:

```
my @capitals = ( { 'Россия' => 'Москва',
                   'Франция' => 'Париж',
                   'Италия' => 'Рим'
                 },
```

```
                { 'США' => 'Вашингтон',
                  'Канада' => 'Оттава'
                },
                { 'Аргентина' => 'Буэнос-Айрес',
                  'Венесуэла' => 'Каракас',
                  'Чили' => 'Сантьяго'
                }
              );
```

Обратите внимание, что для обращения к ключу хеш-массива используется оператор «->», поскольку элементами массива являются *ссылки* на хеш-массивы.

```
print $capitals[ 0 ]->{ 'Россия' } . "\n";  # Москва
```

Однако Perl позволяет опускать оператор «->», если он стоит между смежными скобками. Тем самым синтаксис упрощается.

```
print $capitals[ 0 ]{ 'Россия' } . "\n";    # Москва
```

Еще одним видом сложных структур данных являются хеш-массивы, содержащие в качестве значений массивы. Например:

```
my %satellites = ( 'Земля' => [ 'Луна' ],
                   'Марс' => [ 'Фобос', 'Деймос' ],
                   'Юпитер' => [ 'Ио', 'Европа', 'Ганимед',
                                 'Каллисто' ]
                 );
```

Обратите внимание, что для обращения к элементу массива используется оператор «->», поскольку значениями хеш-массива являются *ссылки* на массивы.

```
print $satellites{ 'Юпитер' }->[ 2 ] . "\n"; # Ганимед
```

Однако Perl позволяет опускать оператор «->», если он стоит между смежными скобками. Тем самым синтаксис упрощается.

```
print $satellites{ 'Юпитер' }[ 2 ] . "\n";   # Ганимед
```

В качестве значений хеш-массива могут использоваться другие хеш-массивы. Например:

```
my %stars = ( 'Альтаир' => { 'constellation' => 'Орел',
                             'temperature' => 7700,
                             'distance' => 16.73
```

```
                    },
        'Сириус' => { 'constellation' => 'Большой Пес',
                      'temperature' => 9940,
                      'distance' => 8.611
                    },
        'Вега' => { 'constellation' => 'Лира',
                    'temperature' => 9602,
                    'distance' => 25.05
                  }
    );
```

Обратите внимание на оператор «->» и на возможность его не использовать.

```
print $stars{ 'Сириус' }->{ 'temperature' } . "\n"; # 9940
print $stars{ 'Сириус' }{ 'temperature' } . "\n";   # 9940
```

Для просмотра сложных структур данных можно использовать команду **x** отладчика. Подробнее об этом мы скажем в главе, посвященной отладке программ на языке Perl.

2.2.4. Операторы языка Perl

Кратко рассмотрим основные группы операторов языка Perl.

Арифметические операторы:

+ сложение;
- вычитание;
* умножение;
/ деление;
** возведение в степень;
++ инкремент;
-- декремент.

Числовое сравнение:

== равенство;
!= неравенство;
< меньше чем;
> больше чем;
<= меньше чем или равно;
>= больше чем или равно.

Есть еще оператор «<=>», который возвращает -1, 0 или 1, если левый аргумент численно меньше, равен или больше правого аргумента соответственно.

Сравнение строк:

eq равенство;

ne неравенство;
lt меньше чем;
gt больше чем;
le меньше чем или равно;
ge больше чем или равно.

Есть еще оператор «cmp», который возвращает -1, 0 или 1, если левый аргумент – строка – меньше, равен или больше правого аргумента – строки – соответственно.

Логические операции:
&& логическое «И»;
|| логическое «ИЛИ»;
! логическое отрицание.

Для операторов «&&» и «||» существуют аналоги «and» и «or», но они имеют более низкий приоритет.

Различные операторы:
= присваивание;
. конкатенация (соединение) строк;
x повторение строки;
.. формирование диапазона (список чисел).

Например, оператор повторения можно использовать таким образом:

```
print '-' x 30 . "\n";
```

В результате будет выведена строка, состоящая из 30 символов «-».

Вот простой пример использования оператора формирования диапазона чисел:

```
my @list = 1 .. 10;
```

В результате будет сформирован массив (список) чисел от 1 до 10. Вывести его можно такой командой:

```
print "@list" . "\n";
```

Многие операторы можно объединить с оператором присваивания «=». Например, вместо

```
$str = $str . "\n";
```

можно написать

```
$str .= "\n";
```

2.2.5. Организация выбора и циклов в языке Perl

Конструкции для организации выбора и циклов в языке Perl похожи на аналогичные конструкции в языке C, хотя есть и отличия.

Начнем рассмотрение с конструкции выбора **if … elsif … else**. Обратите внимание, что, в отличие от языка C, используется ключевое слово elsif, а не else if. Еще одно важное требование: фигурные скобки используются всегда, даже если в блоке всего один оператор, как в нашем примере.

```
my $score = 80;

if ( $score >= 86 )
{
  print "Оценка \"отлично\"\n";
}
elsif ( $score >= 70 )
{
  print "Оценка \"хорошо\"\n";
}
elsif ( $score >= 51 )
{
  print "Оценка \"удовлетворительно\"\n";
}
else
{
  print "Оценка \"плохо\"\n";
}
```

Запись условных операторов можно сделать более компактной, если разместить условие после оператора, связанного с ним.

```
my $planet = 'Марс';
print "Спутники: Фобос и Деймос\n" if $planet eq 'Марс';
```

Существует и отрицательная форма для условного оператора. Вместо условия if (! условие) можно написать unless (условие).

```
my $name = 'guru';
print "Неверное имя пользователя\n"
  unless $name eq 'student' or $name eq 'teacher';
```

Теперь обратимся к конструкциям для организации циклов. Начнем с цикла **for**, который организуется так же, как и на языке C. Создадим массив @authors и выведем его содержимое. Переменная $#authors хранит индекс последнего элемента массива. Обратите внимание, что переменную $i мы объявили непосредственно в заголовке цикла, поэтому она будет доступна только в рамках этого цикла.

```
my @authors = ( 'Тургенев', 'Пушкин', 'Толстой' );

for ( my $i = 0; $i <= $#authors; $i++ )
{
  print "Элемент массива номер $i равен: $authors[ $i ]\n";
}
```

Предположим, что нам требуется найти в массиве конкретное значение и, если оно найдено, выполнить некоторые операции и прекратить дальнейший поиск. Для всех значений, отличающихся от искомого, никаких действий выполнять не требуется. В таком случае могут быть полезными операторы **next** и **last**. Первый из них прерывает текущую итерацию и начинает следующую (если, конечно, условие продолжения цикла еще выполняется). Второй оператор прерывает дальнейшее выполнение цикла. Рассмотрим следующий пример.

```
for ( my $i = 0; $i <= $#authors; $i++ )
{
  next unless $authors[ $i ] eq 'Пушкин';

  print "Элемент массива номер $i равен: $authors[ $i ]\n";
  last;
}
```

Perl позволяет обойтись без счетчика цикла при обработке массива. В переменную $author помещается поочередно каждый элемент массива.

```
foreach my $author ( @authors )
{
  print "$author\n";
}
```

Можно обойтись и без явного объявления переменной для временного хранения значений элементов массива. В этом поможет встроенная переменная $_, в которую автоматически поочередно записывается элемент за элементом. Переменная $_ используется очень часто. Значение ей присваивается неявно, без прямого участия программиста. Эта переменная зачастую используется в операциях, когда происходит перебор элементов массива, чтение строк из файла и т. д. Она позволяет сократить количество переменных.

```
foreach ( @authors )
{
  print "$_\n";
}
```

Этот вариант цикла можно еще более упростить, записав его в стиле Perl:

```
print "$_\n" foreach ( @authors );
```

В циклах for удобно обрабатывать хеш-массивы. Например:

```
my %authors = ( 'Россия' => 'Пушкин',
                'Франция' => 'Дюма',
                'Англия'=> 'Байрон'
              );

foreach my $country ( keys %authors )
{
  print "$country -- $authors{ $country }\n";
}
```

В этом примере использована функция **keys**, которая предназначена для получения списка ключей хеш-массива. Обратите внимание, что при вызове этой функции не используются круглые скобки. Perl позволяет не использовать скобки, когда очевидно, какие параметры получает та или иная функция. Еще одно замечание относительно функции **keys**: массив (список) ключей, который она сформирует, будет неотсортированным. Для сортировки ключей можно воспользоваться функцией **sort**.

```
foreach my $country ( sort keys %authors )
{
  print "$country -- $authors{ $country }\n";
}
```

Мы рассмотрели цикл **for/foreach**. Цикл **while** рассмотрим в разделе, посвященном работе с файлами.

2.2.6. Встроенные функции языка Perl

Perl предлагает много встроенных функций. В документации они условно разделены на категории в соответствии со сферой применения функций. Мы только перечислим основные функции, а примеры их использования покажем в следующих разделах пособия и в примерах программ, представленных на сайте http://www.morgunov.org.

Функции для работы со скалярами или строками:
chomp – часто используется для удаления символа «\n» из строки;

index – определяет позицию первого вхождения подстроки в строке;

lc – переводит строку в нижний регистр;

lcfirst – переводит первый символ строки в нижний регистр;

length – возвращает длину строки;

rindex – определяет позицию последнего вхождения подстроки в строке;

split – разделяет строку на фрагменты в соответствии с заданными символами, служащими в качестве разделителя;

uc – переводит строку в верхний регистр;

ucfirst – переводит первый символ строки в верхний регистр.

Числовые функции:

abs – возвращает абсолютное значение аргумента;

rand – возвращает случайное число в диапазоне от нуля до величины, указанной в качестве аргумента функции;

sqrt – возвращает квадратный корень из аргумента функции.

Функции для работы с массивами и списками:

grep – для каждого элемента массива вычисляет заданное выражение и возвращает только те элементы, для которых в результате вычислений получено значение «истина»;

join – соединяет элементы списка в единую строку, вставляя между элементами указанные символы;

map – для каждого элемента списка выполняет предписанные вычисления, в результате которых может порождаться число элементов, отличное от единицы, и формирует список из результирующих элементов;

pop – удаляет из массива последний элемент, уменьшая тем самым число элементов в массиве на единицу, и возвращает значение этого элемента;

push – добавляет один или более элементов в конец массива;

reverse – упорядочивает элементы массива в обратном порядке;

shift – удаляет из массива первый элемент, уменьшая тем самым число элементов в массиве на единицу и сдвигая оставшиеся элементы к началу массива, и возвращает значение этого элемента;

sort – сортирует массив или список в соответствии с предписанным правилом сопоставления элементов;

unshift – добавляет один или более элементов в начало массива, сдвигая уже присутствующие элементы к концу массива.

Функции для работы с хеш-массивами:

delete – удаляет элемент с заданным ключом;

each – возвращает содержимое хеш-массива в виде двухэлементных списков, состоящих из ключа и значения, соответствующего этому ключу;

exists – проверяет наличие заданного ключа в хеш-массиве;

keys – возвращает все ключи хеш-массива в виде неупорядоченного списка;

values – возвращает все значения, содержащиеся в хеш-массиве, в виде неупорядоченного списка.

Функции для выполнения ввода и вывода, а также для работы с файлами и каталогами:

-X – группа функций для выполнения проверок наличия различных свойств у файлов (например, существование файла, права доступа к нему, тип файла и т. д.);

chdir – изменяет текущий каталог;

chmod – изменяет права доступа к файлам;

chown – изменяет владельца файлов или каталогов;

close – закрывает файл;

closedir – закрывает каталог, открытый с помощью функции opendir;

eof – проверяет условие достижения конца файла;

getc – возвращает следующий символ из открытого файла;

link – создает новую ссылку на файл;

mkdir – создает новый каталог;

open – открывает файл;

opendir – открывает каталог;

print – выполняет неформатированный вывод данных;

printf – выполняет форматированный вывод данных;

read – читает символы из открытого файла и записывает их в строку;

readdir – возвращает следующий элемент каталога, открытого с помощью функции opendir;

readline – читает следующую строку из открытого файла, имеющего глобальный дескриптор;

rename – переименовывает файл;

rmdir – удаляет каталог, если он пустой;

seek – устанавливает указатель в открытом файле в указанную позицию;

stat – возвращает системную информацию о файле;

symlink – создает новую символическую ссылку на файл;

tell – возвращает текущую позицию указателя в открытом файле;

truncate – укорачивает файл до заданной длины;

unlink – удаляет файлы.

Функции для управления выполнением программы:

die – формирует исключение, используется для аварийного завершения программы или процедуры и вывода сообщения об ошибке;

do – выполняет указанный Perl-скрипт;

eval – позволяет организовать обработку ошибок времени выполнения программы;

exit – немедленно завершает работу программы и возвращает заданное значение операционной системе.

Разные функции:

defined – возвращает булево значение, показывающее, отличается ли значение аргумента от неопределенного значения undef;

localtime – представляет время, возвращенное функцией time, в виде списка из девяти элементов в соответствии с местным часовым поясом;

scalar – интерпретирует аргумент этой функции в скалярном контексте;

time – возвращает число секунд, прошедшее с того момента, которое операционная система считает началом отсчета времени (обычно используется в качестве аргумента функции localtime);

undef – присваивает переменной неопределенное значение undef.

В языке Perl существуют такие понятия, как скалярный контекст и контекст списка. Если функция возвращает список, но возвращаемое значение присваивается не списку переменных, а скалярной переменной, то в нее будет записано число элементов в списке (массиве). В этом случае будет иметь место скалярный контекст. Контекст списка имеет место, когда из специфики выполняемой операции следует, что она ожидает в качестве операндов именно список (массив). Для принудительного задания скалярного контекста служит ключевое слово **scalar**.

Термины **массив** и **список** не являются тождественными. Списком может быть группа переменных, объединенных круглыми скобками. Такая группа переменных не имеет общего имени, а массив обязательно имеет имя.

2.2.7. 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.

Сначала рассмотрим примеры регулярных выражений, предназначенных для поиска тех или иных сочетаний символов в символьных строках. В качестве тестового набора данных будем использовать массив имен:

```
my @names = ( 'Andrew', 'Eugene', 'Paul', 'Jonathan',
              'Caroline', 'John', 'Antony', 'Natalia',
```

```
'Olivia', 'Ophelia', 'Roland', 'Ronald' );
```

Предположим, что нам требуется выбрать имена, начинающиеся с символов «An». Для этого воспользуемся регулярным выражением, которое позволит проверять наличие указанных символов в начале строки. Само регулярное выражение ограничивается двумя символами «/». Привязка шаблона регулярного выражения к началу строки обеспечивается символом «^». Когда этот символ стоит в такой позиции, то сам он в проверяемой строке не хранится. Оператор привязки «=~» связывает проверяемую переменную и регулярное выражение, которое к ней применяется.

```
foreach my $name ( sort @names )
{
  print "$name " if ( $name =~ /^An/ );
}
```

Теперь выберем имена, состоящие из четырех букв. Регулярное выражение станет таким (всю конструкцию цикла приводить не будем):

```
/^[A-Z][a-z]{3}$/
```

В этом выражении не только символ «^», но и символ «$» не являются хранимыми. Символ «$» означает привязку регулярного выражения к концу строки. В том имени, которое нас устроит, должно быть ровно 4 буквы, причем, первая – заглавная буква. Комбинация [A-Z] означает любую заглавную букву от A до Z, комбинация [a-z] – строчная буква от a до z, квантификатор {3} предписывает, что число строчных букв должно быть равно трем.

Выбрать имена, начинающие с «J» и завершающиеся символом «n» позволит регулярное выражение

```
/^J.+n$/
```

В нем знак «+» – это квантификатор, который означает, что символ, стоящий перед ним, может повторяться в анализируемой строке один или более раз. Символ «.» означает любой символ, в т. ч. и букву латинского алфавита.

Если нас интересуют только имена Roland и Ronald, то можно воспользоваться конструкцией (…|…) для поиска альтернативных вариантов:

```
/^Ro(lan|nal)d$/
```

Число альтернативных вариантов может быть и больше двух.

Рассмотрим следующую задачу: вычленить из даты ее элементы, т. е. день, месяц и год. При этом в качестве символов-разделителей могут использоваться такие символы: дефис, косая черта и точка.

```
my $date1 = "3-12-2017";
my $date2 = "17/1/2016";
my $date3 = "03.11.2013";
```

Для решения этой задачи нужно предложить такое регулярное выражение, которое можно было бы применить к любой из трех переменных, например, к переменной $date2:

```
$date2 =~ /^(\d{1,2})[-\/\.](\d{1,2})[-\/\.](\d{4})$/;
```

Для того чтобы учесть различные символы-разделители между элементами дат, используем выражение в квадратных скобках:

```
[-\/\.]
```

Оно представляет собой класс (группу) символов, любой из которых может располагаться в этой позиции в анализируемой строке. Обратите внимание, что перед косой чертой и точкой стоит обратная косая черта. Это нужно для того, чтобы эти *специальные* символы воспринимались буквально. Дефис должен располагаться в самом начале или в самом конце группы символов. Если расположить его иначе, то он будет восприниматься не буквально, как дефис, а как служебный символ для формирования диапазона символов. Это мы уже показывали выше в конструкциях вида [A-Z].

Здесь выражения в скобках, например, «(\d{1,2})» – это группировки символов. После вычисления регулярного выражения эти группы символов записываются во встроенные переменные языка Perl с именами $1, $2, $3 и т. д. В нашем случае таких переменных – три. Символ \d означает любую цифру, а {1,2} – квантификатор, означающий, что цифр может быть от одной до двух. Вывести отдельные элементы даты можно, например, так:

```
print "Дата $date2: день -- $1, месяц -- $2, год -- $3\n";
```

Важно помнить, что переменные $1, $2 и т. д. сохраняют свои значения только до вычисления следующего регулярного выражения.

Покажем еще один способ разделения даты на элементы. Полученные элементы будут записаны в массив.

```
my @date_elements = $date3 =~
   /^(\d{1,2})[-\/\.](\d{1,2})[-\/\.](\d{4})$/;
```

Рассмотренный способ можно модифицировать, подставив вместо массива список переменных, заключенный в круглые скобки. Каждая группа символов записывается в соответствующую переменную: первая группа – в первую переменную, вторая – во вторую и т. д.

```
my( $day, $month, $year );

( $day, $month, $year ) =
  $date3 =~ /^(\d{1,2})[-\/\.](\d{1,2})[-\/\.](\d{4})$/;
```

Теперь рассмотрим использование регулярных выражений для модифицирования символьных строк, т. е. для выполнения замен одних символов другими, для удаления и вставки символов.

Вот простой пример – удаление пробелов, которые могут находиться в начале и в конце строки.

```
my $str = '  Perl   ';

print "Длина \$str = " . length( $str ) . "\n";

$str =~ s/^\s*//;
$str =~ s/\s*$//;

print "Длина \$str = " . length( $str ) . "\n";
```

Теперь прокомментируем использованные регулярные выражения. Символ «s», стоящий перед выражением, означает замену одного фрагмента строки на другой (substitute – заменить, подставить). Все выражение состоит из двух частей: первая из них помещается между первым и вторым символами «/», а вторая – между вторым и третьим символами «/». Первая часть выражения отвечает за поиск нужной комбинации символов, а вторая – за выполнение того или иного преобразования этой комбинации.

Символы «^» и «$» привязывают шаблон регулярного выражения к началу и к концу строки соответственно.

В данном случае заменяющим фрагментом является пустая строка. Символы «\s» означают пробел или символ табуляции. Символ «*» – квантификатор. Он указывает, что количество символов пробела или табуляции может быть любым – от 0 и больше. После применения такого регулярного выражения к переменной $str она может измениться. Если в ней присутствовали пробелы, то они будут удалены, если же пробелов в начале и в конце строки не было, то значение переменной не изменится.

Рассмотрим задачу преобразования даты: из формата «день-месяц-год» в формат «год-месяц-день».

```
my $date = "03-11-2017";
my $new_date = $date;

$new_date =~
  s/^(\d{1,2})[-\/\.](\d{1,2})[-\/\.](\d{4})$/$3-$2-$1/;

print "Дата $date, новый формат - $new_date\n";
```

Здесь в первой части выражения фрагменты даты с помощью группировки (….) записываются во встроенные переменные $1, $2 и $3, а затем во второй части выражения выполняется подстановка этих переменных в другом порядке.

В завершение приведем краткую сводку основных элементов, используемых для построения регулярных выражений.

. один любой символ (в т. ч. и точка тоже);
\s пробельный символ (пробел, табуляция, новая строка);
\S непробельный символ;
\d цифра;
\D нецифровой символ;
\w «словесные» символы (a-z, A-Z, 0-9, _);
\W символы, отличные от предыдущей группы символов;
[aeiuo] любой символ из набора в квадратных скобках;
[^aeiuo] любой символ, не входящий в состав набора в квадратных скобках;
(foo|bar|baz) альтернативные варианты;
^ привязка к началу строки;
$ привязка к концу строки;
\/ символ косая черта «/»;
\. символ точка «.»;
\$ символ «$»;
\^ символ «^»;
(….) группировка символов и запись их во встроенные переменные $1, $2, $3 и т. д.

Квантификаторы:
* ноль или более предыдущих символов или групп;
+ одна или более предыдущих символов или групп;
? ноль или один предыдущий символ или группа;
{n} предыдущий символ или группа повторяется точно n раз;
{m,n} предыдущий символ или группа повторяется от m до n раз;
{m,} предыдущий символ или группа повторяется m или более раз.

2.2.8. Работа с файлами

Сначала покажем более простой способ работы с файлами. Он заключается в использовании перенаправления стандартного ввода программы. Это позволяет избежать явного открытия файлов в программе.

Для того чтобы получить какие-либо данные путем ввода их с клавиатуры, нужно сделать так:

```
my $str;

$str = <STDIN>;
chomp $str;

print "$str\n";
```

Здесь конструкция <STDIN> организует ввод с клавиатуры (которая по умолчанию является устройством стандартного ввода), а после нажатия клавиши Enter введенные символы записываются в переменную. При этом в нее будет записан и символ новой строки «\n». Как правило, он не нужен для дальнейшей работы. Чтобы удалить его из переменной, используется функция **chomp**.

Теперь рассмотрим небольшую программу. В ней используется директива **use strict**. При ее наличии Perl выполняет строгую проверку синтаксиса программы и объявлений всех переменных.

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

use strict;

my $i = 0;
my $str;

# Считываем файл построчно со стандартного ввода
# и построчно выводим его, нумеруя строки
while ( $str = <STDIN> )
{
  # В данном случае переменная $str содержит
  # целую строку файла
  print ++$i . " " . $str;
}

exit( 0 );
```

В программе используется оператор <STDIN> в условии цикла. Поэтому строки, полученные со стандартного ввода, поочередно записываются в переменную $str, а затем выводятся на стандартный вывод, т. е. на дисплей. Если мы запустим программу, указав ей файл в качестве устрой-

ства стандартного ввода (с помощью операции переадресации), тогда в цикле будут поочередно считываться строки из этого файла и выводиться на дисплей. Если же мы переадресуем и стандартный вывод в какой-нибудь файл, тогда пронумерованные строки из первого файла будут записываться в этот другой файл.

Если сохранить текст программы в файле с именем, например, **stdinput.pl**, назначить этому файлу права доступа 755, тогда вся команда могла бы выглядеть так:

```
./stdinput.pl < some_file.txt > new_file.txt
```

Здесь знаки «<» и «>» означают переадресацию стандартного ввода и вывода соответственно.

Эту программу можно упростить, используя встроенную переменную Perl с именем $_. Для этого нужно условие цикла записать так:

```
while ( <STDIN> )
```

Тогда очередная строка, прочитанная со стандартного ввода (или из файла), будет записываться неявным образом в переменную $_. В результате можно отказаться от переменной $str, а в команде вывода использовать переменную $_:

```
  print ++$i . " " . $_;
```

Для явного открытия файла в программе используется функция **open**. Типичная конструкция для открытия файла такова (он будет открыт в режиме чтения):

```
open( my $fread, "<", "input.txt" ) ||
  die "Не могу открыть файл input.txt: $!\n";
```

В качестве первого параметра функция **open** получает переменную – дескриптор файла. Вторым параметром является режим открытия файла. Предусмотрены следующие режимы:

< режим чтения;

> режим записи (если файл уже существует, то сначала он будет усечен до нулевой длины, если же файл не существует, то он будет создан);

>> режим дополнения (если файл не существует, то он будет создан);

+< режим чтения-записи;

+> режим чтения-записи (но в этом режиме, если файл существует, он будет сначала усечен до нулевой длины).

При выполнении модификаций текстовых файлов нужно учитывать, что записи в таких файлах могут иметь различную длину.

Второй и третий параметры можно объединять в один, например:

```
open( my $fread, "< input.txt" )
```

Устаревшим способом является использование в качестве имени дескриптора не переменной, а слова без кавычек (bareword). Например:

```
open( READ, "<", "input.txt" ) ||
  die "Не могу открыть файл input.txt: $!\n";
```

Функция **open** в случае успешного открытия файла возвращает ненулевое значение, а в случае возникновения ошибки – неопределенное значение (**undef**). Поэтому можно с помощью логической операции «ИЛИ» объединить вызов функции **open** и функции **die**. Эта функция выводит сообщение об ошибке и завершает работу программы. Как правило, пользовательское сообщение дополняется информацией об ошибке, которая содержится во встроенной переменной $!. Если же требуется обработать возникшую ошибку, тогда нужно использовать более сложную конструкцию с функцией **eval**. Например:

```
my $fread;

eval { open( $fread, "< input.txt" ) ||
       die "Не могу открыть файл input.txt: $!\n";
     };

if ( $@ )
{
  print $@ . "\n";
  print "Проверка открытия файла в рамках eval\n";
  exit( 0 );
}
```

Обратите внимание, что переменная $fread была объявлена вне блока eval {}. Встроенная переменная $@ содержит сообщение об ошибке, которое сформировала функция die. Если же ошибки не было, то значением переменной $@ будет пустая строка.

При запуске программы можно передать ей параметры в командной строке. Они записываются во встроенный массив @ARGV. В 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.

Теперь покажем, как открыть файл изнутри программы. Создайте текст этой программы в файле с именем, например, **openfile.pl**.

Файл **openfile.pl**

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

use strict;

my $i = 0;

# Требуем два параметра командной строки. Запись $#ARGV
# означает индекс последнего элемента массива @ARGV.
if ( $#ARGV != 1 )
{
  # Выводим сообщение об ошибке на стандартное
  # устройство ошибок.
  # Обратите внимание на отсутствие запятой после STDERR.
  print STDERR "Укажите параметры: имя входного файла " .
               "и имя результирующего\n";
  exit( 1 );    # возвращаем операционной системе значение 1
}

# Откроем файл в режиме чтения.
open( my $fread, "<", $ARGV[ 0 ] ) ||
  die "Не могу открыть файл $ARGV[ 0 ]: $!\n";

# Откроем файл в режиме записи.
open( my $fwrite, ">", $ARGV[ 1 ] ) ||
  die "Не могу открыть файл $ARGV[ 1 ]: $!\n";

# Считываем файл построчно и построчно выводим его,
# нумеруя строки.
while ( <$fread> )
{
  # Обратите внимание на отсутствие запятой
  # после переменной $fwrite.
  print $fwrite ++$i . " " . $_;
}

# Закроем файлы.
close( $fread ) ||
  die "Не могу закрыть файл $ARGV[ 0 ]: $!\n";
close( $fwrite ) ||
  die "Не могу закрыть файл $ARGV[ 1 ]: $!\n";

exit( 0 );
```

Назначьте файлу права доступа 755. Выполните программу с помощью команды

```
./openfile.pl input.txt output.txt
```

Вместо имен файлов input.txt и output.txt подставьте какие-либо осмысленные имена.

2.2.9. Процедуры языка Perl

Создание процедур рассмотрим на примере следующей задачи. Предположим, что у нас есть текстовый файл, содержащий список студентов и их оценок, полученных по нескольким учебным дисциплинам. Формат файла такой:

```
Иванов Иван Иванович  3 4 3 3 3
Петров Петр Петрович  3 3 5 4 5
Новиков Ян Викторович 4 5 5 5 4
```

Требуется вычислить средний балл для каждого студента. Нужно предложить такой способ обработки, который корректно работал бы и при числе оценок, не равном пяти.

Предполагается, что файл с данными передается программе с помощью переадресации стандартного ввода, поэтому в программе используется оператор <STDIN>, а функция **open** для явного открытия файла не используется.

Файл **avg_mark.pl**

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

use strict;

my( $student, $avg_mark ); # Ф. И. О. и средний балл

# Считываем файл построчно.
while ( <STDIN> )
{
  # Удалим символ новой строки.
  chomp;

  # Вычисляем средний балл для данного студента.
  # Процедура возвращает список из двух значений,
  # которые можно также присвоить списку переменных.
  ( $student, $avg_mark ) = calculate( $_ );

  print "$student $avg_mark\n";
}
```

```
exit( 0 );

sub calculate
{
  my $line = shift;  # получим значение параметра процедуры

  my @tmp;           # временный массив
  my $stud;          # фамилия, имя, отчество
  my $total = 0;     # общая сумма оценок

  # Функция split разделяет строку $line на фрагменты,
  # в качестве разделителя служат один или более пробелов.
  @tmp = split( /\s+/, $line );

  # Элементы с индексами 0, 1 и 2 -- фамилия, имя, отчество.
  # $#tmp -- это индекс последнего элемента.
  # 3 .. $#tmp -- это элементы с индексами от 3-го
  # до последнего.
  foreach my $mark ( @tmp[ 3 .. $#tmp ] )
  {
    # print "\$mark = $mark\n";  # тестовая печать
    $total += $mark;             # суммируем оценки
  }

  $stud = "$tmp[ 0 ] $tmp[ 1 ] $tmp[ 2 ]";

  # Процедура может возвратить список значений.
  # Среднее значение вычислим, не используя
  # дополнительной переменной.
  # scalar( @tmp ) - 3 -- это число оценок.
  return ( $stud, $total / ( scalar( @tmp ) - 3 ) );
}
```

Как видно из текста программы, для создания процедур используется ключевое слово **sub**. Важный вопрос – получение значений параметров внутри процедуры. Когда процедура вызывается и ей передаются значения параметров, то все они (будь даже параметр всего один) помещаются во встроенный массив @_, который доступен изнутри процедуры. Традиционным способом получения значений параметров внутри процедуры является использование этого массива и функции **shift**. Эта функция удаляет первый элемент из массива, переданного ей в качестве параметра, и возвращает значение этого элемента, которое можно присвоить переменной. Если же параметр ей не передан, она по умолчанию работает с массивом @_. Именно такой прием мы и видим в тексте программы.

При вычислении среднего балла использовалась конструкция scalar(@tmp). Функция **scalar** заставляет Perl интерпретировать массив в так называемом *скалярном контексте*. В таком случае результатом опера-

ции будет число элементов в этом массиве. Поэтому, отняв от общего числа элементов массива число элементов, содержащих фамилию, имя и отчество, мы получим число оценок.

Сохраните текст программы в файле с именем, например, **avg_mark.pl**. Назначьте этому файлу права доступа 755. Создайте файл с данными в соответствии с тем форматом, который был приведен в начале этого раздела. Выполните программу с помощью команды

```
./avg_mark.pl < data.txt
```

2.2.10. Вызов справки по языку Perl

На сайте http://perldoc.perl.org представлена полная документация в формате HTML по языку Perl. Она очень подробная, содержит много примеров. В ней есть и руководства по отдельным темам для тех, кто только начинает изучать этот язык.

Та же документация, но представленная в текстовом формате, доступна из командной строки по команде **man**. Для обращения к главной странице Perl выполните команду

```
man perl
```

На этой странице есть краткие описания всех остальных разделов документации. Например, для просмотра страницы с описаниями всех встроенных функций используйте команду

```
man perlfunc
```

Можно получить справку и по конкретной функции. Для этого служит команда **perldoc**. Например, описание функции **open** будет выведено по команде

```
perldoc –f open
```

Аналогичным способом можно получить описания встроенных переменных. Например, для переменных @_ и $_ это будут такие команды:

```
perldoc -v @_
```

```
perldoc -v \$_
```

Обратите внимание, что перед символом «$» стоит символ обратной комой черты. Этого требует командная оболочка операционной системы.

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
```

Версию Perl, установленную в вашей системе, можно узнать с помощью команды

```
perl -v
```

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 символьные 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.

3. Отладка и профилирование 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 отладку программ, написанных на языках C/C++ и Perl.

Программа, работающая правильно, подчас делает свою работу слишком медленно. В этом случае можно прибегнуть к профилированию программы с помощью специального инструмента – профайлера (profiler).

3.1. Языки C и 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. Давайте скомпилируем программу, рассмотренную в главе 2.

```
gcc –g -c circle.c
gcc –g -c square.c
gcc –g -c demo.c
```

В команде сборки исполняемого файла параметр **-g** не требуется:

```
gcc -o demo demo.o circle.o square.o
```

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**.

Запускаем отладчик и программу с помощью команды

```
gdb ./demo
```

Если бы скомпилировали модули программы без параметра **-g**, тогда при запуске отладчика было бы выведено примерно такое сообщение:

```
Reading symbols from ./demo...(no debugging symbols
found)...done.
```

Поскольку мы включили необходимый параметр, то сообщение будет таким:

```
Reading symbols from ./demo...done.
```

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 0x60d: file demo.c, line 14.
```

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/TEACHING.../C/demo

Breakpoint 1, main () at demo.c:14
14        printf( "Введите радиус окружности: " );
```

Выполнение программы останавливается на строке под номером 14, 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 17:

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

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=4) at circle.c:25
25        len = 2 * PI * rad;
```

Вы видите, что функция **circle_len** вызвана из файла **circle.c**. При этом в скобках показано фактическое значение параметра rad, переданное ей при вызове: оно равно 4. Номер строки относится уже к файлу **circle.c**, а не **demo.c**.

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=4) at circle.c:25
#1  0x80000663 in main () at demo.c:17
```

Выведенные строки показывают, что функция **circle_len** была вызвана со строки номер 17 программного файла **demo.c**, а в настоящий момент времени управление находится на строке 25 программного файла **circle.c**.

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

```
p len
```

Отладчик «знает» и это:

```
$1 = 25.1327419
```

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=4) at circle.c:26
0x80000663 in main () at demo.c:17
17        printf( "Длина окружности: %f\n", circle_len( rad )
);
Value returned is $3 = 15
```

Если бы в нашей функции **circle_len** были предусмотрены операторы для вывода на экран какой-либо информации, то она была бы выведена. Программа вывела на экран 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 square.c:7
```

Отладчик выполнит команду и сообщит об этом:

```
Breakpoint 2 at 0x800007bb: file square.c, line 7.
```

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=6) at square.c:11
11        area = side * side;
```

Обратите внимание, что отладчик остановился на строке 11, а не 7, хотя мы задавали точку останова на строке 7. Это объясняется тем, что на строке 11 находится первая исполняемая команда данной функции – **square_area**. Если бы мы в качестве точки останова задавали не строку, а имя функции, тогда отладчик сразу «сказал бы», что в качестве точки останова будет назначена строка 11.

```
b square.c:square_area
Breakpoint 1 at 0x7bb: file square.c, line 11.
```

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

```
[Inferior 1 (process 1793) 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.

3.2. 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. Для изучения основ работы с отладчиком воспользуемся программой **avg_mark.pl** из предыдущей главы пособия.

Запуск отладчика производится следующим образом:

```
perl -d ./avg_mark.pl < data.txt
```

Здесь мы не только вызываем отладчик, но и передаем файл с данными на стандартный ввод отлаживаемой программе.

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 организовать постраничный просмотр с помощью команды

```
| h h
```

Здесь первым символом в команде является «|», который называют символом конвейера. На клавиатуре он изображается в виде вертикальной черты с разрывом.

В среде ОС FreeBSD можно воспользоваться клавишей Scroll Lock, которая позволяет организовать «прокрутку» буфера экрана вверх и вниз.

Команда **run** в отладчике Perl отсутствует. 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 11
```

назначит точку останова на строке под номером 11 текущего программного файла.

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

```
h b
```

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

```
p $total
```

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

```
p $total=10
```

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
```

3.3. Профилирование программ на языке C

Бывают ситуации, когда программа работает недостаточно быстро. В подобных случаях нужно найти тот фрагмент кода, который вызывает наибольшую задержку выполнения программы. Инструментом, который позволяет определить время, затрачиваемое на выполнение тех или иных функций (процедур), является профилировщик (profiler). В операционных системах Debian и FreeBSD это gprof.

Мы рассмотрим только самые простые приемы использования профилировщика. Воспользуемся программой, которая суммирует значения элементов матрицы, т. е. двухмерного массива. Причем делает это двумя способами: первый способ заключается в суммировании элементов по строкам матрицы, а второй – по столбцам. На основе знания порядка хранения массивов в памяти компьютера можно обоснованно предположить, что суммирование по строкам должно выполняться быстрее, поскольку в этом случае один за другим выбираются смежные ячейки памяти. При суммировании элементов по столбцам расстояния между суммируемыми ячейками памяти гораздо больше. Здесь вступает в действие принцип так называемой пространственной локальности.

Создайте в редакторе следующую программу.

Файл **sum_arrays.c**

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

int sum_array_rows( int **a, int rows, int cols );
int sum_array_cols( int **a, int rows, int cols );

int main( int argc, char **argv )
{
```

```
  int i;
  int j;

  int **array;

  int rows = atoi( argv[ 1 ] ); /* число строк в матрице */
  int cols = atoi( argv[ 2 ] ); /* число столбцов в матрице */

  /* сумма всех элементов матрицы */
  long sum_rows = 0;  /* суммирование по строкам */
  long sum_cols = 0;  /* суммирование по столбцам */

  /* Сначала выделим память под указатели
     на будущие строки матрицы. */
  array = ( int ** )malloc( rows * sizeof( int * ) );

  /* Для каждой строки ... */
  for ( i = 0; i < rows; i++ )
  {
    /* ... выделим память. */
    array[ i ] = ( int * )malloc( cols * sizeof( int ) );

    /* Значения всех элементов строки будут равны ее номеру.*/
    for ( j = 0; j < cols; j++ )
    {
      array[ i ][ j ] = i;
    }
  }

  /* Находим сумму всех элементов матрицы. */
  /* суммирование по строкам */
  sum_rows = sum_array_rows( array, rows, cols );
  /* суммирование по столбцам */
  sum_cols = sum_array_cols( array, rows, cols );

  printf( "Сумма по строкам = %ld\n", sum_rows );
  printf( "Сумма по столбцам = %ld\n", sum_cols );

  return 0;
}


int sum_array_rows( int **a, int rows, int cols )
{
  int i;
  int j;
  int sum = 0;

  for ( i = 0; i < rows; i++ )
  {
    for ( j = 0; j < cols; j++ )
```

```
    {
      /* При суммировании идем вдоль строки. */
      sum += a[ i ][ j ];
    }
  }

  return sum;
}

int sum_array_cols( int **a, int rows, int cols )
{
  int i;
  int j;
  int sum = 0;

  for ( j = 0; j < cols; j++ )
  {
    for ( i = 0; i < rows; i++ )
    {
      /* При суммировании переходим от одной строки
         к другой, выбирая элемент из того же столбца. */
      sum += a[ i ][ j ];
    }
  }

  return sum;
}
```

ПРИМЕЧАНИЕ. В этой программе работа с памятью выполняется не совсем корректно, а именно: динамически выделенная память не освобождается. Мы поступили так намеренно, чтобы использовать эту программу для ознакомления с утилитой Valgrind, которая «умеет» находить в программах подобные ошибки.

Использование профилировщика требует специальной подготовки программы. Для этого в команду компиляции добавляется параметр **-pg**.

```
gcc -pg -o sum_arrays sum_arrays.c
```

Теперь выполните программу, задав в качестве параметров командной строки число строк и столбцов в матрице. Для того чтобы различия во времени работы функций, реализующих два способа суммирования значений элементов, были заметными, нужно задать достаточно большие значения параметров. Определить их можно опытным путем. Например:

```
./sum_arrays 5000 10000
```

После завершения работы программы в текущем каталоге появится файл **gmon.out**, содержащий информацию для профилировщика. Для про-

филировки, с выводом результатов в файл, например, **a.prof**, нужно воспользоваться утилитой **gprof**:

```
gprof sum_arrays > a.prof
```

В файле **a.prof** мы увидим отчет. Ограничимся ознакомлением лишь с его первой частью. Она имеет такой вид:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 76.08      2.64     2.64                             main
 16.71      3.22     0.58        1   580.00   580.00
sum_array_cols
  7.20      3.47     0.25        1   250.00   250.00
sum_array_rows
```

В операционной системе FreeBSD этот раздел отчета находится ближе к концу файла.

На вашем компьютере могут быть другие конкретные значения времени работы функций, но общее соотношение будет примерно таким же. Заголовки столбцов означают следующее:

– % time – процент общего времени работы программы, использованного этой функцией;

– cumulative seconds – накопленная сумма времени (в секундах), использованного этой функцией и теми функциями, которые расположены выше нее в этом отчете;

– self seconds – сумма времени (в секундах), использованного только этой функцией;

– calls – число раз, которое эта функция была вызвана;

– self ms/call – среднее время (в миллисекундах) по всем вызовам этой функции, которое программа провела в этой функции;

– total ms/call – среднее время (в миллисекундах) по всем вызовам этой функции, которое программа провела в этой функции и в ее дочерних функциях;

– name – имя функции.

Строки в этой таблице отсортированы в порядке убывания значений в столбце self seconds.

Таким образом, наша гипотеза о том, что суммирование элементов матрицы по строкам выполняется быстрее, чем суммирование по столбцам, подтвердилась.

3.4. Отладка и профилирование программ на языке C с помощью утилиты Valgrind

Установить этот набор утилит в среде ОС Debian можно с помощью следующей команды:

```
apt-get install valgrind
```

Если вы хотите установить самую последнюю версию **Valgrind**, тогда можно воспользоваться установкой из исходных текстов. Получить их можно с сайта http://valgrind.org. Последовательность шагов будет такая (символами «x» обозначены младшие цифры номера версии):

```
tar jxvf valgrind-3.x.x.tar.bz2
cd valgrind-3.x.x
./configure > conf_log 2>&1
make > make_log 2>&1
make install > make_install_log 2>&1
```

Если выполнить пробный запуск

```
valgrind ls -l
```

то окажется, что требуется еще и библиотека **libc6-dbg**. Установите и ее:

```
apt-get install libc6-dbg
```

Для установки **Valgrind** в среде ОС FreeBSD нужно сделать так:

```
cd /usr/ports/devel/valgrind
make install clean
```

Чтобы подготовить программу к профилированию с помощью **Valgrind**, нужно в команде компиляции задать параметр **-g** для включения в исполняемый файл информации для отладчика. Кроме того, рекомендуется использовать параметр **-O0** (заглавная латинская буква «O» и цифра 0). Этот параметр отключает выполнение оптимизации компилируемого кода. Если оптимизацию не отключить, тогда **Valgind** может выдавать неточные номера строк, содержащих ошибки.

Давайте скомпилируем программу **sum_arrays.c**, представленную в предыдущем разделе пособия.

```
gcc -g -O0 -o sum_arrays sum_arrays.c
```

Теперь выполним ее под контролем **Valgrind**:

```
valgrind ./sum_arrays 100 200
```

По умолчанию **Valgrind** выведет только информация об использовании памяти. Однако доступны и другие возможности для детального инспектирования программы.

```
==1104== HEAP SUMMARY:
==1104==     in use at exit: 80,400 bytes in 101 blocks
==1104==   total heap usage: 102 allocs, 1 frees, 81,424 bytes
allocated
==1104==
==1104== LEAK SUMMARY:
==1104==    definitely lost: 400 bytes in 1 blocks
==1104==    indirectly lost: 80,000 bytes in 100 blocks
==1104==      possibly lost: 0 bytes in 0 blocks
==1104==    still reachable: 0 bytes in 0 blocks
==1104==         suppressed: 0 bytes in 0 blocks
==1104== Rerun with --leak-check=full to see details of leaked
memory
```

Здесь 1104 – это ID процесса, в рамках которого выполнялась программа. Как правило, эта информация не играет роли. Сначала выводится общая информация об использовании динамической памяти (HEAP SUMMARY). Здесь говорится о том, сколько всего памяти, выделенной программе, оставалось в использовании на момент завершения программы. В разделе «LEAK SUMMARY» приводятся детальные сведения об утечках памяти, т. е. о тех ее блоках, которые были выделены программе динамически, но не были корректно освобождены с помощью соответствующих действий. Обратите внимание, что утечки, вызванные выделением памяти для указателей на указатели, показаны в подразделе «definitely lost». Утечки, вызванные выделением памяти для целых чисел, показаны в подразделе «indirectly lost».

Далее рекомендуется добавить в команду еще параметр **--leak-check=full** для получения более детальной информации.

```
valgrind --leak-check=full ./sum_arrays 100 200
```

В результате будут дополнительно выведены ведения о номерах строк в исходном тексте, на которых находятся операторы, выполняющие динамическое распределение памяти, которая впоследствии не освобождается.

Для выполнения других видов инспекций программы необходимо обратиться к документации на **Valgrind**.

В среде ОС FreeBSD эта утилита работает не так предсказуемо, как в ОС Debian. Возможны сбои и проблемы в ее работе.

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 работы оптимизированного кода 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.

Скомпилируйте программу **sum_arrays.c**, добавив в команду компиляции параметр **-O1**.

```
gcc -g -O1 -o sum_arrays sum_arrays.c
```

Параметр **-O1** указывает компилятору, что необходимо генерировать оптимизированный код.

Запустите программу под управлением отладчика:

```
gdb ./sum_arrays
```

Поскольку эта программа принимает два параметра командной строки, то их нужно ей передать. Но делается это уже *после запуска* отладчика с помощью его команды **set args**. Для удобства работы давайте зададим небольшие значения, например, 3 и 5:

```
set args 3 5
```

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

```
b main
```

Затем с помощью команды **r** (**run**) запустите программу и начните проходить ее в пошаговоом режиме с помощью команды **n (next)**. Для входа в функции, вызываемые из функции **main**, используйте команду **s (step)**.

Выполняя программу в пошаговом режиме, обратите внимание на некоторые особенности поведения программы. Например, некоторые операторы выполняются не в логическом порядке, который предписан исходным текстом программы, а в ином порядке. При этом некоторые операто-

ры могут повторяться более одного раза при отсутствии циклов. В частности, обратите внимание на следующие моменты:

– в каком порядке выполняются строки

```
int rows = atoi( argv[ 1 ] );  /* число строк в матрице */
```

и

```
int cols = atoi( argv[ 2 ] );  /* число столбцов в матрице */
```

– сколько раз выполняются строки

```
array = ( int ** )malloc( rows * sizeof( int * ) );
```

и

```
int rows = atoi( argv[ 1 ] );  /* число строк в матрице *
```

Затем откомпилируйте программу с параметром **-O2** и повторите эксперимент с отладчиком. Затем то же самое проделайте с параметром **-O3**. Он означает самый высокий уровень оптимизации.

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.

5. Создайте следующую программу:

```
#include <stdlib.h>

void func( void )
{
  int *x = malloc( 20 * sizeof( int ) );
  x[ 20 ] = 0;
}

int main( void )
{
  func();
  return 0;
}
```

С помощью утилиты **Valgrind** выявите ошибки в этой программе.

6. Существует утилита **gcov**, также предназначенная для профилирования программ. Она позволяет выяснить, сколько раз была фактически выполнена каждая строка в программе. Если программа работает медленно, то, зная частоту выполнения строк программы, можно направить усилия на оптимизацию конкретных, часто вызываемых фрагментов кода.

Самостоятельно разберитесь, как используется утилита **gcov**. Можно воспользоваться документацией на странице https://gcc.gnu.org/onlinedocs/.

4. 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 в главе 2, 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.c**, **circle.c**, **square.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 **geometry.h** находится в текущем каталоге.

Файл **Makefile**

```
#-------------------------------------------------------------
# Makefile для компиляции программы demo
# ------------------------------------------------------------

# имя исполняемого файла
PRG = demo

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

```
# исполняемый файл
OBJS = demo.o circle.o square.o

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

# Параметры компилятора:
# параметр -g означает, что в объектные модули будет
# включена информация, необходимая для отладки программы
# с помощью отладчика, например, gdb;
# параметр -c означает, что требуется только
# скомпилировать исходный файл, но создавать исполняемый
# файл не нужно.
CPARAMS = -g -c

# 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.c
# и geometry.h. Переменная $@ означает имя правила
# (target), в данном случае это demo.o.
demo.o: demo.c geometry.h
        $(CC) $(CPARAMS) demo.c -o $@
# Это пример команды, в которой используются
# дополнительные каталоги для поиска заголовочных файлов.
#       $(CC) $(CPARAMS) demo.c $(INC) -o $@

# Эти правила аналогичны предыдущему правилу, с той
# лишь разницей, что объектный модуль зависит не от
# двух файлов, а от одного.
circle.o: circle.c
```

```
        $(CC) $(CPARAMS) circle.c -o $@

square.o: square.c
        $(CC) $(CPARAMS) -c square.c -o $@

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

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

```
make
```

Можно скомпилировать и лишь один из исходных файлов, например, **circle.c**:

```
make circle.o
```

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

```
make clean
```

Контрольные вопросы и задания

1. Какой символ должен находиться в начале строки, содержащей команду, реализующую выполнение правила в make-файле?

2. Какое имя по умолчанию имеет make-файл?

3. Удалите один из объектных модулей, от которых зависит создание исполняемого модуля, например, **circle.o**. Запустите компиляцию командой **make** и посмотрите, какие команды выполняются (при выполнении команд утилита **make** выводит команды на стандартный вывод).

4. Откройте файл **geometry.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?

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

5. Система управления версиями Git

Системы управления версиями в настоящее время широко используются при создании программного обеспечения. Они облегчают коллективную разработку и повышают надежность выполнения всех операций с исходным кодом. При этом в такой системе могут обрабатываться не только файлы с исходными текстами программ, но и любые другие текстовые файлы, например, файлы в формате tex. Из таких файлов может быть сгенерирован файл в формате pdf.

Мы рассмотрим только систему Git, причем, познакомимся лишь с основными ее возможностями.

5.1. Основная терминология

Любая система управления версиями тем или иным способом ведет учет изменений файлов и позволяет вернуться к предшествующим версиям этих файлов при необходимости.

Первоначально были разработаны локальные системы управления версиями. Примером такой системы является **RCS**. Такие системы хранят в специальной базе данных все изменения файлов в специальном формате. Если эти изменения применить к текущим версиям файлов, то можно воссоздать версии файлов на более ранний момент времени.

Однако при использовании локальных систем было трудно организовать взаимодействие разработчиков. Поэтому сначала были разработаны централизованные системы контроля версий, например, **Subversion**. В таких системах на центральном (и единственном) сервере хранились все версии всех файлов проекта. Клиенты пролучают файлы из этого централизованного хранилища. Недостатком такого подхода является то, что централизованное хранилище представляет собой единственную точку отказа. В случае отказа сервера взаимодействие между разработчиками будет затруднено, т. к. обмен выполненными изменениями файлов станет невозможен.

Для устранения этого недостатка были предложены децентрализованные системы управления версиями, например, **Git**, **Mercurial**, **Bazaar**. В таких системах клиенты не просто копируют на свои компьютеры снимок файлов на какой-то определенный момент времени, они создают у себя копию всего репозитория, т. е. хранилища всех файлов, в том числе и тех, которые содержат служебную информацию. Это повышает надежность всей системы.

Файлы в системе Git могут находиться в трех основных состояниях: зафиксированном (committed), измененном (modified) и подготовленном к фиксации (staged). Зафиксированными являются файлы, уже сохраненные в локальной базе данных. К измененным относятся файлы, которые были модифицированы, но еще не были зафиксированы. Подготовленные к фик-

сации файлы – это измененные файлы, отмеченные для включения в следующий коммит (commit).

Git хранит метаданные и базу данных объектов вашего проекта в каталоге **.git**. Это самая важная часть **Git**. Рабочий каталог является снимком одной версии проекта. Файлы извлекаются из сжатой базы данных, находящейся в каталоге **.git**, и помещаются в рабочий каталог для того, чтобы их можно было использовать и изменять. Область подготовленных файлов – это специальный файл, располагающийся в вашем каталоге **.git**. В нем содержится информация о том, какие изменения попадут в следующий коммит, т. е. будут зафиксированы в следующий раз. Эту область иногда называют «индексом».

Базовый подход в работе с **Git** выглядит так.

1. Сначала нужно отредактировать файлы в вашем рабочем каталоге.

2. Затем следует добавить эти файлы в индекс. Тем самым их моментальные снимки добавляются в область файлов, подготовленных к фиксации.

3. Наконец, снимок файлов, подготовленных к фиксации (в том состоянии, в котором они находятся в этой области), сохраняется в ваш каталог **.git**.

Git имеет целый ряд положительных качеств: большинство операций в этой системе выполняются локально, не требуя связи с сервером; для проверки целостности данных используются контрольные суммы; при выполнении различных действий данные только *добавляются*, поэтому потерять что-то в **Git** довольно сложно.

5.2. Установка Git

В среде операционной системы Debian можно установить **Git** из заранее скомпилированных пакетов с помощью команды

```
apt-get install git-all
```

В среде операционной системы FreeBSD сделать то же самое можно с помощью команды

```
pkg install git
```

Если же вы хотите работать с самой последней версией, тогда можно воспользоваться установкой из исходных текстов. Получить их можно с сайта https://git-scm.com/.

Возможно, вам придется установить ряд программных пакетов, чтобы стала возможной компиляция исходных текстов Git и сопутствующей документации. Это такие пакеты:

– **autoconf** – требуется для автоматического создания скрипта configure;

– **asciidoc** – требуется для создания файлов документации;

– **zlib1g-dev** – библиотека сжатия (файлы для разработчиков);

– **gettext** – набор утилит для интернационализации и локализации программ.

Установить их, если они еще не установлены, можно с помощью следующих команд в среде ОС Debian:

```
apt-get install autoconf
apt-get install asciidoc
apt-get install zlib1g-dev
apt-get install gettext
apt-get install gettext-doc
```

Теперь можно перейти к непосредственной установке **Git**. Загрузите с сайта https://git-scm.com последнюю версию **Git** – файл **git-2.x.x.tar.xz**. В этом имени файла символы «x» означают младшие номера текущей версии **Git**. Поместите файл **git-2.x.x.tar.xz** в тот каталог, в котором вы храните исходные тексты программных пакетов и извлеките файлы из архива:

```
tar xJvf git-2.x.x.tar.xz
```

Перейдите в созданный каталог:

```
cd git-2.x.x
```

Подготовьте скрипт **configure**:

```
make configure > make_conf_log 2>&1
```

Выполните скрипт конфигурирования исходных текстов. В этой команде параметр **--prefix** указывает путь для установки **Git**. Для вывода всех сообщений в файл conf_log используется переадресация вывода. При этом сообщения об ошибках будут также выводиться в этот файл. Это обеспечивается за счет включения в команду такого фрагмента: 2>&1. Весь он записывается без пробелов.

```
./configure --prefix=/usr/local > conf_log 2>&1
```

Скомпилируйте исходные тексты и документацию, при этом сообщения также перенаправьте в файл:

```
make all doc > make_all_doc_log 2>&1 &
```

Поскольку компиляция запущена в фоновом режиме (на это указывает знак «&» в конце команды), то можно следить за ходом процесса таким образом:

```
tail -f make_all_doc_20170907
```

Если не случилось ошибок, то можно установить скомпилированные программы и документацию (следующая команда пишется целиком на одной строке):

```
make install install-doc install-html > make_install_log 2>&1
&
```

Для проверки успешности установки выполните команду

```
git
```

Должна быть выведена подсказка по использованию **Git**.

5.3. Основы использования Git

После установки **Git** необходимо выполнить первоначальную настройку, т. е. задать значения параметров конфигурации с помощью команды **git config**. Эти параметры могут сохраняться в файлах, имеющих различную сферу влияния. Если они сохраняются в файле **/etc/gitconfig**, то их значения будут общими для всех пользователей системы и для всех их репозиториев. Чтобы записать параметры именно в этот файл, нужно при запуске **git config** указать параметр --**system**.

Можно задать параметры для конкретного пользователя, если в команде **git config** указать параметр --**global**. В этом случае параметры размещаются в файле **~/.gitconfig** или **~/.config/git/config**. В этой записи символ «~» означает домашний каталог пользователя.

И, наконец, можно задать параметры для конкретного репозитория. Они хранятся в файле **.git/config** в том репозитории, который вы используете в данный момент.

Нужно обязательно задать ваше имя и адрес электронной почты, посткольку эти сведения включаются в каждый коммит (т. е. в каждое за-

фиксированное состояние проекта). Давайте зададим эти параметры для того пользователя, с учетной записью которого вы работаете в системе:

```
git config --global user.name "Your Name"
```

Обратите внимание на два символа «-» перед параметром global. Если ваше имя содержит пробелы, то нужно заключить его в двойные кавычки.

С адресом электронной почты поступим аналогично:

```
git config --global user.email yourmail@somedomain.ru
```

Зададим текстовый редактор, который будет вызываться для создания сообщений, формируемых в среде **Git**.

```
git config --global core.editor joe
```

Вы можете найти файл **.gitconfig** в домашнем каталоге пользователя и посмотреть содержимое файла.

Для просмотра значения конкретного параметра можно воспользоваться этой же командой. Например, так можно узнать имя пользователя:

```
git config user.name

Your Name
```

Полный список настроек можно вывести таким способом:

```
git config --list

user.name=Your Name
user.email=yourmail@somedomain.ru
core.editor=joe
```

Эти параметры **Git** собирает из всех созданных конфигурационных файлов: на уровне всей системы, на уровне конкретного пользователя и на уровне данного проекта.

Для получения справки по командам **Git** существует три способа. Например, для вызова справки по команде **config**, которую мы сейчас рассмотрели, эти три способа выглядят так:

```
git help config
git config --help
man git-config
```

Теперь мы можем приступить к непосредственному изучению основных приемов использования **Git**. Первым шагом является создание репозитория проекта. В качестве учебного проекта воспользуемся той программой на языке С, которую мы рассматривали в главе 2. Создайте каталог, скопируйте в него файлы программных модулей и добавьте **Makefile** для этой программы, созданный в главе 4. Перейдите в созданный каталог и выполните команду

```
git init
```

В текущем каталоге будет создан подкаталог **.git**. Именно в нем находится репозиторий проекта. Давайте посмотрим, в каком состоянии находится наш проект на данной стадии. Выполните команду

```
git status
```

Будет выведен список неотслеживаемых файлов:

```
Makefile
circle.c
demo.c
geometry.h
square.c
```

Чтобы **Git** начал отслеживать их, выполните следующую команду:

```
git add *.c *.h Makefile
```

Если снова посмотреть состояние репозитория, то мы увидим, что теперь эти файлы включены в состав отслеживаемых.

```
git status
```

```
Изменения, которые будут включены в коммит:
  (используйте «git rm --cached <файл>…», чтобы убрать
   из индекса)

        новый файл:    Makefile
        новый файл:    circle.c
        новый файл:    demo.c
        новый файл:    geometry.h
        новый файл:    square.c
```

Git зачастую не только выдает запрашиваемые сведения, но и подсказывает возможные действия пользователя в дальнейшем. В вышеприведенном выводе команды говорится, каким образом можно исключить ка-

кой-либо файл из списка отслеживаемых. Поэтому если вы ошибочно включили какой-то файл, то его всегда можно исключить из этого списка. Давайте проделаем такую операцию, например, с файлом **Makefile** (обратите внимание на двойной дефис):

```
git rm --cached Makefile
git status
```

```
Изменения, которые будут включены в коммит:
  (используйте «git rm --cached <файл>…», чтобы убрать
   из индекса)

        новый файл:    circle.c
        новый файл:    demo.c
        новый файл:    geometry.h
        новый файл:    square.c

Неотслеживаемые файлы:
  (используйте «git add <файл>…», чтобы добавить в то,
   что будет включено в коммит)

        Makefile
```

Если это новый файл или файл, исключенный из индекса, то он отражается в списке неотслеживаемых файлов. Когда вы добавляете новый файл в проект и хотите, чтобы система контроля версий его отслеживала, используйте команду **git add**. Не забудьте вернуть **Makefile** в состав отслеживаемых файлов:

```
git add Makefile
```

Теперь мы можем сделать первый коммит, т. е. сохранить в репозитории текущее состояние нашего проекта. При этом можно задать текстовое сообщение, связанное с данным коммитом.

```
git commit -m "Начальное состояние проекта"
```

Git сообщит о проделанной работе: в частности, о числе измененных файлов и числе строк текста, добавленных в репозиторий.

```
[master (корневой коммит) 5416587] Начальное состояние проекта
 5 files changed, 144 insertions(+)
 create mode 100644 Makefile
 create mode 100644 circle.c
 create mode 100644 demo.c
 create mode 100644 geometry.h
 create mode 100644 square.c
```

Если с помощью команды **git status** посмотреть состояние проекта сразу после операции commit (фиксация изменений), то **Git** сообщит, что фиксировать нечего, т. к. изменений не было.

Давайте создадим файл **README** (пусть для начала он будет даже пустым) и добавим его в состав отслеживаемых файлов, а потом посмотрим состояние проекта.

```
touch README
git add README
git status
```

```
На ветке master
Изменения, которые будут включены в коммит:
  (используйте «git reset HEAD <файл>…», чтобы убрать
   из индекса)

        новый файл:    README
```

Как видно из этого сообщения, если мы сейчас сделаем фиксацию изменений проекта (коммит), то файл **REAME** будет сохранен в очередной версии проекта в том виде, в котором он находится сейчас (т. е. пустым).

Внесите в файл **README** краткое описание вашего проекта и сохраните файл. Снова посмотрите состояние проекта:

```
git status
```

```
На ветке master
Изменения, которые будут включены в коммит:
  (используйте «git reset HEAD <файл>…», чтобы убрать
   из индекса)

        новый файл:    README

Изменения, которые не в индексе для коммита:
  (используйте «git add <файл>…», чтобы добавить файл
   в индекс)
  (используйте «git checkout -- <файл>…», чтобы отменить
   изменения в рабочем каталоге)

        изменено:      README
```

На первый взгляд, здесь налицо противоречие: один и тот же файл находится одновременно в двух состояниях. Однако противоречия здесь нет. Если бы мы сейчас зафиксировали состояние проекта (сделали коммит), то в этой версии проекта был бы сохранен пустой файл **README**, т. е. файл в том состоянии, в котором он находился на момент включения в

проект. А в рабочем каталоге находится уже измененная версия файла, что и отражено в нижней части сообщения.

Если мы сейчас выполним команду

```
git reset HEAD README
```

о которой говорится в верхней части сообщения, то этот файл станет неотслеживаемым, т. е. будет выведен из-под контроля версий. А если мы выполним команду (обратите внимание на пробелы слева и справа от удвоенного символа «-»)

```
git checkout -- README
```

то изменения файла, проведенные в рабочем каталоге, будут отменены. Если же мы решим подготовить выполненные изменения файла к фиксированию, тогда нужно еще раз выполнить команду **git add**:

```
git add README
```

Эта команда многофункциональная: она не только добавляет файлы в список отслеживаемых файлов, но и переводит их в статус готовых к фиксации. Поэтому если вы изменяли какой-то файл *после* вызова команды **git add** для него, то для подготовки к фиксации последних изменений этого файла нужно выполнить команду **git add** повторно.

Поскольку в рабочем каталоге могут появляться различные файлы, которые отслеживать и сохранять в репозитории проекта не нужно, то можно сделать так, что **Git** будет их игнорировать. К таким файлам относятся объектные модули и библиотеки, временные файлы и предыдущие версии текстовых файлов, которые создают многие редакторы. Для настройки этой функции **Git** нужно создать файл с именем **.gitignore** и включить в него шаблоны, соответствующие таким файлам. Содержимое этого файла может быть, например, таким (комментарии начинаются с символа «#»):

```
# игнорировать объектные модули и библиотеки
*.[oa]

# игнорировать старые версии текстовых файлов
*~
```

Будут игнорироваться файлы, имена которых оканчиваются на **.o** и **.a**, т. е. объектные файлы и файлы статических библиотек, а также старые версии текстовых файлов, имена которых оканчиваются символом «~». Вообще с помощью системы шаблонов можно задать очень детальные

настройки в файле **.gitignore**. Эти шаблоны представляют собой упрощенные регулярные выражения, используемые командными интерпретаторами.

Теперь настало время внести изменения в файлы исходных текстов нашей программы. Предположим, что мы решили, что функции должны не только производить вычисления, но и выводить сообщение о том, по какой формуле эти вычисления были проведены. Допустим, что это нужно сделать, *только* исходя из методических соображений, поскольку, конечно, включение в тексты функций операторов для вывода таких сообщений сделает эти функции менее универсальными: ведь в принципе возможны такие ситуации, когда выводить сообщение не требуется.

Изменения в файле **circle.c**

```
...
  /* площадь равна: пи * радиус в квадрате */
  printf( "Формула: PI * rad * rad " );
  area = PI * rad * rad;
...
  /* длина окружности равна: 2 * пи * радиус */
  printf( "Формула: 2 * PI * rad " );
  len = 2 * PI * rad;
...
```

Изменения в файле **square.c**

```
...
  /* площадь равна: длина стороны в квадрате */
  printf( "Формула: side * side " );
  area = side * side;
...
  /* периметр равен: 4 * длина стороны */
  printf( "Формула: 4 * side " );
  len = 4 * side;
...
```

Теперь посмотрим, что покажет команда

git status

```
На ветке master
Изменения, которые будут включены в коммит:
  (используйте «git reset HEAD <файл>…», чтобы убрать
   из индекса)

        новый файл:    README

Изменения, которые не в индексе для коммита:
```

```
  (используйте «git add <файл>…», чтобы добавить файл
   в индекс)
  (используйте «git checkout -- <файл>…», чтобы отменить
   изменения в рабочем каталоге)

        изменено:      circle.c
        изменено:      square.c
```

Чтобы увидеть, какие изменения мы произвели, но пока не проиндексировали (т. е. не подготовили к фиксации), нужно выполнить следующую команду:

git diff

```
diff --git a/circle.c b/circle.c
index 300e6aa..86a4640 100644
--- a/circle.c
+++ b/circle.c
@@ -11,6 +11,7 @@ float circle_area( float rad )
   float area;

   /* площадь равна: пи * радиус в квадрате */
+  printf( "Формула: PI * rad * rad " );
   area = PI * rad * rad;
   return ( area );  /* краткая запись: return ( PI * rad *
rad ); */
 }
@@ -22,6 +23,7 @@ float circle_len( float rad )
   float len;

   /* длина окружности равна: 2 * пи * радиус */
+  printf( "Формула: 2 * PI * rad " );
   len = 2 * PI * rad;
   return ( len );
 }
diff --git a/square.c b/square.c
index bd21661..96b537d 100644
--- a/square.c
+++ b/square.c
@@ -9,6 +9,7 @@ float square_area( float side )
   float area;

   /* площадь равна: длина стороны в квадрате */
+  printf( "Формула: side * side " );
   area = side * side;
   return ( area );
 }
@@ -20,6 +21,7 @@ float square_perim( float side )
   float len;
```

```
   /* периметр равен: 4 * длина стороны */
+  printf( "Формула: 4 * side " );
   len = 4 * side;
   return ( len );
 }
```

Чтобы показать контекст, в рамках которого произведены изменения, выводятся не только измененные (добавленные, удаленные) строки, но и несколько строк, предшествующих измененным строкам и следующих за ними. По умолчанию число таких строк равно трем.

В полученном выводе указываются номера строк в сопоставляемых версиях файлов. Например, следующая запись

```
@@ -11,6 +11,7 @@ float circle_area( float rad )
```

означает, что изменения произведены в функции **circle_area**. Выражения «-11,6» и «+11,7» относятся к файлу, находящемуся в индексе и в рабочем каталоге соответственно. Выражение «-11,6» говорит о том, что для файла, *находящегося в индексе,* фрагмент кода, в котором производятся изменения, начинается на строке 11 и включает в себя 6 строк (поскольку число строк контекста равно по умолчанию трем). Поскольку мы добавили одну строку (она в выводе команды показана со знаком «+» в ее начале), то для файла *в рабочем каталоге* выражение будет уже таким – «+11,7». Это говорит о том, что фрагмент кода начинается также на строке 11, но его размер составляет уже 7 строк текста.

Рассмотрим следующую запись:

```
@@ -22,6 +23,7 @@ float circle_len( float rad )
```

Обратите внимание, что соответствующие фрагменты кода начинаютсчя на строках 22 и 23. Это объясняется тем, что в файле, находящемся в рабочем каталоге, выше уже добавлена одна строка, поэтому для данного файла начало фрагмента кода, в котором производятся изменения, будет смещено на одну строку и равно 23. Длина фрагмента также будет не 6, а 7, т. к. одна строка добавлена. Она обозначена знаком «+» в начале строки.

Если бы мы удалили какую-нибудь строку, то она была бы показана в выводе команды **git diff** со знаком «-» в начале строки. Если бы строка была изменена, тогда в выводе команды **git diff** ей соответствовали бы две строки: старая версия строки со знаком «-» в ее начале и новая версия строки со знаком «+».

Добавим измененные файлы в индекс (подготовим их к фиксации изменений):

```
git add circle.c square.c
```

Снова посмотрим, что покажет команда

git status

```
На ветке master
Изменения, которые будут включены в коммит:
  (используйте «git reset HEAD <файл>…», чтобы убрать
   из индекса)

        новый файл:    README
        изменено:      circle.c
        изменено:      square.c
```

Выполним команду **git diff**, и она не выведет ничего, поскольку все изменения, произведенные в файлах, находящихся в рабочем каталоге, мы проиндексировали, т. е. подготовили к фиксации (коммиту). Поэтому файлы в рабочем каталоге и в индексе *совпадают*, а данная команда сообщает именно о *различиях*.

Давайте теперь сравним содержимое индекса и последнего коммита. Для этого в команду **git diff** нужно добавить параметр --**staged** (или его синоним --**cached**).

git diff --staged

```
diff --git a/README b/README
new file mode 100644
index 0000000..6e6de68
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+Это краткое описание программы.
diff --git a/circle.c b/circle.c
index 300e6aa..86a4640 100644
--- a/circle.c
+++ b/circle.c
@@ -11,6 +11,7 @@ float circle_area( float rad )
   float area;

   /* площадь равна: пи * радиус в квадрате */
+  printf( "Формула: PI * rad * rad " );
   area = PI * rad * rad;
   return ( area );  /* краткая запись: return ( PI * rad *
rad ); */
 }
...
```

Если бы сейчас мы выполнили фиксацию изменений, то именно в таком виде они и были бы зафиксированы в репозитории.

Давайте скомпилируем программу **demo**, выполнив команду

```
make
```

Будут выведены предупреждения, касающиеся функции **printf**. Чтобы их устранить, необходимо добавить директиву #include в файлы **circle.c** и **square.c**.

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

Для выявления различий между состояниями файлов, представленных в рабочем каталоге и в индексе, а затем между состояниями файлов, представленных в индексе и сохраненных в результате выполнения предыдущего коммита, выполните команды

```
git diff
git diff --cached
```

Вы увидите, что выводы этих двух команд будут различаться.

Если бы мы по какой-то причине захотели отменить изменения, произведенные в файлах **circle.c** и **square.c** в рабочем каталоге с момента последнего добавления этих файлов в индекс, то нужно было бы воспользоваться командами

```
git checkout -- circle.c
git checkout -- square.c
```

В данной ситуации после выполнения этих команд директивы #include были бы удалены из этих файлов.

Но мы не будем отменять изменения, а добавим измененные файлы в индекс:

```
git add circle.c square.c
```

Снова выполните команды

```
git diff
git diff --cached
```

Первая из них не выведет ничего, поскольку состояния файлов в рабочем каталоге и в индексе синхронизированы. Внимательноь изучите вывод второй команды.

Теперь снова скомпилируем программу **demo**:

```
make clean
make
```

Убедившись, что компиляция выполняется без ошибок и предупреждений, запустите программу. Если она работает без ошибок, то можно зафиксировать текущее состояние нашего проекта в репозитории.

```
git commit -m "Вторая версия программы demo"
```

При выполнении коммита **Git** сообщит о результатах:

```
[master 5c1ea89] Вторая версия программы demo
 3 files changed, 9 insertions(+)
 create mode 100644 README
```

Выполните команду

```
git status
```

Естественно, она сообщит о том, что изменений в рабочем каталоге нет, фиксировать нечего.

Контрольные вопросы и задания

1. Информацию о состоянии проекта можно получить с помощью команды **git status**. Эта команда может выводить информацию в сжатом виде. Для этого нужно использовать параметр **-s** или **--short**:

```
git status -s
git status --short
```

Самостоятельно разберитесь с тем, как нужно интерпретировать краткую информацию, выводимую командой **git status**.

2. Файл **.gitignore** содержит шаблоны имен файлов, которые **Git** должен игнорировать. А каким образом можно сделать так, чтобы сам этот файл не попадал в состав отслеживаемых файлов?

3. Выполните команду **git diff** с параметром **-U**, в котором укажите число строк контекста, например:

```
git diff -U2
```

или даже

```
git diff -U1
```

Как вы думаете, что будет выведено, если значение параметра **-U** задать равным нулю? Проверьте вашу гипотезу на практике.

4. Попробуйте удалить и изменить какие-нибудь строки в одном из файлов, а затем выполните команду **git diff**. Каким образом эти действия отражаются в выводе данной команды?

5. **Git** позволяет удалять файлы из индекса. При этом можно либо оставить файл в рабочем каталоге, либо совсем удалить его. Самостоятельно изучите эту функцию **Git**.

6. Библиотеки

Библиотеки подпрограмм являются средством ускорения процесса разработки программного обеспечения и повышения его надежности за счет повторного использования кода. Мы рассмотрим процесс создания библиотек на языке C, а также порядок создания модулей (пакетов) на языке Perl. Эти модули (пакеты) можно считать аналогом библиотек в их традиционном понимании.

6.1. Язык 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 такие виды библиотек, как статические и разделяемые. Последние можно использовать также и путем их динамической загрузки.

6.1.1. Статические библиотеки

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

Для создания библиотеки воспользуемся теми исходными модулями, которые мы разработали в главе 2. В качестве предметной области была выбрана элементарная геометрия. Мы создали два файла исходных текстов: в одном содержались функции для вычисления параметров круга (окружности), во втором – параметров квадрата. Кроме того, была создана и программа, из которой вызывались эти функции для демонстрации их работоспособности. Мы компилировали эти исходные файлы по отдельности, получив три объектных файла – **circle.o**, **square.o** и **demo.o**. А уже затем формировали исполняемый файл с помощью такой команды:

```
gcc -o demo demo.o circle.o square.o
```

Предположим, что функции для вычисления параметров круга и квадрата предполагается использовать и при разработке других программ. Тогда будет логичным включить их в библиотеку. Ее наличие облегчит повторное использование кода.

Создайте новый каталог для выполнения последующих заданий и скопируйте в него файлы **circle.c**, **square.c**, **demo.c** и **geometry.h** из каталога, созданного при изучении материала главы 2. Это должны быть первоначальные версии данных файлов, а не те, которые мы создали в процессе изучения материала главы 5.

Скажем сразу, что создаваемая нами библиотека может служить только в качестве иллюстрации технологии создания библиотек и не может рассматриваться как пример реального проекта.

Еще раз напомним, что в текстах модулей, включаемых в библиотеку, 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**.

В качестве первого шага необходимо скомпилировать исходные тексты 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 **c** означает, что файл библиотеки необходимо создать, поскольку его еще нет.

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

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**.

В случае внесения изменений в исходные тексты модулей, входящих в состав библиотеки, нужно ее переформировать.

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
00000028 T circle_len
         U _GLOBAL_OFFSET_TABLE_
00000000 T __x86.get_pc_thunk.ax

square.o:
         U _GLOBAL_OFFSET_TABLE_
00000000 T square_area
0000001e T square_perim
00000000 T __x86.get_pc_thunk.ax
```

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.

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

Теперь мы можем скомпилировать тестовую программу, использующую созданную библиотеку:

```
gcc -o demo_static_lib demo.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**. Поиск библиотек по умолчанию производится в ряде системных каталогов. После проведенных перемещений файлов команда для компиляции программы **demo_lib.c** изменится:

```
gcc -o demo_static_lib demo.c -lgeometry
```

Если по какой-то причине потребуется указать полные пути к каталогу, в котором находятся библиотечные файлы, и к каталогу, содержащему заголовочные файлы, тогда следует воспользоваться параметрами **-L** и **-I** соответственно. Вышеприведенная команда в таком – развернутом – варианте будет выглядеть так (вводить эту команду необходимо в одной командной строке, не нажимая клавишу Enter, пока не введена вся команда):

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

В этом варианте команды компиляции тестовой программы параметр **-L/usr/local/lib** указывает на тот каталог, в который вы перенесли библиотеку, а параметр -I/**usr/local/include** указывает компилятору каталог для поиска заголовочных (включаемых) файлов.

Предположим, что наша библиотека будет развиваться и в будущем потребуется несколько заголовочных файлов. Тогда целесообразно создать для них подкаталог с именем, например, **geometry**. Но если мы поместим файл **geometry.h** в подкаталог **geometry**, то в этом случае будет необходимо в тексте программы **demo.c** изменить имя файла в директиве #include таким образом:

```
#include "geometry/geometry.h"
```

Добавим еще, что если бы вы поместили библиотеку **libgeometry.a** в системный каталог **/lib** или **/usr/lib**, то параметр **-L** компилятору также был

бы не нужен. Однако следует учитывать, что в этих каталогах традиционно находятся только библиотеки, входящие в состав операционной системы.

6.1.2. 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 включается (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 той области памяти, 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.

Создайте новый каталог для выполнения последующих заданий и скопируйте в него файлы **circle.c**, **square.c**, **demo.c** и **geometry.h** из каталога, созданного при изучении материала главы 2.

Сначала сомпилируем исходные модули для библиотеки:

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

Включаем скомпилированные модули в библиотеку **libgeometry.so** (команду нужно вводить в виде одной строки):

```
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 них – 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 слу-

чае это имя **libgeometry.so.1**. Обратите внимание, что в команде создания библиотеки параметр **-Wl** является параметром компоновщика (здесь после буквы W идет строчная буква L). Все выражение **-Wl,-soname=libgeometry.so.1** не должно содержать 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.

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.c,** 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, выглядит так:

```
gcc -o demo_shared_lib demo.c -L. -lgeometry
```

Обратите внимание, что с параметром **-L** связан символ «.». Это означает текущий каталог.

Если попытаться запустить созданную программу, то мы получим сообщение об ошибке:

```
./demo_shared_lib
```

```
./demo_shared_lib: error while loading shared libraries:
 libgeometry.so.1: cannot open shared object file: No such
file or directory
```

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:

```
        linux-gate.so.1 (0xb77b5000)
        libgeometry.so.1 => not found
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6
(0xb75d7000)
        /lib/ld-linux.so.2 (0xb77b7000)
```

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
```

Переменная среды LD_LIBRARY_PATH создается только на время выполнения программы **demo_shared_lib**. В этой переменной содержится имя каталога, в котором динамический загрузчик должен искать разделяемую библиотеку.

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:

```
        linux-gate.so.1 (0xb7738000)
        libgeometry.so.1 => /usr/lib/libgeometry.so.1
(0xb770f000)
```

```
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6
(0xb7557000)
        /lib/ld-linux.so.2 (0xb773a000)
```

Сейчас для запуска программы достаточно лишь ввести ее имя:

```
./demo_shared_lib
```

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

```
gcc -o demo_shared_lib demo.c -lgeometry
```

Предположим, что мы захотели улучшить нашу библиотеку и с этой целью решили создать новые версии обоих модулей, из которых состоит библиотека. Поскольку в главе 5 мы уже вносили изменения в эти исходные файлы, то сейчас сможем воспользоваться ими. Напомним, что изменения заключались в том, что функции стали не только возвращать результат вычислений, но и выводили формулу, по которой эти вычисления выполнялись.

Для продолжения работы создайте новый каталог и скопируйте в него файлы **circle.c**, **square.c** и **demo.c** из того каталога, который вы создавали при изучении материала главы 5. Заголовочный файл **geometry.h** скопируйте в каталог **/usr/local/include**. Можно скопировать его и в каталог **/usr/local/include/geometry**, если вы его создавали. Но в таком случае 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.

Давайте скомпилируем новые версии исходных модулей для библиотеки:

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

Создадим новую версию разделяемой библиотеки (вся команда вводится на одной строке):

```
gcc -shared -Wl,-soname=libgeometry.so.2
-o libgeometry.so.2.0.1 circle.o square.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
```

Скомпилируем программу **demo.c** с новой версией библиотеки:

```
gcc -o demo_shared_lib2 demo.c -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_lib2
```

```
        linux-gate.so.1 (0xb76fa000)
        libgeometry.so.2 => /usr/lib/libgeometry.so.2
(0xb76d1000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6
(0xb7519000)
        /lib/ld-linux.so.2 (0xb76fc000)
```

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 обновленных модулей **circle.c** и **square.c** приводить не будем. Приведем лишь старую и новую версии измененных строк.

Фрагмент файла **circle.c**

```
/* printf( "Формула: PI * rad * rad " ); */
printf( "Формула: PI * радиус * радиус " );

/* длина окружности равна: 2 * пи * радиус */
printf( "Формула: 2 * пи * радиус " );
```

Фрагмент файла square**.c**

```
  /* printf( "Формула: side * side " ); */
  printf( "Формула: сторона * сторона " );

  /* printf( "Формула: 4 * side " ); */
  printf( "Формула: 4 * сторона " );
```

Скомпилируем новые версии исходных модулей для библиотеки:

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

Создадим новую версию библиотеки, присвоим ей номер 2.1.1:

```
gcc -shared -Wl,-soname=libgeometry.so.2
-o libgeometry.so.2.1.1 circle.o square.o
```

Обратите внимание, что так называемое имя **soname** не изменилось, хотя имя файла библиотеки стало другим. Младший номер версии, который был равен 0, 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**:

```
cd /usr/lib
ln -sf libgeometry.so.2.1.1 libgeometry.so.2
```

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

```
ls -l *geom*
```

```
lrwxrwxrwx 1 root root   16 окт  7 18:10 libgeometry.so ->
libgeometry.so.2
lrwxrwxrwx 1 root root   20 окт  7 11:16 libgeometry.so.1 ->
libgeometry.so.1.0.1
-rwxr-xr-x 1 root root 7052 окт  7 11:00 libgeometry.so.1.0.1
lrwxrwxrwx 1 root root   20 окт  7 18:37 libgeometry.so.2 ->
libgeometry.so.2.1.1
-rwxr-xr-x 1 root root 7156 окт  7 18:05 libgeometry.so.2.0.1
-rwxr-xr-x 1 root root 7156 окт  7 18:35 libgeometry.so.2.1.1
```

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

лиотечные функции незначительные изменения, касающиеся формата вывода формул.

Перейдите в каталог, в котором находится программа **demo_shared_lib**, и выполните ее. Вы увидите, что формулы на экран не выводятся. Это значит, что данная программа использует первую версию разделяемой библиотеки, как и должно быть при наличии в системе нескольких версий библиотеки. Это объясняется тем, что в момент компиляции программы **demo_shared_lib** ссылка **libgeometry.so** указывала на ссылку **libgeometry.so.1**, а та, в свою очередь, – на файл библиотеки **libgeometry.so.1.0.1**. В специальное поле программы было записано имя **libgeometry.so.1**, поэтому при запуске программы **demo_shared_lib** динамический загрузчик может найти библиотеку **libgeometry.so.1.0.1**.

В момент компиляции программы **demo_shared_lib2** ссылка **libgeometry.so** указывала на ссылку **libgeometry.so.2**, а та, в свою очередь, – на файл библиотеки **libgeometry.so.2.0.1**. В специальное поле программы было записано имя **libgeometry.so.2**. Но, поскольку мы перенаправили ссылку **libgeometry.so.2** на файл библиотеки **libgeometry.so.2.1.1**, то при запуске программы **demo_shared_lib2** динамический загрузчик загружает обновленную версию библиотеки **libgeometry.so.2.1.1**.

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

6.1.3. 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 разделяемых 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 познако-

миться 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.

Файл **demo_dl_lib.c**

```
/* ------------------------------------------------------------
   Программа, иллюстрирующая использование динамической
   загрузки разделяемой библиотеки.
------------------------------------------------------------ */

#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 "geometry.h" */

/* Определим указатель на функцию, принимающую один параметр
   и возвращающую значение типа float
   (таковы все функции из нашей библиотеки). */
typedef float ( *library_func )( float );

/* 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.
```

```
                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 ) );

/* 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 ) );

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 ) );

if ( ( lib_fn = get_func( lib, "square_area" ) ) != NULL )
```

```
    printf( "Площадь квадрата: %f\n", ( *lib_fn )( side ) );

  /* 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();

  /* Получим адрес требуемой библиотечной функции. */
  lib_func = dlsym( lib_handle, func_name );

  /* Обратите внимание, что здесь используется оператор
     присваивания, а не проверки на равенство. */
  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 включать имя этой библотеки в команду

компиляции не нужно, однако при работе в среде ОС Debian в нее нужно добавить библиотеку **libdl.so**:

```
gcc -o demo_dl_lib demo_dl_lib.c -ldl
```

В среде ОС FreeBSD функции **dlopen**, **dlclose**, **dlsym**, **dlerror** входят в состав стандартной библиотеки, поэтому добавлять параметр -ldl в команду компиляции не нужно.

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

```
ldd demo_dl_lib
```

На экран будет выведено следующее:

```
        linux-gate.so.1 (0xb7759000)
        libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2
(0xb772e000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6
(0xb7576000)
        /lib/ld-linux.so.2 (0xb775b000)
```

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.

6.2. 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 С 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, содержа-

щаяся в файле модуля, станет доступной в вызывающей программе при использовании в ней специальной директивы **use**.

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

Файл **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.

Файл **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.

Файл **INC.pl**

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

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

```
exit 0;
```

Эта программа в среде ОС Debian выведет примерно следующее:

```
/etc/perl
/usr/local/lib/i386-linux-gnu/perl/5.24.1
/usr/local/share/perl/5.24.1
/usr/lib/i386-linux-gnu/perl5/5.24
/usr/share/perl5
/usr/lib/i386-linux-gnu/perl/5.24
/usr/share/perl/5.24
/usr/local/lib/site_perl
/usr/lib/i386-linux-gnu/perl-base
```

В среде ОС FreeBSD будет выведено примерно следующее:

```
/usr/local/lib/perl5/site_perl/mach/5.24
/usr/local/lib/perl5/site_perl
/usr/local/lib/perl5/5.24/mach
/usr/local/lib/perl5/5.24
.
```

В последней строке выводится символ «.», что означает текущий каталог).

Создайте подкаталог **Math** в каталоге, например, **/usr/local/lib/i386-linux-gnu/perl/5.24.1** в среде ОС Debian (или **/usr/local/lib/perl5/site_perl** в среде ОС FreeBSD), 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:

```
chmod 755 Geometry.pm
chmod 755 demo_module.pl
```

ПРИМЕЧАНИЕ. Не забывайте, что в UNIX-подобной операционной системе (в нашем пособии это Debian и 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
```

При отладке программ, использующих модули, подключаемые с помощью директивы use, можно назначить в качестве точки останова какую-либо процедуру, находящуюся в одном из них. Например, для назначения точки останова на процедуру **square_area**, находящуюся в модуле **Math::Geometry**, нужно сделать так (не забывайте о регистре символов):

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

Контрольные вопросы и задания

1. Чем разделаемая библиотека отличается от статической библиотеки?

2. Программа-библиотекарь **ar** позволяет удалять отдельные модули из библиотеки и добавлять новые модули в уже существующую библиотеку. С помощью электронного руководства **man** научитесь это делать. Например, удалите модуль **circle.o** из библиотеки **libgeometry.a**, затем модифицируйте файл **circle.c**, скомпилируйте его и добавьте в библиотеку обновленный модуль **circle.o**.

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

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

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

6. В программе **demo_dl_lib.c** передайте функции **dlopen** в качестве параметра неверное имя библиотеки, например, **libgeometry10.so**. Откомпилируйте и запустите программу. Вы увидите, что при компиляции никаких ошибок не возникает. Ошибка возникает на стадии исполнения программы.

7. Поэкспериментируйте с программой **demo_dl_lib.c**. Например, последуйте рекомендациям, приведенным в связи с использованием функции **dlopen**, и внимательно изучите сообщения, выводимые программой на экран.

8. В программе **demo_dl_lib.c** есть вызов функции **dlopen**. Вторым ее параметром является RTLD_LAZY. С помощью электронного руководства **man** выясните, что означает этот параметр.

9. Добавьте в библиотеку **libgeometry** свой модуль, содержащий две функции. Это могут быть, например, функции для вычисления площади поверхности цилиндра и его объема. Эти функции получают по два параметра, в отличие от тех функций, которые мы рассмотрели в тексте главы. Модифицируйте программу **demo_dl_lib.c** таким образом, чтобы динамическая загрузка новых функций также могла успешно выполняться.

10. 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?

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

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

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 Debian (а также 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, универсализация, а локализация – это конкретизация, подгонка программного продукта под нужды 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 FreeBSD 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). В настоящее время все шире используется стандарт Unicode. В операционной системе Debian по умолчанию используется кодировка символов UTF-8.

Предположим, что мы используем какую-либо программу, имеющую русскоязычный интерфейс, в среде FreeBSD и в среде Windows. В этом случае нам придется выполнить перекодирование символьных строк, содержащихся в программе, в кодировку KOI8-R для работы в среде FreeBSD 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. Как в среде ОС Debian, так и в среде ОС FreeBSD эти библиотека и утилита установлены по умолчанию.

В качестве примера использования утилиты **iconv** рассмотрим перекодирование текстового файла, например, **hello.c**, представленного в кодировке KOI8-R, в кодировку UTF-8:

```
iconv -f KOI8-R –t UTF-8 hello.c > hello.c.utf
```

Обратите внимание на то, что новый, перекодированный, файл направляется на стандартный вывод и поэтому его можно перенаправить в файл.

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 закодиро-

вать 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 пакета **gettext**. Если он отсутствует в вашей системе Debian, то установите его и сопутствующую документацию с помощью команд

```
apt-get install gettext
apt-get install gettext-doc
```

В операционной системе FreeBSD этот пакет должен быть установлен по умолчанию.

Мы покажем использование пакета **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.

Файл **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 среды LANG, которая, например, в системе FreeBSD назначается в файле **/etc/profile**). Функция **bindtextdomain** указывает базовый каталог для домена текстовых сообщений. Как правило, каждая программа имеет свой домен сообщений. В нашем примере в качестве базового установлен текущий каталог, в котором находится программа. Однако можно использовать и системный каталог, указанный в том варианте вызова этой функции, который закомментирован. Функция **textdomain** назначает текущий домен, т. е. тот домен текстовых сообщений, из которого будут выбираться сообщения, выводимые нашей программой. Точнее говоря, это тот домен, к которому относятся все последующие вызовы функции **gettext**. Удобно, чтобы имя домена совпадало с именем программы.

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:

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

В среде ОС FreeBSD эта команда будет выглядеть так (она 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. В среде ОС Debian команда будет такой:

```
msginit -l ru_RU.UTF-8 -o hello_ru.po -i hello.pot
```

А в среде ОС FreeBSD – такой:

```
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.UTF-8 (или ru_RU.KOI8-R) символы ru означают язык (русский), символы RU – страну (Россия), UTF-8 или 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: ...\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.

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. Ознакомьтесь с содержимым каталога **/usr/share/locale**. Найдите в нем подкаталоги для русского языка. Посмотрите содержимое каждого из этих подкаталогов.

4. Выполните команду создания POT-файла в такой форме:

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

т. е. без указания кодировки – UTF-8 или KOI8-R, и сравните результирующий файл с тем, который был получен при наличии этого параметра.

5. Выполните локализацию программы **hello.c** для немецкого языка.

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.

Для ОС Debian:

```
msginit -l de_DE.UTF-8 -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 ./de_DE/LC_MESSAGES
```

Для ОС FreeBSD:

```
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 ./de_DE/LC_MESSAGES
```

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

Проверьте, установлены ли в вашей системе параметры локализации для de_DE:

```
locale -a
```

```
C
C.UTF-8
POSIX
ru_RU.utf8
```

Как видно, параметров локализации для немецкого языка нет. Установите эти параметры:

```
localedef -f UTF-8 -i de_DE de_DE.UTF-8
```

Снова проверьте, что получилось:

```
locale -a
```

```
C
C.UTF-8
de_DE.utf8
POSIX
ru_RU.utf8
```

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

Для ОС Debian:

```
LANG=de_DE.UTF-8; export LANG
```

Для ОС FreeBSD:

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

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

```
setenv LANG de_DE.UTF-8
```

Для ОС FreeBSD:

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

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

```
LANG=de_DE.UTF-8
LANGUAGE=
LC_CTYPE="de_DE.UTF-8"
LC_NUMERIC="de_DE.UTF-8"
LC_TIME="de_DE.UTF-8"
LC_COLLATE="de_DE.UTF-8"
LC_MONETARY="de_DE.UTF-8"
LC_MESSAGES="de_DE.UTF-8"
```

```
LC_PAPER="de_DE.UTF-8"
LC_NAME="de_DE.UTF-8"
LC_ADDRESS="de_DE.UTF-8"
LC_TELEPHONE="de_DE.UTF-8"
LC_MEASUREMENT="de_DE.UTF-8"
LC_IDENTIFICATION="de_DE.UTF-8"
LC_ALL=
```

Для ОС FreeBSD:

```
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 же в вашей системе не установлены параметры локализации для немецкого языка, то вы увидите такие сообщения:

```
locale: Cannot set LC_CTYPE to default locale: No such file or
directory
locale: Cannot set LC_MESSAGES to default locale: No such file
or directory
locale: Cannot set LC_ALL to default locale: No such file or
directory
```

Если необходимая 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. Как в операционной системе Debian, так и в 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

Настоящая глава представляет собой элементарное введение в технологии баз данных. Для получения основательной подготовки в этой области 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 (как правило, 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 прикладными 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 эпоху, предшествующую рождению реляционной теории, базы данных традиционно рассматривались, как набор **файлов**, состоящих из **записей**, а записи, в свою очередь, подразделялись на отдельные **поля**. Поле являлось элементарной единицей данных.

В настоящее время преобладают базы данных реляционного типа. Их характерной чертой является тот факт, что данные воспринимаются пользователем как таблицы. Поэтому термину «файл» соответствует термин «таблица», вместо термина «запись» используется термин «строка», а вместо термина «поле» – термин «столбец» (или «колонка»). Таким образом, таблицы состоят из строк и столбцов, на пересечении которых должны находиться «атомарные» значения, которые нельзя разбить на более мелкие элементы без потери смысла.

В теории баз данных эти таблицы называют **отношениями (relations)** – 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 (tuples)**, 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**.

Таким образом, в теории и практике баз данных существует три группы терминов. Иногда термины из разных групп используют в качестве синонимов, например, запись и строка.

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	2017/2018	1	5
55500	Математика	2017/2018	1	4

55800	Физика	2017/2018	1	4
55800	Физика	2017/2018	2	5

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 (referenced table)». 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 (referencing table), а таблица, содержащая соот-

ветствующий потенциальный ключ, – **ссылочной (целевой)** таблицей (referenced table).

Для обеспечения ограничений ссылочной целостности применяются специальные способы проектирования базы данных. Может предусматриваться, 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 отсутствие 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 и первоначальная настройка

В этом разделе речь пойдет об установке PostgreSQL в среде операционных систем Debian и FreeBSD.

Устанавливать рекомендуется последнюю **стабильную** версию СУБД. На данном «историческом» отрезке времени она имеет номер 10.x. Здесь символом «x» обозначен номер выпуска, который может изменяться при внесении изменений и исправлений в стабильную версию.

Установить СУБД можно как из заранее скомпилированных (бинарных) пакетов, так и из исходных текстов. Перед установкой из бинарных пакетов в среде ОС Debian сначала выясните, какая версия СУБД содержится в репозитории:

```
apt-cache search postgresql|grep "^postgresql"
```

Выяснив номер версии, можно приступить к непосредственной установке СУБД из бинарных пакетов в среде ОС Debian с помощью следующей команды (вместо символов «x» подставьте конкретные значения)

```
apt-get install postgresql-x.x
```

Эта команда позволит установить только непосредственно сервер СУБД, а дополнительные подсистемы нужно устанавливать отдельно. Например, команда для установки клиентских утилит такова:

```
apt-get install postgresql-client-x.x
```

Можно воспользоваться и метапакетом, который включает все доступные серверные пакеты СУБД PostgreSQL:

```
apt-get install postgresql-all
```

В среде ОС FreeBSD существует так называемая коллекция портов (ports and packages collection). Для установки СУБД можно воспользоваться ее возможностями. Подробнее об установке СУБД из бинарных пакетов, как в ОС Debian, так и FreeBSD, смотрите на сайте http://www.postgresql.org в разделе «Download».

Теперь мы подробно рассмотрим процедуру установки СУБД PostgreSQL из исходных текстов в среде операционных систем Debian и FreeBSD. Загрузить исходные тексты можно с сайта http://www.postgresql.org. На этом сайте пройдите по ссылке «Download» и на этой странице в разделе «Source code» откройте ссылку «file browser», а в открывшемся списке каталогов выберите тот, который соответствует самой последней версии СУБД. Войдите в этот каталог и в нем выберите для загрузки последнюю **стабильную** версию архивного файла с именем вида **postgresql-10.x.tar.gz** или **postgresql-10.x.tar.bz2**. Если к тому моменту, когда вы будете устанавливать PostgreSQL, выйдет версия 11, то учтите, что имена архивных файлов изменятся соответствующим образом.

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-10.x**:

```
cp postgresql-10.x.tar.gz /usr/src
cd /usr/src
tar xzvf postgresql-10.x.tar.gz
cd postgresql-10.x
```

либо

```
cp postgresql-10.x.tar.bz2 /usr/src
cd /usr/src
bzcat postgresql-10.x.tar.bz2 | tar xvf -
cd postgresql-10.x
```

Обратите внимание, что в команде извлечения файлов из архива последний символ – дефис. Он означает, что команда tar читает данные не из файла, а со стандартного ввода. Команду извлечения файлов из архива tar.bz2 можно и заменить такой:

```
tar xjvf postgresql-10.x.tar.bz2
```

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.

Список всех параметров, принимаемых программой **configure**, можно получить с помощью следующей команды (обратите внимание на два дефиса перед параметром **help**):

```
./configure --help
```

Из множества параметров мы опишем лишь некоторые, которые представляют, на наш взгляд, наибольший интерес. Параметр --**enable-nls** позволяет всем программам, входящим в комплект СУБД, выводить сообщения на том языке, который установлен параметрами локализации операционной системы. Например, в ОС Debian это будет, скорее всего, «ru_RU.UTF-8», а в ОС FreeBSD – «ru_RU.KOI8-R». Сообщения будут выводиться на русском языке. Посмотреть эти параметры можно в обеих системах с помощью команды

```
locale
```

СУБД PostgreSQL позволяет разрабатывать хранимые процедуры на различных языках, в том числе и на языке Perl. Для реализации этой возможности служит параметр --with-perl.

Если у вас возникнет желание изучить работу СУБД на уровне исходных текстов, то мы рекомендуем вам скомпилировать их 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 присвоения переменной CFLAGS, создаваемой в среде операционной системы, значения **-O0** (заглавная латинская буква «О» и цифра 0), которое предписывает компилятору генерировать неоптимизированный код. Изучать в отладчике оптимизированный код гораздо сложнее.

Все программы СУБД PostgreeSQL будут по умолчанию установлены в каталог **/usr/local/pgsql**.

Для выполнения поставленной задачи введите команду (она вводится на одной строке):

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

Все сообщения, выводимые в процессе конфигурирования, мы перенаправим 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 (обратите внимание, что в этом компоненте нет пробелов). Последний знак & в командной строке означает, что команда будет выполняться в фоновом режиме. Поэтому вы можете наблюдать за процессом 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, просто не вводя знак & в самом конце команды.

По окончании процедуры конфигурирования загляните в файл **conf_log.txt**. Если в конце файла нет сообщения об ошибке, то все в порядке. Кроме того, программа **configure** сама создает файлы-журналы в текущем каталоге: **config.log** и **config.status**. Однако они не являются точной копией файла **conf_log.txt**.

Процесс конфигурирования может завершиться с ошибкой. В этом случае сообщение об ошибке вы увидите в самом конце файла **conf_log.txt**. Если оно гласит, что не найдена какая-нибудь библиотека, тогда нужно приостановить процесс установки PostgreSQL и разобраться, почему про-

грамма конфигурирования не нашла библиотеку. Возможны две причины: библиотека в вашей операционной системе установлена, но находится в нестандартном месте, а вторая причина – библиотека не установлена вообще. В первом случае может помочь использование параметра **--with-libraries** (или его синонима **--with-libs**), а во втором случае придется сначала установить эту недостающую библиотеку, а потом повторить команду **configure**. Вообще, если программа **configure** не смогла найти требуемые ей файлы, посмотрите с помощью команды

```
./configure --help
```

нет ли параметра, отвечающего за эти файлы.

В ОС Debian вам, вероятно, придется установить библиотеки readline, zlib и gettext. Можно использовать такие команды:

```
apt-get install libreadline-dev
apt-get install zlib1g-dev
apt-get install gettext
```

В ОС FreeBSD эти библиотеки, как правило, уже установлены. Проверить это можно с помощью команды **pkg**:

```
pkg info readline
```

Если какой-то из библиотек нет, то ее можно установить, например:

```
pkg install readline
```

Возможно также, что пакет в вашей системе уже установлен, но утилита **configure** не может его обнаружить. В этом случае, возможно, потребуется использовать не только параметры **--with-libs**, но и **--with-includes**. В результате команда конфигурирования может стать, например, такой (она вводится на одной строке):

```
./configure --enable-nls --enable-debug
--with-libs=/usr/local/lib --with-includes=/usr/local/include
CFLAGS=-O0 > conf_log.txt 2>&1 &
```

2. Следующий этап – компиляция программ. Для этой цели нужно использовать только утилиту **GNU make**. Если она не установлена в вашей операционной системе, то установите ее, загрузив исходные тексты с сайта http://www.gnu.org. В операционной системе Debian по умолчанию используется именно такая программа **make**. А в ОС FreeBSD ее нужно установить. Она устанавливается в каталог **/usr/local/bin**. Поскольку «родная»

утилита **make** находится в каталоге **/usr/bin**, то утилиту **GNU make** в каталоге **/usr/local/bin** нужно для удобства использования переименовать в **gmake**.

Команда для компиляции исходных текстов в среде ОС Debian будет такой:

```
make world > make_log.txt 2>&1 &
```

Для слежения за ходом процесса используйте команду

```
tail -f make_log.txt
```

В команде для ОС FreeBSD нужно **make** заменить на **gmake**.

Процесс может занять от двух-трех минут до получаса, в зависимости от производительности вашего компьютера. Последняя строка, выведенная в файл-журнал, должна быть такой:

```
PostgreSQL, contrib, and documentation successfully made.
Ready to install.
```

3. Теперь нужно установить скомпилированные программы в системные каталоги операционной системы. Команда для ОС Debian будет такой:

```
make install-world > make_install_log.txt 2>&1 &
```

Для слежения за ходом процесса используйте команду

```
tail -f make_install_log.txt
```

В команде для ОС FreeBSD нужно **make** заменить на **gmake**.

Последняя строка, выведенная в файл-журнал, должна быть такой:

```
PostgreSQL, contrib, and documentation installation complete.
```

4. Для удобства использования утилит, входящих в комплект СУБД, рекомендуется добавить путь к этим утилитам в переменную среды PATH. Если вы используете командный процессор **Bourne shell (/bin/sh)** или другой, совместимый с ним (например, **/bin/bash**), то нужно поступить так. Если вы хотите, чтобы изменения были применены для всех пользователей системы, то корректировать нужно файл **/etc/profile**, а если необходимо сделать изменения только для конкретного пользователя, то нужно коррек-

тировать файл ~/.profile в домашнем каталоге пользователя (знак **~** означает домашний каталог пользователя). Перед строкой

```
export PATH
```

нужно добавить строку

```
PATH=/usr/local/pgsql/bin:$PATH
```

Если в выбранном вами файле **(~/.profile** в домашнем каталоге пользователя или **/etc/profile**) вообще нет команды **export PATH**, тогда добавьте туда обе команды:

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

Если вы используете **csh** или **tcsh**, тогда команда будет такой (изменения нужно внести в файл **~/.cshrc**):

```
set path = ( /usr/local/pgsql/bin $path )
```

ПРИМЕЧАНИЕ. При установке операционной системы FreeBSD могут устанавливаться 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.

Аналогично можно настроить и путь к электронным руководствам по утилитам СУБД. В те же файлы нужно добавить строки:

```
MANPATH=:/usr/local/pgsql/share/man
export MANPATH
```

Двоеточие означает, что содержимое переменной MANPATH будет добавлено в конец списка путей, определяемых в файле **/etc/manpath.config**.

Если вы вносили изменения в файл **/etc/profile**, то необходимо перезагрузить операционную систему, чтобы эти изменения вступили в силу.

В ОС Debian настройка переменной MANPATH не является обязательной, поскольку путь к каталогу **/usr/local/pgsql/share/man**, содержащему электронные руководства, определяется автоматически на основе значения переменной PATH, а в нее мы добавили путь **/usr/local/pgsql/bin**.

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**.

```
adduser postgres
```

В ОС Debian большая часть параметров принимается по умолчанию автоматически. В качестве командного интерпретатора будет назначен **/bin/bash**. В ОС FreeBSD вам будет предложено ввести ваши значения, но также можно основную часть параметров 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 от имени пользователя postgres с помощью следующей команды:

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

На экран выводится целый ряд сообщений. Обратите внимание на информацию о параметрах локализации. Например, в ОС Debian она будет такой:

```
Кластер баз данных будет инициализирован с локалью "ru_RU.UTF-
8".
Кодировка БД по умолчанию, выбранная в соответствии с настрой-
ками: "UTF8".
```

В ОС FreeBSD информация о параметрах локализации будет такой:

```
The database cluster will be initialized with locale
ru_RU.KOI8-R.
```

```
The default database encoding has accordingly been set to
KOI8.
```

Последние сообщения в ОС Debian примерно такие:

```
Готово. Теперь вы можете запустить сервер баз данных:

    /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l
logfile start
```

В ОС FreeBSD они выглядят так:

```
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
```

В ОС Debian можно проверить, запущен ли серверный процесс, с помощью команды

```
ps axw
```

В ОС FreeBSD требуется добавить дефис:

```
ps -axw
```

Если вы все сделали правильно, то на экране найдете примерно такие строки, означающие, что сервер успешно запущен:

```
 2532 pts/5    S      0:00 /usr/local/pgsql/bin/postgres -D
/usr/local/pgsql/data
 2534 ?        Ss     0:00 postgres: checkpointer process
 2535 ?        Ss     0:00 postgres: writer process
 2536 ?        Ss     0:00 postgres: wal writer process
 2537 ?        Ss     0:00 postgres: autovacuum launcher pro-
cess
 2538 ?        Ss     0:00 postgres: stats collector process
```

При первом запуске серверного процесса создается файл-журнал **logfile** в каталоге **/home/postgres**.

8. Если сервер СУБД запущен, можно создать базу данных. В качестве ее имени выберем test. Конечно, вы можете выбрать и какое-то другое имя. От имени пользователя postgres выполните команду

```
/usr/local/pgsql/bin/createdb test
```

Поскольку вы добавили каталог **/usr/local/pgsql/bin** в переменную PATH в глобальном файле **/etc/profile** (или в конфигурационном файле в домашнем каталоге конкретного пользователя), то эту команду можно сократить до

```
createdb test
```

9. Пробуем подключиться к только что созданной базе данных test (опять как пользователь postgres):

```
/usr/local/pgsql/bin/psql -d test
```

А можно и короче:

```
psql -d test
```

или даже

```
psql test
```

На экране вы увидите (у вас номер версии может быть другой):

```
psql (10.1)
Введите "help", чтобы получить справку.

test=#
```

Для выхода введите команду

```
\q
```

10. Для останова сервера баз данных необходимо выполнить команду:

```
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data stop
```

На этом установка СУБД PostgreSQL может считаться в основном завершенной. Добавим лишь несколько замечаний.

Конечно, настоящее краткое руководство не может и не должно заменять документацию, входящую в комплект поставки СУБД. Традиционно мы рекомендуем обратиться к электронным руководствам **man**.

В комплект поставки входит также и очень подробная документация в формате HTML. Она находится в каталоге **/usr/local/pgsql/share/doc/html**. Стартовый файл, содержащий оглавление, – **index.html**.

8.3. Запуск и останов сервера

Одним из возможных подходов является запуск сервера только в случае возникновения потребности в его использовании. Для упрощения решения такой задачи можно в каталоге **/home/postgres** создать файл **pg_start**, содержащий команду для запуска сервера БД:

```
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data
 -l logfile start
```

Приведенную команду можно ввести в одну строку или использовать символ «\» для продолжения команды на следующей строке.

Назначьте этому файлу права на исполнение:

```
chmod 755 pg_start
```

Теперь при необходимости запуска сервера баз данных нужно зарегистрироваться в системе под именем пользователя postgres и запустить этот файл на исполнение:

```
./pg_start
```

Для решения задачи останова сервера баз данных необходимо в каталоге **/home/postgres** создать файл **pg_stop**, поместив в него одну строку:

```
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data stop
```

Назначьте этому файлу права на исполнение:

```
chmod 755 pg_stop
```

Перед завершением работы операционной системы нужно запускать этот файл для корректного «выключения» сервера баз данных:

```
./pg_stop
```

Эта операция также выполняется с правами пользователя postgres.

Если же вы хотите, чтобы сервер СУБД PostgreSQL автоматически запускался при загрузке операционной системы и выгружался при ее выключении, то необходимо использовать специальные скрипты, примеры которых содержатся в подкаталоге **contrib/start-scripts** дистрибутивного комплекта исходных текстов. Рекомендации по использовании этих скриптов в различных операционных системах приведены в разделе документации 18.3 «Запуск сервера баз данных».

В качестве упрощенного решения, например, в ОС FreeBSD можно использовать файлы **/etc/rc.local** и **/etc/rc.shutdown.local**. Если таких файлов еще нет, их нужно создать. 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.

8.4. Создание учетной записи пользователя базы данных

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 добавили 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.

Создать учетную запись пользователя можно и с помощью утилиты, запускаемой непосредственно из командной строки операционной системы. Это делается так (вместо new_user подставьте осмысленное имя):

```
/usr/local/pgsql/bin/createuser  new_user
```

Для получения справки по этой утилите следует сделать так (обратите внимание на два дефиса перед словом help):

```
/usr/local/pgsql/bin/createuser  --help
```

8.5. 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_целостности],
  [ограничение_целостности],
  ...
  [ограничение_целостности],
  [первичный_ключ]
  [внешний_ключ]
  ...
  [внешний_ключ]
);
```

В квадратных скобках показаны необязательные элементы команды. После команды нужно обязательно поставить символ «;».

Для получения полной информации о команде языка SQL непосредственно из среды утилиты **psql** можно сделать так:

```
\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 3 в конце главы).

Команда для создания таблицы «Студенты» будет такой:

```
CREATE TABLE students
( record_book numeric( 5 ) NOT NULL,
  name text NOT NULL,
  psp_ser numeric( 4 ),
  psp_num numeric( 6 ),
  PRIMARY KEY ( record_book )
);
```

Для СУБД регистр символов (прописные или строчные буквы) значения не имеет. Однако зачастую ключевые слова языка SQL вводят в верхнем регистре, что повышает наглядность SQL-операторов. Вы можете следовать тому стандарту кодирования, котрый принят в вашей организации.

Эту команду для создания таблицы students (как и все SQL-команды) в утилите **psql** можно ввести двумя способами. Первый 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. На экране это выглядит так:

```
test=# CREATE TABLE students ( record_book numeric( 5 ) NOT
NULL, name text NOT NULL, psp_ser numeric( 4 ), psp_num
numeric( 6 ), PRIMARY KEY ( record_book ) );
```

Второй 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. Обратите внимание, что 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.

```
test=# CREATE TABLE students
test-# ( record_book numeric( 5 ) NOT NULL,
test(#   name text NOT NULL,
test(#   psp_ser numeric( 4 ),
test(#   psp_num numeric( 6 ),
test(#   PRIMARY KEY ( record_book )
test(# );
```

Впоследствии можно с помощью клавиши «стрелка вверх» вызвать на экран (из буфера истории введенных команд) всю команду полностью в компактном виде и при необходимости отредактировать ее либо выполнить еще раз без редактирования. При этом для команды, введенной построчно, сохраняется ее построчная структура, а приглашение выводится только для первой строки:

```
test=# CREATE TABLE students
( record_book numeric( 5 ) NOT NULL,
  name text NOT NULL,
  psp_ser numeric( 4 ),
  psp_num numeric( 6 ),
  PRIMARY KEY ( record_book )
);
```

Для перемещения курсора по «виртуальным» строкам команды при ее редактировании нужно использовать клавиши «стрелка влево» и «стрелка вправо», но не «стрелка вверх» или «стрелка вниз».

Если вы хотите непосредственно из среды **psql** вызвать внешний редактор для редактирования текущего буфера запроса, то нужно воспользоваться командой **\e**.

Если вы решили прервать ввод команды, еще не введя ее полностью, то просто нажмите клавиши Ctrl-C, в результате ввод команды будет прерван, а приглашение к вводу, выводимое утилитой **psql**, примет свой первоначальный вид:

```
test=# CREATE TABLE students
```

```
test-# ( record_book numeric( 5 ) NOT NULL,
test(# ^C
test=#
```

От обсуждения способов ввода команды вернемся к ее сути. В приведенной команде выражение NOT NULL означает, что не допускается ввод таких записей в таблицу students, у которых значение поля record_book или name не указано. Выражение PRIMARY KEY (record_book) означает, что поле record_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:

```
CREATE TABLE
```

Теперь введите несколько записей в созданную таблицу. Упрощенный формат команды ввода записей (вставки строк) таков:

```
INSERT INTO имя_таблицы ( имя_поля, имя_поля, ... )
VALUES ( значение_поля, значение_поля, ... );
```

В нашем случае это будет выглядеть так:

```
INSERT INTO students ( record_book, name, psp_ser, psp_num )
VALUES ( 12300, 'Иванов Иван Иванович', 0402, 543281 );
```

При успешной вставке новой строки будет выведено сообщение:

```
INSERT 0 1
```

В этом сообщении второе число означает количество строк, успешно добавленных в таблицу. Первое число, в данном случае это 0, имеет, можно сказать, историческое значение – его можно просто не учитывать.

Обратите внимание на одинарные кавычки, в которые заключено значение поля name. Для символьных полей кавычки обязательны, а для числовых они не нужны. 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.

```
 record_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 строки)
```

Можно выбрать только один столбец – 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 ( record_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 (record_book, psp_ser, psp_num )
VALUES ( 13700, 0402, 432781 );
```

Вы получите сообщение об ошибке (внимательно изучите его):

```
ОШИБКА:  нулевое значение в столбце "name" нарушает ограниче-
ние NOT NULL
ПОДРОБНОСТИ:  Ошибочная строка содержит (13700, null, 402,
432781).
```

Сообщение может быть выведено и на английском языке, в зависимости от настроек параметров локализации вашей операционной системы:

```
ERROR:  null value in column "name" violates not-null con-
straint
```

Попробуйте ввести запись с таким значением поля «Номер зачетной книжки» (record_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 record_book = 12300;
```

```
 record_book |         name          | psp_ser | psp_num
-------------+-----------------------+---------+---------
       12300 | Иванов Иван Иванович |     402 |  543281
(1 строка)
```

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 = 0405, psp_num = 123456
WHERE record_book = 13200;
```

С помощью команды SELECT убедитесь, что обновление требуемой строки в таблице прошло успешно.

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 record_book = 12300;
```

Вы можете указать и какое-нибудь более сложное условие, например:

```
DELETE FROM students
WHERE record_book >= 12300 AND record_book <= 12500;
```

или

```
DELETE FROM students
WHERE name <> 'Иванов Иван Иванович' AND
      record_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
( record_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 (record_book )
    REFERENCES students (record_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 record_book. Это означает, что в таблицу «Успеваемость» (progress) нельзя ввести запись, значение поля record_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 record_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 ( 12700, 'Физика', '2017/2018', 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 выполнить команду добавления новой строки, не задавая значение для поля «Оценка» (mark):

```
INSERT INTO progress
VALUES ( 13200, 'Математика', '2017/2018', 1 );
```

Не задавать значение для поля «Оценка» (mark) допустимо, т. к. оно идет последним в списке полей и для него предусмотрено значение по

умолчанию (5). Именно оно 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» (record_book) такое, которого нет в таблице «Студенты» (students). Вы получите сообщение об ошибке, которое демонстрирует работу правил ссылочной целостности:

```
INSERT INTO progress
VALUES ( 23200, 'Математика', '2017/2018', 1 );

ОШИБКА:  INSERT или UPDATE в таблице "progress" нарушает
ограничение внешнего ключа "progress_record_book_fkey"
ПОДРОБНОСТИ:  Ключ (record_book)=(23200) отсутствует в таблице
"students".
```

Аналогичное сообщение на английском языке выглядит так:

```
ERROR:  insert or update on table "progress" violates foreign
key constraint "progress_record _book_fkey"
DETAIL:  Key (record_book)=(23200) is not present in table
"students".
```

Теперь решим более сложную задачу. Предположим, что нам нужно получить экзаменационную ведомость по физике (или другому предмету) за первый семестр 2017/2018 учебного года. Для этого мы выполняем такую команду:

```
SELECT * FROM students, progress
WHERE progress.acad_year = '2017/2018' AND
      progress.term = 1 AND
      progress.subject = 'Физика' AND
      students.record_book = progress.record_book;
```

Обратите внимание на наличие имен двух таблиц в предложении FROM, а также на строку students.record_book = progress.record_book в предложении WHERE. Данная строка (условие) позволяет выполнить так называемое **соединение** таблиц. При этом формируются объединенные за-

писи из тех записей двух таблиц, у которых значения поля record_book одинаковые.

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. Кроме того, отсортируем нашу ведомость по фамилии, имени и отчеству студентов.

```
SELECT progress.record_book, students.name, progress.mark
FROM students, progress
WHERE progress.acad_year = '2017/2018' AND
      progress.term = 1 AND
      progress.subject = 'Физика' AND
      students.record_book = progress.record_book
ORDER BY students.name;
```

А теперь мы можем выбрать все оценки для одного студента за весь период обучения:

```
SELECT *
FROM progress
WHERE record_book = ( SELECT record_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 record_book для того студента, значение поля «Ф. И. О.» (name) которого равно 'Иванов Иван Иванович'.

Для проверки работы правил ссылочной целостности выполните обновление поля record_book в одной из записей таблицы students:

```
UPDATE students
SET record_book = 12900
WHERE record_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 record_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**.

8.6. 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-10.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-10.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
```

В среде ОС Debian сообщение будет таким:

```
linux-gate.so.1 (0xb77d3000)
        libpq.so.5 => /usr/lib/i386-linux-gnu/libpq.so.5
(0xb7778000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6
(0xb75c0000)
...
```

В среде ОС FreeBSD – таким:

```
testlibpq:
        libpq.so.5 => not found (0)
        libc.so.7 => /lib/libc.so.7 (0x2806e000)
```

Мы устанавливали СУБД PostgreSQL в каталог **/usr/local/pgsql**, библиотеки, поставляемые в комплекте с СУБД, находятся в каталоге **/usr/local/pgsql/lib**. Однако программа **testlibpq** в среде ОС Debian оказалась связанной с библиотекой **libpq**, которая находится в системном каталоге – **/usr/lib/i386-linux-gnu**. В этот каталог библиотека был помещена в процессе установки операционной системы Debian. В среде ОС FreeBSD эта библиотека вообще не будет найдена при запуске программы **testlibpq** (вы можете это проверить).

Предположим, что нам требуется связать программу **testlibpq** с той версией библиотеки **libpq**, которая находится в каталоге **/usr/local/pgsql/lib**. Это может обосновываться, например, тем, что в этом каталоге находится библиотека, скомпилированная с поддержкой отладчика (параметр **-g**). 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
```

```
...
libpq.so.5 => /usr/local/pgsql/lib/libpq.so.5 (0xb7736000)
...
```

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 -pthread
```

Обратите внимание на параметр -pthread. Он нужен для подключения библиотеки, предназначенной для работы с потоками.

В среде ОС FreeBSD команду нужно дополнить:

```
gcc -o testlibpq_st testlibpq.c -I/usr/local/pgsql/include
-L/usr/local/pgsql/lib -lpq -lcrypt -static -pthread
-L/usr/local/lib -lintl
```

Если с помощью программы **ldd** проверить зависимость полученного исполняемого модуля от динамических библиотек, то он сообщит следующее:

```
ldd testlibpq_st

не является динамическим исполняемым файлом
```

или

```
ldd: testlibpq_st: not a dynamic ELF executable
```

Как можно увидеть, 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.

Файл **test.c**

```
/*
 * Программа: test.c
 * Проверка возможности доступа к базе данных с помощью
 * библиотеки libpq
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.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 в главе 32 «libpq – библиотека для языка C».

8.7. Язык Perl и СУБД PostgreSQL

Организовать взаимодействие с базой данных под управлением СУБД PostgreSQL можно и из программы на языке Perl. Сначала нужно установить модули (пакеты) DBI и DBD::Pg. Процедура их установки такова.

1. Войдите в систему под именем root.
2. Выполните следующие команды:

```
perl -MCPAN -e 'install DBI'
perl -MCPAN -e 'install DBD::Pg'
```

Для получения подробных инструкций по использованию этих пакета обратитесь к электронной документации:

```
man DBI
man DBD::Pg
```

Рассмотрим программу, которая позволяет добавить новые строки в таблицу Students, а затем выбирает все строки из нее. Эта программа аналогична программе **test.c**, рассмотренной в предыдущем параграфе.

Файл **test_pq.pl**

```
#!/usr/bin/perl -w
# ------------------------------------------------------------
# Программа test_pq.pl.
# Организация доступа к базе данных с помощью
# пакетов DBI и DBD::Pg.
# ------------------------------------------------------------

# Эта директива означает строгую проверку синтаксиса исходного
# текста программы.
use strict;

# Эта директива означает, что исходный текст скрипта
# представлен в Unicode в кодировке UTF-8 (речь идет
# о символьных строках). Условие if означает, что
# подключение модуля utf8 будет выполнено, если
# операционная система -- Lnux.
use if ( $^O eq 'linux' ), "utf8";

# Этот модуль нужен для использования функции decode().
use if ( $^O eq 'linux' ), "Encode";

# Модуль для работы с базами данных.
use DBI;
```

```
# Так можно объявить константу.
use constant DBNAME => 'test';   # имя базы данных

# Назначим вывод в кодировке UTF-8 для устройства
# STDERR (стандартное устройство вывода сообщений
# об ошибках), т. к. функция die() выводит сообщение
# на STDERR (см. perldoc -f die).
# Попробуйте закомментировать эту команду. Будет
# выведено предупреждение "Wide character in die at ...".
# ПРИМЕЧАНИЕ. Можно сделать и так (хотя эти два метода
#             не тождественны):  binmode STDOUT, ':utf8';
binmode STDERR, ':encoding(UTF-8)';

# При выводе данных на стандартный вывод также
# назначаем кодировку UTF-8. Если эту строку
# закомментировать, то при вызове функций print будет
# выведено предупреждение "Wide character in print ...".
binmode STDOUT, ':encoding(UTF-8)';

# Попробуйте закомментировать эту строку и выполнить
# ввод записей в базу данных. Что получается в результате?
binmode STDIN, ':encoding(UTF-8)';

my $conn;     # дескриптор соединения с БД
my $rc;       # результат выполнения SQL-запроса
my $row;      # ссылка на хеш-массив для хранения
              # одной записи из выборки
my $command;  # строка SQL-запроса для работв с БД
my $sth;      # результат подготовки запроса к БД
my $num_recs; # число записей, введенных в таблицу Students
my $err_flag; # признак наличия ошибки в рамках транзакции

# Сначала выполним подключение к базе данных.
# Параметры означают следующее: RaiseError => 0 --
# будем сами проверять результат выполнения каждой
# операции с базой данных; AutoCommit => 1 --
# каждая операция автоматически фиксируется в базе данных,
# если только не была начата транзакция явным образом;
# PrintError => 0 -- не выводить дополнительную
# информацию об ошибке. Если значение RaiseError будет
# равно 1, тогда не нужно проверять результат выполнения
# каждой операции: в случае сбоя будет сформировано
# исключение.
$conn = DBI->connect( "dbi:Pg:dbname=" . DBNAME,
                      'postgres', '',
                      { RaiseError => 0, AutoCommit => 1,
                        PrintError => 0 } );

# Проверим успешность подключения.
die "Не могу подключиться к базе данных " . DBNAME . ": " .
    ( $^O eq 'linux' ) ?
```

```
    decode( 'UTF-8', $DBI::errstr ) :
    $DBI::errstr
    unless defined $conn;

# Сначала добавим несколько записей в таблицу Students.
# В качестве иллюстрации работы с транзакциями выполним
# ввод записей в рамках единой транзакции.
# ПРИМЕЧАНИЕ. Можно выполнение операции и проверку
#             ее успешности объединить с помощью
#             логической операции or (пишется строчными
#             буквами!).
$rc = $conn->begin_work or
die "Не удалось начать транзакцию: " . $DBI::errstr;

# Сформируем SQL-запрос в виде символьной строки.
# Знаки "?" (placeholders) соответствуют фактическим
# значениям, которые будут переданы впоследствии
# методу execute.
$command = "INSERT INTO students " .
           "( record_book, name, psp_ser, psp_num ) " .
           "VALUES ( ?, ?, ?, ? )";

# Подготовим (prepare) эту операцию ввода записей
# в таблицу. Это позволит затем выполнять ее повторно,
# подставляя разные значения полей. При этом затраты
# ресурсов сервера уменьшаются.
$sth = $conn->prepare( $command ) or
die "Не удалось подготовить запрос: " . $DBI::errstr;

print "Вводите данные о студенте в одну строку,\n";
print "разделяя значения полей пробелами:\n";
print "Номер зачетной книжки Ф. И. О. " .
      "Серия паспорта Номер паспорта\n";
print "Например:\n";
print "12345 Котов Петр Андреевич 0205 123098\n";
print "Для отказа от дальнейшего ввода просто " .
      "нажмите клавишу Enter\n";

$num_recs = 0;
$err_flag = 0;

# Получаем данные со стандартного ввода.
while ( <STDIN> )
{
  chomp;
  # Если пустая строка -- завершаем ввод записей.
  last if /^$/;

  # Разделим единую строку на отдельные поля.
  my @values = split( /\s+/ );
```

```
  # Выполним запрос к базе данных.
  # Если произошла ошибка при выполнении запроса
  # к базе данных, то не будем завершать работу
  # с помощью вызова функции die, а в качестве
  # примера обработаем эту ошибку сами.
  $rc = $sth->execute( $values[ 0 ],
                       "$values[ 1 ] $values[ 2 ] " .
                       "$values[ 3 ]",
                       $values[ 4 ], $values[ 5 ] );

  if ( defined $rc )  # Запись успешно добавлена.
  {
    $num_recs++;
    print "Запись добавлена в таблицу Students\n";
    print "Для отказа от ввода просто нажмите " .
          "клавишу Enter\n";
  }
  else
  {
    # Можно получить сведения об ошибке не только
    # с помощью $DBI->errstr, но и с помощью
    # обращения к $sth. Переменная state содержит код
    # ошибки. Такие коды предусмотрены стандартом SQL.
    print "Не удалось добавить запись в таблицу Students\n";
    print "Ошибка (код PostgreSQL): " . $sth->state . "\n";
    print $sth->errstr . "\n\n";
    $err_flag = 1;
    last;
  }

}

# Если ввели хотя бы одну запись и не произошло ошибки
# в процессе выполнения транзакции.
if ( $num_recs > 0 && ! $err_flag )
{
  print "Сохранить записи в базе данных (y/n)?\n";

  my $save = <STDIN>;
  chomp $save;

  if ( $save eq 'y' )
  {
    # Зафиксируем транзакцию в базе данных.
    $rc = $conn->commit or
    die "Не удалось зафиксировать транзакцию: " .
        $DBI::errstr;
  }
  else
  {
    print "Отменим транзакцию\n\n";
```

```
    $rc = $conn->rollback or
    die "Не удалось отменить транзакцию: " . $DBI::errstr;
  }
}
# Не ввели ни одной записи или была ошибка в транзакции.
else
{
  print "Отменим транзакцию\n\n";
  $rc = $conn->rollback or
  die "Не удалось отменить транзакцию: " . $DBI::errstr;
}

# Выведем все строки из таблицы Students, упорядочим
# их по значениям поля name. Сформируем SQL-запрос
# в виде символьной строки.
$command = "SELECT * FROM students ORDER BY name";

# Подготовим запрос.
$sth = $conn->prepare( $command ) or
die "Не удалось подготовить запрос: " . $DBI::errstr;

# Выполним запрос к базе данных.
$rc = $sth->execute or
die "Не удалось выполнить запрос: " . $DBI::errstr;

# Если в выборке нет ни одной строки, завершим работу.
if ( $sth->rows == 0 )
{
  print "В таблице Students нет записей\n";

  # Отключаемся от сервера баз данных.
  $conn->disconnect or
  die "Сбой при отключении от сервера: " . $DBI::errstr;

  exit ( 0 );
}

# Выведем заголовки столбцов.
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_hashref считывает в хеш-массив
# очередную строку из выборки и возвращает ссылку
# на этот хеш-масив.
while ( $row = $sth->fetchrow_hashref )
{
  # Значения NULL, которые могут присутствовать
```

```
  # в базе данных, представляются в Perl в виде значений
  # undef. Для корректного вывода данных на экран
  # заменим значения undef пустыми строками.
  foreach ( keys %{ $row } )
  {
    $row->{ $_ } = '' unless defined $row->{ $_ };
  }

  # Обращаемся к хеш-массиву, используя в качестве
  # ключей имена полей (столбцов) таблицы Students.
  printf "%-12s", $row->{ 'record_book' };
  printf "%-30s", $row->{ 'name' };
  printf "%-12s", $row->{ 'psp_ser' };
  printf "%-12s", $row->{ 'psp_num' };
  print "\n";
}

# Отключаемся от сервера баз данных.
$conn->disconnect or
die "Сбой при отключении от сервера: " . $DBI::errstr;

exit ( 0 );
```

Выполнить эту программу можно таким образом:

```
perl ./test_pq.pl
```

Возможен и такой подход:

```
chmod 755 test_pq.pl
./test_pq.pl
```

Контрольные вопросы и задания

1. Иногда сервер баз данных PostgreSQL не запускается, а на экран выводится сообщение об ошибке. Если вы работаете в UNIX-подобной операционной системе, тогда с помощью команды

```
man postgres
```

самостоятельно ознакомьтесь с электронным руководством по программе postgres. В разделе «Diagnostics (диагностика)» изучите с причины возможного отказа сервера от запуска. Самостоятельно ознакомьтесь также с разделом документации 18.3.1 «Сбои при запуске сервера».

2. Организуйте автоматический запуск сервера баз данных при загрузке операционной системы. Воспользуйтесь разделом документации 18.3 «Запуск сервера баз данных».

3. Какая команда предназначена для создания таблиц в базе данных?

4. Что такое внешний ключ? Для чего он предназначен?

5. Что означает термин «каскадное удаление записей»?

6. 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.

7. Модифицируйте базу данных test. Создайте еще одну таблицу «Учебные дисциплины». Эта таблица должна содержать два столбца (поля): «Код учебной дисциплины» (числовой тип данных) и «Наименование учебной дисциплины» (символьный тип данных). В таблице «Успеваемость» замените столбец «Предмет» столбцом «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.

8. Модифицируйте программу **test.c**, например, реализуйте операции ввода новых записей в таблицу «Студенты» (students) и удаления существующих записей из этой таблицы.

9. Изучите остальные программы из набора тестовых программ, входящих в комплект поставки СУБД PostgreSQL.

10. Модифицируйте программу **test_pq.pl**: реализуйте операции обновления записей в таблице «Студенты» (students) и удаления записей из нее.

11. Модифицируйте программу **test_pq.pl**: для таблицы «Успеваемость» (progress) реализуйте операции ввода, обновления и удаления записей. При этом сделайте так, чтобы все сведения об экзаменационных оценках программа «умела» читать из списка, хранящегося в текстовом файле.

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 в настоящей главе будут рассмотрены некоторые элементы web-технологий.

9.1. Установка web-сервера Apache и интерпретатора языка программирования PHP

Web-сервер – это программно-аппаратный комплекс, который управляет процессом предоставления информации пользователям, обращающимся к услугам web-сервера с помощью браузеров и других программных средств. 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://httpd.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 символом «x».

Поскольку процесс установки программных продуктов в операционных системах Debian и FreeBSD вам уже знаком, ограничимся лишь краткой инструкцией. Напомним, что для установки программного обеспечения нужно войти в систему под именем пользователя root.

Для установки web-сервера Apache может потребоваться наличие ряда библиотек в вашей системе. Если в процессе работы утилиты **configure** или утилиты **make** выяснится, что каких-то библиотек или других пакетов в вашей системе не хватает, нужно прервать установку web-сервера и установить недостающие программы.

Расскажем только о двух библиотеках, которые, возможно, потребуется установить. Первая – это библиотека **libexpat1-dev**. Ее в ОС Debian можно установить с помощью команды

```
apt-get install libexpat1-dev
```

В среде ОС FreeBSD эта библиотека установлена по умолчанию.

Вторая библиотека – это **PCRE**, т. е. Perl Compatible Regular Expressions. В среде ОС FreeBSD эта библиотека также установлена по умолчанию. А в среде Debian установим ее из исходных текстов. Получить их можно с сайта http://pcre.org. Процедура установки такова:

```
tar xzvf pcre-8.x.tar.gz
cd pcre-8.x
```

Создадим библиотеки с поддержкой Unicode (8, 16 и 32), поэтому в команду конфигурирования добавим соответствующе параметры (команда вводится одной строкой):

```
./configure --enable-utf --enable-pcre16 --enable-pcre32 >
conf_log 2>&1 &
```

Теперь скомпилируем программы, протестируем и установим:

```
make > make_log 2>&1 &
make check > make_check_log 2>& &
make install > make_install_log 2>&1 &
```

Перейдем непосредственно к установке web-сервера.

1. Извлеките файлы из архива и перейдите в созданный подкаталог **httpd-2.4.x**:

```
tar xzvf httpd-2.4.x.tar.gz
cd httpd-2.4.x
```

2. С сайта http://apr.apache.org получите исходные тексты библиотек **APR** и **APR-util** (Apache Portable Runtime) и скопируйте их в подкаталог **srclib** того каталога, в котором уже находятся исходные тексты web-сервера. Перейдите в этот подкаталог и извлеките файлы из архивов. Затем переименуйте эти два подкаталога, чтобы их имена не содержали номеров версий.

```
cp apr-1.6.x.tar.gz httpd-2.4.x/srclib
cp apr-util-1.6.x.tar.gz httpd-2.4.x/srclib

cd httpd-2.4.x
cd srclib

tar xzvf apr-1.6.x.tar.gz
tar xzvf apr-util-1.6.x.tar.gz

mv apr-1.6.x apr
mv apr-util-1.6.x apr-util
```

3. Вернитесь в каталог на один уровень выше, т. е. в базовый каталог исходных текстов web-сервера:

```
cd ..
```

Запустите утилиту конфигурирования. В среде ОС Debian команда будет такой:

```
./configure --enable-so --enable-include --with-included-apr
LIBS="-lexpat" > conf_log.txt 2>&1 &
```

А в среде ОС FreeBSD нужно дополнить параметр LIBS:

```
./configure --enable-so --enable-include --with-included-apr
LIBS="-L/usr/local/lib -lexpat -liconv" > conf_log.txt 2>&1 &
```

Параметр --**enable-so** нужен для возможной дальнейшей установки интерпретатора языка программирования PHP. Параметр --**enable-include** подключает технологию Server Side Includes (SSI). Параметр --**with-included-apr** предназначен для подключения библиотек **APR** и **APR-util**, о которых мы говорили выше. Переменная LIBS задает имена библиотек **libexpat** и **libiconv**, а также имя каталога, в котором они находятся в среде ОС FreeBSD. Если, например, библиотека **libexpat** не будет найдена, то утилита **make** сообщит, что не может найти ряд функций, имена которых начинаются с префикса XML_, например, **XML_SetEntityDeclHandler**.

4. Скомпилируйте программы:

```
make > make_log.txt 2>&1 &
```

5. Установите программы (по умолчанию они будут установлены в каталог **/usr/local/apache2**):

```
make install > make_install_log.txt 2>&1 &
```

6. Запустите web-сервер:

```
/usr/local/apache2/bin/apachectl start
```

Можно это сделать и таким образом:

```
cd /usr/local/apache2/bin
./apachectl start
```

ПРИМЕЧАНИЕ. Если будут выведены предупреждающие сообщения, то возможно, что web-сервер все же успешно запустится.

Чтобы web-сервер запускался при загрузке компьютера, можно включить эту команду в один из файлов начальной загрузки, 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:

```
1671 ?   Ss   0:00 /usr/local/apache2/bin/httpd -k start
1672 ?   S    0:00 /usr/local/apache2/bin/httpd -k start
1673 ?   Sl   0:00 /usr/local/apache2/bin/httpd -k start
1674 ?   Sl   0:00 /usr/local/apache2/bin/httpd -k start
1676 ?   Sl   0:00 /usr/local/apache2/bin/httpd -k start
```

7. Протестируйте работу web-сервера, введя в адресной строке браузера

```
http://localhost
```

На экран должно быть выведено лишь короткое сообщение:

```
It works!
```

8. Выполните ряд настроек в конфигурационном файле, необходимых для более эффективного использования web-сервера. Не забудьте, что выполнять все эти действия нужно с правами пользователя root. Сначала создайте резервную копию конфигурационного файла:

```
cd /usr/local/apache2/conf
cp httpd.conf httpd.conf.orig
```

Для обеспечения возможности запуска CGI-программ раскомментируйте следующую строку (т. е. удалите символ «#» в начале строки):

```
LoadModule cgid_module modules/mod_cgid.so
```

При установке программного продукта устанавливается подробная документация в каталог **/usr/local/apache2/manual**. Чтобы сделать ее просмотр доступным через браузер, нужно раскомментировать следующие строки:

```
Include conf/extra/httpd-manual.conf
LoadModule negotiation_module modules/mod_negotiation.so
```

Для того чтобы впоследствии можно было использовать язык PHP, нужно раскомментировать еще одну строку:

```
LoadModule rewrite_module modules/mod_rewrite.so
```

После внесения изменений в конфигурационный файл необходимо «сказать» web-серверу, чтобы он заново перечитал конфигурационный 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`
```

Можно использовать для этой цели и команду **apachectl**, с помощью которой вы запускали web-сервер:

```
/usr/local/apache2/bin/apachectl restart
```

Теперь проверьте, доступна ли документация:

```
http://localhost/manual
```

9. Чтобы остановить работу web-сервера, выполните команду (как пользователь root)

```
/usr/local/apache2/bin/apachectl stop
```

или

```
cd /usr/local/apache2/bin
./apachectl stop
```

Если вы настраивали автоматический запуск web-сервера при загрузке операционной системы, то настройте и его автоматический останов при ее выключении. Эти настройки аналогичны тем, которые вы выполняли для сервера баз данных PostgreSQL.

10. Для получения дополнительной информации используйте электронные руководства:

```
man -M /usr/local/apache2/man httpd
```

```
man -M /usr/local/apache2/man apachectl
```

Теперь установим интерпретатор языка программирования PHP. Установку выполним из исходных текстов, получить которые можно с сайта http://php.net.

Извлечем исходные тексты из архива:

```
tar xzvf php-7.x.x.tar.gz
cd php-7.x.x
```

Сконфигурируем исходные тексты, при этом укажем, что должна быть включена поддержка СУБД PostgreSQL (вся команда вводится одной строкой):

```
./configure --prefix=/usr/local/php
--with-apxs2=/usr/local/apache2/bin/apxs
--with-gettext=/usr/lib --enable-intl
--with-pgsql=/usr/local/pgsql
--with-pdo-pgsql=/usr/local/pgsql > conf_log  2>&1 &
```

В среде ОС FreeBSD значение параметра --with-gettext нужно заменить на такое:

```
--with-gettext=/usr/local/lib
```

Скомпилируем, протестируем и установим интерпретатор:

```
make > make_log 2>&1 &
make test > make_test_log 2>&1 &
make install > make_install_log 2>&1 &
```

Скопируем конфигурационный файл в системный каталог:

```
cp php.ini-development /usr/local/lib/php.ini
```

Эта строка добавлена в файл /usr/local/apache2/conf/httpd.conf при выполнении команды make install:

```
LoadModule php7_module        modules/libphp7.so
```

В конец файла /usr/local/apache2/conf/httpd.conf нужно добавить следующие строки:

```
<FilesMatch "\.ph(p[2-7]?|tml)$">
    SetHandler application/x-httpd-php
</FilesMatch>
```

```
<FilesMatch "\.phps$">
    SetHandler application/x-httpd-php-source
</FilesMatch>

RewriteEngine On
RewriteRule (.*\.php)s$ $1 [H=application/x-httpd-php-source]

PHPIniDir    /usr/local/lib
```

Без последней директивы конфигурационный файл php.ini не виден самому интерпретатору PHP.

Теперь нужно запустить (или перезапустить) web-сервер Apache:

```
/usr/local/apache2/bin/apachectl restart
```

Для проверки работоспособности PHP создадим файл phpinfo.php следующего содержания:

```
<?php phpinfo(); ?>
```

Скопируем этот файл в каталог /usr/local/apache2/htdocs и в адресной строке браузера введем

```
http://localhost/phpinfo.php
```

Если в окне браузера будет выведена подробная информация о вашей инсталляции PHP, то, значит, установка завершилась успешно.

9.2. Основы web-разработки

В настоящее время в web-разработке используются различные технологии и языки программирования, в том числе:

– язык гипертекстовой разметки HTML – для создания HTML-документов и описания их содержимого;

– каскадные таблицы стилей CSS (Cascading Style Sheets) – для описания того, как должны выглядеть HTML-элементы. Использование CSS позволяет убрать стилевое оформление из HTML-документа и вынести это оформление в отдельный файл, а также применить единый стиль к группе документов;

– язык программирования JavaScript – для создания интерактивных, динамически изменяемых элементов HTML-страниц;

– язык программирования PHP – для создания динамических и интерактивных HTML-страниц, а также для организации их взаимодействия с базой данных.

Мы выбрали для демонстрационного примера язык PHP 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.

Следует отметить, что язык PHP используется, как говорят, на стороне сервера, а JavaScript – на стороне клиента, т. е. в браузере. Также существует и серверная реализация языка JavaScript – Node.js, но мы ограничимся только работой в браузере.

Начнем рассмотрение перечисленных технологий с языка HTML. В качестве примера HTML-документа представим создание простой формы для получения данных от пользователя. Сначала приведем весь исходный текст документа, а затем детально разберем назначение каждой его строки.

Файл **students.html**

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Простая программа на языке PHP</title>
  <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8" />
  <link rel="stylesheet" href="university.css"
        type="text/css" />
</head>

<body>
<a href="university.php" target="_self">
   Главная страница</a>
<h1>Система &quot;Учет успеваемости студентов&quot;</h1>
<hr />
<script src="students.js"></script>
<form id="student_form"
      action="/students.php?operation=insert"
      method="post"
      onsubmit="return check_student_form()">
  <fieldset id="personal">
    <legend>Персональные данные</legend>
    <label for="r_book">Номер зачетной книжки</label>
    <input id="r_book" type="text" name="record_book"
     size="6" maxlength="5" value=""
     title="Число цифр в номере равно 5" />

    <input type="hidden" name="current_record_book"
     value="12345" />
```

```
    <label for="f_name">Фамилия, имя, отчество</label>
    <input id="f_name" type="text" name="name"
     size="40" maxlength="50" value="" />

    <label for="p_ser">Серия паспорта</label>
    <input id="p_ser" type="text" name="psp_ser"
     size="5" maxlength="4" value=""
     title="Число цифр в серии паспорта равно 4" />

    <label for="p_num">Номер паспорта</label>
    <input id="p_num" type="text" name="psp_num"
     size="7" maxlength="6" value=""
     title="Число цифр в номере паспорта равно 6" />

    <input class="submit" type="submit"
     name="add_student" value="Записать" />

    <p id="error"></p>
  </fieldset>
</form>
<a href="/students.php?operation=list" target="_self">
   Показать список студентов</a>
<hr />
</body>
</html>
```

HTML-документ состоит из так называемых **тегов** (tags). Каждый тег имеет определенное назначение. В первой строке размещается объявление <!DOCTYPE html>, которое не является тегом, а служит для информирования браузера о том, какая версия HTML используется в HTML-документе.

Тег <html> говорит браузеру, что этот документ является HTML-документом:

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

Данный тег служит корневым элементом документа и выступает в роли контейнера для всех остальных элементов (тегов) документа. Большинство тегов имеют так называемый закрывающий (замыкающий) тег. Для тега <html> это будет тег </html>, размещаемый в самом конце документа. Между открывающим и закрывающим тегами располагается содержимое этого элемента.

Тег <head> означает заголовок документа. В заголовке располагаются другие теги. Тег <title> задает заголовок документа, который выводится на панели инструментов браузера и используется поисковыми машинами в сети Интернет.

Тег <meta> содержит ряд **атрибутов**, которые служат для уточнения содержания тега:

```
<meta http-equiv="Content-Type"
      content="text/html; charset=utf-8" />
```

Этот тег задает служебную информацию для документа. Атрибут content говорит о том, что документ является текстовым документом в формате HTML, представленным в кодировке символов UTF-8. В этом теге могут указываться автор документа, ключевые слова и другие сведения, которые не отображаются в окне браузера, но используются как браузерами, так и поисковыми машинами. Обратите внимание, что для этого тега нет парного закрывающего тега </meta>, однако перед правой угловой скобкой располагается символ «/», выполняющий роль закрывающего тега. Также важно заключать значения атрибутов в двойные кавычки.

Тег <link> позволяет с помощью соответствующих атрибутов (rel, href, type) связать HTML-документ с файлом, содержащим таблицы стилей, используемых для отображения документа:

```
<link rel="stylesheet" href="university.css"
      type="text/css" />
```

Параметры стиля в нашем примере содержатся в файле **university.css**. Здесь также в качестве закрывающего тега служит символ «/», стоящий перед правой угловой скобкой.

Между парой тегов <body> . . . </body> содержится собственно сам документ, все его элементы.

Тег <a> формирует HTML-ссылку для перехода на другие страницы. В данном случае эта ссылка приведет пользователя на главную страницу сайта:

```
<a> href="university.php" target="_self">Главная страница</a>
```

Атрибут target предписывает браузеру открыть страницу **university.php** в этом же окне.

Тег <h1> задает заголовок. Цифра (от 1 до 6) обозначает уровень заголовка: 1 – заголовок самого высокого уровня, 6 – заголовок самого низкого уровня. Обратите внимание, что в тексте заголовка присутствуют комбинации символов «"»:

```
<h1>Система &amp;quot;Учет успеваемости студентов&amp;quot;</h1>
```

Таким способом обозначаются так называемые HTML-сущности (entities). Ими заменяются символы, зарезервированные, т. е. являющиеся

служебными, в HTML. С помощью этих сущностей можно обозначить и символы, отсутствующие на клавиатуре. Вышеупомянутая комбинация символов «"» означают двойные кавычки.

Тег <hr /> означает изменение темы, т. е. является смысловым тегом, но может использоваться и в качестве горизонтальной линии, отделяющей части документа друг от друга. В нашем примере это именно так.

Тег <script> предназначен для подключения программы, написанной на языке JavaScript, к HTML-документу. Имя этой программы указывается в атрибуте src:

```
<script src="students.js"></script>
```

Тег <form> позволяет создавать web-формы, предназначенные для получения данных от пользователя и отправки их на web-сервер для обработки. Обработка выполняется с помощью программы, URL (адрес) которой указан в атрибуте action:

```
<form id="student_form"
      action="/students.php?operation=insert"
      method="post" onsubmit="return check_student_form()">
```

Программа, получающая от браузера данные, введенные пользователем в поля формы, может принимать параметры. Их дописывают после символа «?», при этом наименования параметров и их значения разделяются символом «=», а пары параметр/значение разделяются символом «&». Атрибут id присваивает этой форме уникальный идентификатор, что позволяет связать с ним стилевое оформление. Как это делается, мы покажем позднее. Атрибут method назначает метод отправки данных на web-сервер – POST. Существует еще метод GET. Значения полей формы будут переданы методом POST, а те параметры, которые добавлены к имени программы в атрибуте action, будут переданы методом GET. Метод POST считается более безопасным, чем метод GET, поскольку при использовании метода POST данные передаются на сервер в теле HTTP-запроса, а при использовании метода GET – в составе URL скрипта, используемого для их обработки, поэтому данные становятся более доступными для несанкционированного просмотра.

В теге <form> есть еще один атрибут – onsubmit. В нем содержится фрагмент кода на языке JavaScript, который срабатывает, когда пользователь нажимает кнопку типа submit (речь о ней пойдет ниже). В данном случае этот код предписывает выполнить функцию check_student_form для проверки значений полей формы, введенных пользователем. Если эта функция возвратит значение true, тогда работа с формой завершается и данные отправляются на сервер. В случае возвращения значения false дан-

ные на сервер не отправляются и работа с формой должна быть продолжена.

Тег <fieldset> предназначен для объединения нескольких полей в единую группу. Вокруг этих полей будет изображена рамка. Для дальнейших ссылок на этот элемент документа он также имеет атрибут id:

```
<fieldset id="personal">
```

Тег <legend> формирует заголовок для тега <fieldset>:

```
<legend>Персональные данные</legend>
```

Тег <input> предназначен для создания полей для ввода данных пользователем. Эти поля могут быть различных типов: окно для ввода текста, список опций для выбора, радиокнопки и т. д. Тип поля указывается в атрибуте type (это поле – текстовое).

```
<input id="r_book" type="text" name="record_book"
     size="6" maxlength="5" value=""
     title="Число цифр в номере равно 5" />
```

Атрибут id также присутствует: он позволит впоследствии ссылаться на это поле в таблицах стилей и в программах на языке JavaScript. Остальные атрибуты имеют такое назначение:

– name – может использоваться для обращения к элементам по их именам в программах на JavaScript (функция getElementsByName);

– size – задает ширину поля ввода (в символах);

– maxlength – задает максимальное число символов, которые можно ввести в это поле. Если значение этого атрибута будет больше, чем значение атрибута size, тогда в это поле можно будет ввести больше символов, чем можно отобразить в окне. Будет организована прокрутка символов (скроллинг);

– value – задает начальное значение поля (это удобно для организации корректировки информации в базе данных);

– title – задает текст всплывающей подсказки, которая появляется при наведении указателя мыши на поле.

Обратите внимание, что в самом конце тега <input> стоит символ «/», которым закрывается тег. Такой прием вы уже видели применительно к другим тегам.

Тег <label> задает заголовок для поля ввода. Для связи заголовка и поля используется атрибут for заголовка. Его значение должно совпадать со значением id соответствующего поля:

```
<label for="r_book">Номер зачетной книжки</label>
```

Существуют и так называемые скрытые поля. Они используются, например, для задания значений по умолчанию и для передачи значений ключевых полей от одного документа к другому. При корректировке информации в базе данных может (хотя и не всегда) допускаться изменение значений ключевых полей таблицы. В таком случае текущее значение ключевого поля (полей) сохраняется в скрытом поле формы, а при передаче введенных пользователем данных оно также отправляется web-серверу. Если пользователь изменил значение ключевого поля (полей) в HTML-форме, то на основе прежнего значения ключевого поля (полей) таблицы, переданных в скрытых полях формы, будет найдена требуемая запись в таблице базы данных и операция UPDATE будет выполнена успешно.

```
<input type="hidden" name="current_record_book"
 value="12345" />
```

Чтобы отправить введенные данные web-серверу, нужно воспользоваться специальной кнопкой, которая также создается с помощью тега <input>, но имеет тип submit.

```
<input class="submit" type="submit" name="add_student"
       value="Записать" />
```

Значение атрибута value будет написано на кнопке. Атрибут class позволяет назначить одинаковое стилевое оформление группе HTML-элементов, имеющих одинаковые значения этого атрибута. Попутно скажем, что значения различных атрибутов в рамках одного HTML-элемента могут совпадать, если этого требует логика создания конкретной программы. В нашем примере совпадают значения атрибутов class и type.

Для создания абзацев (параграфов) служит тег <p>. В нашем примере этот абзац – пустой. Он предназначен для вывода в него сообщений об ошибках, допущенных пользователем при заполнении полей формы.

```
<p id="error"></p>
```

В конце документа есть еще одна ссылка:

```
<a href="/students.php?operation=list" target="_self">
   Показать список студентов</a>
```

Для того чтобы увидеть рассмотренный пример в работе, введите исходный текст документа в файл и сохраните его под именем **students.html**.

Теперь рассмотрим каскадные таблицы стилей – CSS. Как уже было сказано выше, они позволяют отделить стилевое оформление HTML-документа от его содержания.

Синтаксис CSS прост: каждое правило состоит из селектора и набора объявлений, заключенного в фигурные скобки, а объявление состоит из наименования свойства и его значения, разделенных символом «:». В конце каждого объявления нужно поставить символ «;».

```
селектор
{
  свойство: значение;
  свойство: значение;
  ...
}
```

В качестве селектора могут использоваться:

– наименование HTML-элемента (например, h1 – заголовок первого уровня);

– уникальный идентификатор HTML-элемента, назначенный ему с помощью атрибута id. К идентификатору добавляется символ «#» (например, #error);

– идентификатор класса HTML-элементов, назначенный им с помощью атрибута class. К идентификатору добавляется символ «.» (например, .messages);

– группа селекторов, отделенных друг от друга запятыми (например, #students_table, #exams_table, #disciplines_table).

Далее приведем весь текст файла, содержащего стили для HTML-документа. Комментарии, содержащиеся в файле, должны помочь вам понять назначение использовавшихся свойств CSS. В этом файле содержатся стили и для тех HTML-элементов, которых нет в HTML-документе, приведенном выше. Но в дальнейшем мы рассмотрим программу на языке PHP, в которой эти HTML-элементы будут использоваться.

Файл **university.css**

```
/* тело документа */
/* body -- имя HTML-тега, к которому относятся стили */
body
{
  /* Обратите внимание, что между числовым значением
     свойства и единицей измерения пробел не ставится. */
  padding: 0px;   /* свободное пространство внутри
                     элементов */
  margin: 10px;   /* свободное пространство снаружи
                     элементов 10 пикселов (пикселей),
                     т. е. расстояние между элементами
```

```
                    и границей окна */
  font-size: 16px; /* размер шрифта 16 пикселов
                      (пикселей) */
  /* начертание шрифта */
  font-family: arial,helvetica,sans-serif;
  background-color: #ffffff;  /* цвет фона -- белый */
}

/* ------------------------------------------------------
   заголовки
------------------------------------------------------*/
/* главный заголовок программы */
/* h1, h2, h3 -- имена HTML-тегов, к которым
   относятся стили */
h1
{
  font-weight: bold;     /* полужирный шрифт */
  text-align: center;    /* по центру */
  /* размер шрифта относительно базового размера,
     указанного для body */
  /* Обратите внимание, что между числовым значением
     свойства и единицей измерения пробел не ставится. */
  font-size: 2em;
  color: #0000cc;        /* цвет текста */
}

/* заголовки модулей программы */
h2
{
  font-weight: bold;     /* полужирный шрифт */
  text-align: center;    /* по центру */
  font-size: 1.5em;      /* размер шрифта */
  color: #0000cc;
}

/* подзаголовки */
h3
{
  font-weight: bold;     /* полужирный шрифт */
  text-align: center;    /* по центру */
  font-size: 1.2em;
  color: #0000cc;
}

/* заголовок для группы элементов, объединенных
   с помощью тега <fieldset> */
legend
{
  /* размер шрифта относительно базового размера,
     указанного для body */
  font-size: 1.5em;
```

```
  color: #0000cc;
}

/* заголовки полей */
label
{
  margin-top: 15px;    /* сверху отступ 15 пикселов */
  margin-bottom: 5px;  /* снизу отступ 5 пикселов */

}

/* ------------------------------------------------------
  таблицы для вывода списков записей
------------------------------------------------------ */

/* собственно таблицы */
/* Символ # означает, что указанные идентификаторы
   являются значениями атрибутов id в различных
   HTML-тегах. */
#students_table, #exams_table, #disciplines_table
{
  width: 100%;        /* ширина во весь экран */
  /* border: 1px; */  /* граница */
}

/* заголовки этих таблиц */
/* Запись "#students_table th" означает, что стили
   относятся к HTML-тегам th, содержащимся внутри
   тега, атрибут id которого имеет значение
   students_table. В данном случае речь идет
   о заголовках таблиц. */
#students_table th, #exams_table th,
#disciplines_table th
{
  text-align: center;         /* выравнивание по центру */
  background-color: #b2e0ff;  /* цвет фона */
}

/* ячейки этих таблиц */
#students_table td, #exams_table td,
#disciplines_table td
{
  /* background-color: #e6f5ff; */ /* цвет фона */
  /* расстояние между границей и текстом слева */
  padding-left: 5px;
  padding-right: 5px;         /* ... справа */
}

/* чередование цвета фона в строках таблиц */
/* Используются псевдоклассы :nth-child, позволяющие
   сделать фон в четных (even) строках (tr) таблицы
```

```
   отличающимся от фона нечетных строк таблицы. */
#students_table tr:nth-child(even), #exams_table tr:nth-
child(even),
#disciplines_table tr:nth-child(even)
{
  background-color: #f2f2f2;
}

/* фон для нечетных (odd) строк таблицы */
#students_table tr:nth-child(odd),
#exams_table tr:nth-child(odd),
#disciplines_table tr:nth-child(odd)
{
  background-color: #e6f5ff;
}

/* центрируемые ячейки таблиц */
/* center -- это значение атрибута class, имеющегося
   у тегов td, обозначающих ячейки таблицы. Эти таблицы
   имеют id со значениями students_table, exams_table
   и disciplines_table. */
#students_table td.center, #exams_table td.center,
#disciplines_table td.center
{
  text-align: center;
}

/* ------------------------------------------------------
  различные сообщения
------------------------------------------------------ */
/* информационные сообщения, выводимые при завершении
   того или иного режима работы */
/* messages -- это значение атрибута class */
.messages
{
  color: #008000;
  /* background-color: #90ee90; */
  font-size: 1.5em;
}

/* сообщения об ошибках пользователя */
/* error -- это значение атрибута id */
#error
{
  color: #ff0000;
  font-size: 1.0em;
}

/* ------------------------------------------------------
  различные поля для ввода, кнопки и ссылки
------------------------------------------------------ */
```

```
/* поле формы, которое получило фокус ввода */
/* :focus -- псевдокласс */
input:focus, textarea:focus, select:focus
{
  background-color: #dbffff;
}

/* кнопки "Записать" */
/* submit -- это класс, хотя существует
   и тип кнопок -- submit */
.submit
{
  margin-top: 15px;     /* отступ сверху 15 пикселов */
  background-color: #0033cc;
  color: #ffffff;
}

/* поля для ввода типа text, списки для выбора,
   текстовые окна и заголовки полей */
/* input[type="text"] -- это означает, что стили
   относятся к тегам input, у которых значение атрибута
  type равно text. Через запятую перечислены
  и другие теги. */
input[type="text"], select, textarea, label
{
  /* отображение каждого элемента на отдельной строке */
  display: block;
}

/* перекрасим кнопку при наведении на нее указателя мыши */
/* :hover -- псевдокласс */
input[type=submit]:hover
{
    background-color: #1e90ff;
}

/* все HTML-ссылки */
/* :link и :visited -- псевдоклассы */
a:link, a:visited
{
  color: blue;
  /* не будем подчеркивать ссылки */
  text-decoration: none;
}

/* изменение цвета при наведении указателя мыши на ссылку */
a:hover
{
  color: red;
  /* не будем подчеркивать ссылки */
```

```
  text-decoration: none;
}
```

Введите этот текст в редакторе и сохраните его под именем university.css в том же каталоге, где вы сохранили файл **students.html**. Запустите браузер и введите в его адресной строке полный путь к HTML-файлу, например:

file:///home/stud/my_work/students.html

Посмотрите, как выглядит документ в окне браузера. Для того чтобы лучше понять, на что влияют конкретные CSS-свойства, измените значения некоторых из них и обновите документ в браузере.

Как было сказано выше, для динамического создания страниц можно использовать язык программирования PHP. Поскольку наша учебная программа будет использовать базу данных, нужно эту базу данных создать. Подготовьте файл с именем edu_db.sql, содержащий следующие команды:

Файл **edu_db.sql**

```
DROP DATABASE IF EXISTS edu;
CREATE DATABASE edu;
\connect edu

-- ------------------------------------------------------
-- Таблица "Студенты"
-- ------------------------------------------------------
CREATE TABLE students
( record_book numeric( 5, 0 ) -- номер зачетной книжки
    NOT NULL,
  name text NOT NULL,         -- Ф. И. О. студента
  psp_ser numeric( 4, 0 ),    -- серия паспорта
  psp_num numeric( 6, 0 ),    -- номер паспорта
  PRIMARY KEY ( record_book )
);

-- ------------------------------------------------------
-- Таблица "Учебные дисциплины"
-- ------------------------------------------------------
CREATE TABLE disciplines
( discipline_code smallint    -- код дисциплины
    NOT NULL,
  discipline_name text        -- название дисциплины
    NOT NULL,
  CHECK ( discipline_code > 0 ),
  UNIQUE ( discipline_name ),
  PRIMARY KEY ( discipline_code )
);
```

```
-- ------------------------------------------------------
-- Таблица "Экзаменационные оценки"
-- ------------------------------------------------------
CREATE TABLE progress
( record_book numeric( 5, 0 )  -- номер зачетной книжки
    NOT NULL,
  acad_year text NOT NULL,     -- ученый год
  term numeric( 1, 0 )         -- семестр
    NOT NULL,
  discipline_code smallint     -- код дисциплины
    NOT NULL,
  mark numeric( 1, 0 ),        -- оценка
  CHECK ( mark >= 3 AND mark <= 5 ),
  CHECK ( term = 1 OR term = 2 ),
  PRIMARY KEY ( record_book, acad_year,
                term, discipline_code ),
  FOREIGN KEY ( discipline_code )
    REFERENCES disciplines ( discipline_code )
    ON UPDATE CASCADE,
  FOREIGN KEY ( record_book )
    REFERENCES students ( record_book )
    ON UPDATE CASCADE
    ON DELETE CASCADE
);
```

Обратите внимание, что при создании внешнего ключа, ссылающегося на таблицу disciplines, не предусмотрено каскадного удаления записей из таблицы progress. Поэтому при попытке удаления записи из таблицы disciplines СУБД PostgreSQL сформирует сообщение об ошибке.

Для создания базы данных нужно запустить сервер PostgreSQL и выполнить команду:

```
psql -f edu_db.sql -U postgres
```

Теперь перейдем к языку PHP. Он позволяет создавать динамические HTML-документы путем добавления в обычные HTML-документы небольших фрагментов программного кода. Однако язык PHP позволяет создавать HTML-документы и путем формирования HTML-тегов программным путем. Мы выберем именно этот способ.

Ранее вы уже познакомились с языком Perl, а язык PHP очень похож на него. Поэтому мы ограничимся приведением исходных текстов программ с подробными комментариями, которые рекомендуем вам основательно изучить.

Главный файл программы – university.php. Обратите внимание, что весь текст файла располагается между парой тегов <?php ... ?>.

Файл **university.php**

```
<?php
/*
  Система: "Учет успеваемости студентов"
  Модуль:  главный модуль
*/

// Включаемые файлы содержат код, который будет выполнен
// при выполнении этого скрипта.
// В этом файле содержится информация, на основе которой
// формируется верхняя часть HTML-страницы.
include 'head.php';

echo "<a href=\"students.php\" target=\"_self\">" .
     "Персональные данные студентов</a><br />\n";

echo "<a href=\"disciplines.php\" target=\"_self\">" .
     "Справочник учебных дисциплин</a><br />\n";

// В этом файле содержится информация, на основе которой
// формируется нижняя часть HTML-страницы.
include 'foot.php';
?>
```

Файл **students.php**

```
<?php
/*
  Система: "Учет успеваемости студентов"
  Модуль:  главный модуль для работы с информацией
           о студентах
*/

// Включаемые файлы содержат тексты функций, которые
// могут быть вызваны из этого файла.
include 'students_db.php';
include 'students_view.php';

// Поместив подобную конструкцию в каждую процедуру,
// можно выяснить, какие процедуры работают при
// выполнении того или иного запроса.
// $log = fopen( 'log.txt', 'a' );
// fwrite( $log, "students.php\n" );
// fclose( $log );

// В глобальном массиве $_POST содержатся значения
// полей, переданных скрипту методом POST.

// Если была нажата кнопка с именем 'add_student' ...
// Функция isset проверяет существование указанной
```

```
// переменной или элемента массива.
if ( isset( $_POST[ 'add_student' ] ) )
{
  // Добавим запись в таблицу students.
  $rec_num = insert_student();
  // Выведем сообщение об этом.
  inform_inserted_student( $rec_num );
  exit( 0 );
}
// Если была нажата кнопка с именем 'update_student' ...
elseif ( isset( $_POST[ 'update_student' ] ) )
{
  // Обновим запись в таблице students.
  $rec_num = update_student();
  // Выведем сообщение об этом.
  inform_updated_student( $rec_num );
  exit( 0 );
}

// В глобальном массиве $_GET содержатся значения
// полей, переданных скрипту методом GET.

// Если была выбрана операция добавления новой записи ...
if ( isset( $_GET[ 'operation' ] ) &&
     $_GET[ 'operation' ] == 'insert' )
{
  // Выведем на экран форму с пустыми полями.
  display_student_form( NULL );
}
// Если была выбрана операция обновления записи ...
elseif ( isset( $_GET[ 'operation' ] ) &&
         $_GET[ 'operation' ] == 'update' &&
         isset( $_GET[ 'record_book' ] ) )
{
  // Получим из таблицы students сведения для
  // конкретного студента.
  $student_info =
    get_student_info( $_GET[ 'record_book' ] );
  // Выведем на экран форму с полями, содержащими
  // значения полученной записи из таблицы.
  display_student_form( $student_info );
}
// Если была выбрана операция удаления записи ...
elseif ( isset( $_POST[ 'operation' ] ) &&
         $_POST[ 'operation' ] == 'delete' )
{
  // Удалим из таблицы students сведения о конкретном
  // студенте.
  $rec_num = delete_student( $_POST[ 'record_book' ] );
}
// Выберем все записи из таблицы students.
```

```
else
{
  // Получим выборку из таблицы.
  $students = get_students_list();
  // Выведем ее на экран.
  display_students_list( $students );
}

?>
```

Распределим программный код по различным файлам следующим образом:

– файл **students.php** является главным модулем, он содержит управляющие конструкции для вызова различных функций в зависимости от действий пользователя;

– файл **students_view.php** содержит код для создания экранных форм, вывода данных и сообщений;

– файл **students_db.php** содержит код для выполнения операций с базой данных.

Такая архитектура программы немного напоминает архитектурный шаблон Model – View – Controller (MVC).

Файл **students_view.php**

```
<?php
/*
  Система: "Учет успеваемости студентов"
  Модуль:  отображение персональной информации
           о студентах на экране
*/

/*
  Отображение формы для ввода новой записи
  или корректировки существующей записи в таблице
  students.
*/
function display_student_form( $student_info )
{
  // Параметр: $student_info -- массив данных об одном
  //                            студенте.
  // Этот параметр может и не быть передан, если форма
  // используется для ввода новой записи, а не для
  // корректировки существующей.

  include 'head.php';

  // Таким образом можно подключить файл, содержащий
  // код функций на языке JavaScript.
  echo "<script src=\"students.js\"></script>\n";
```

```
  // Если операция -- обновление записи, тогда эта
  // функция получила параметр $student_info.
  if ( isset( $student_info ) )
  {
    $script_name = $_SERVER[ 'SCRIPT_NAME' ] .
                   "?operation=update";
    $operation = 'update_student';
    $record_book = $student_info[ 'record_book' ];
    $name = $student_info[ 'name' ];
    $psp_ser = $student_info[ 'psp_ser' ];
    $psp_num = $student_info[ 'psp_num' ];
  }
  else  // добавление новой записи
  {
    $script_name = $_SERVER[ 'SCRIPT_NAME' ] .
                   "?operation=insert";
    $operation = 'add_student';
    $record_book = '';
    $name = '';
    $psp_ser = '';
    $psp_num = '';
  }

  // Атрибут action содержит имя скрипта, который
  // получает набор значений, введенных в поля формы.
  // Событие onsubmit вызывает функцию для проверки
  // значений полей формы. Скрытое поле (type="hidden")
  // с именем name="current_record_book" нужно для того,
  // чтобы при изменении значения поля "Номер зачетной
  // книжки" мы могли найти в таблице запись со старым
  // значением этого поля.
  echo <<<END
<form id="student_form" action="$script_name"
      method="post"
      onsubmit="return check_student_form()">
  <fieldset id="personal">
    <legend>Персональные данные</legend>
    <label for="r_book">Номер зачетной книжки</label>
    <input id="r_book" type="text" name="record_book"
     size="6" maxlength="5" value="$record_book"
     title="Число цифр в номере равно 5" />

    <input type="hidden" name="current_record_book"
     value="$record_book" />

    <label for="f_name">Фамилия, имя, отчество</label>
    <input id="f_name" type="text" name="name" size="40"
     maxlength="50" value="$name" />

    <label for="p_ser">Серия паспорта</label>
```

```
    <input id="p_ser" type="text" name="psp_ser" size="5"
     maxlength="4" value="$psp_ser"
     title="Число цифр в серии паспорта равно 4" />

    <label for="p_num">Номер паспорта</label>
    <input id="p_num" type="text" name="psp_num" size="7"
     maxlength="6" value="$psp_num"
     title="Число цифр в номере паспорта равно 6" />

    <input class="submit" type="submit" name="$operation"
     value="Записать" />

    <p id="error"></p>
  </fieldset>
</form>

END;

  echo "<a href=\"" . $_SERVER[ 'SCRIPT_NAME' ] .
       "?operation=list\" target=\"_self\">
       Показать список студентов</a>\n";

  include 'foot.php';
}

/*
  Вывод всех записей из таблицы students.
*/
function display_students_list( $students_list )
{
  // Параметр: $students_list -- двухмерный массив
  //                             данных о студентах.

  include 'head.php';

  // Таким образом можно подключить файл, содержащий
  // код функций на языке JavaScript.
  echo "<script src=\"students.js\"></script>\n";
  echo "<script src=\"university.js\"></script>\n";

  echo "<h2>Персональные данные студентов</h2>\n";

  echo "<a href=\"" . $_SERVER[ 'SCRIPT_NAME' ] .
       "?operation=insert\" target=\"_self\">
       Добавить запись</a><br />\n";

  // Выведем результаты в виде таблицы.
  echo "<table id=\"students_table\">\n";

  // Заголовок таблицы.
  echo "  <tr>\n";
```

```
echo "    <th>Номер зачетной книжки</th>\n" ;
echo "    <th>Фамилия, имя, отчество</th>\n" ;
echo "    <th>Серия паспорта</th>\n" ;
echo "    <th>Номер паспорта</th>\n" ;
echo "    <th> </th>\n" ;  // пустая ячейка
echo "    <th> </th>\n" ;  // пустая ячейка
echo "    <th> </th>\n" ;  // пустая ячейка
echo "  </tr>\n";

// Будем формировать уникальные идентификаторы всех
// строк в таблице: row0, row1 и т. д.
$row = 0;

// Берем из двухмерного массива строку за строкой.
// Каждая строка является хеш-массивом, поэтому
// возможно обращение типа: $stud[ 'name' ].
foreach ( $students_list as $stud )
{
  echo "<tr id=\"row" . $row . "\">\n";

  echo "<td class=\"center\">" . $stud[ 'record_book' ] .
       "</td>\n";
  echo "<td>" . $stud[ 'name' ] . "</td>\n";
  echo "<td class=\"center\">" . $stud[ 'psp_ser' ] .
       "</td>\n";
  echo "<td class=\"center\">" . $stud[ 'psp_num' ] .
       "</td>\n";

  // Конструкция javascript:void( 0 ) предназначена
  // для того, чтобы при выборе этой ссылки только
  // вызывались функции, связанные с событием onclick,
  // а сам скрипт повторно не запускался.
  // Выражение $_SERVER[ 'SCRIPT_NAME' ] означает имя
  // скрипта, который и сформировал эту HTML-страницу.
  // Он вызывается повторно для удаления текущей записи
  // из таблицы. При этом скрипту передаются параметры
  // методом GET. Они идут парами: имя параметра
  // и значение параметра. Пары разделяются знаком "&".
  echo "<td><a href=\"javascript:void( 0 );\" " .
       "onclick=\"confirm( 'Удалить запись?' ) && " .
       "delete_student( '" .
       $_SERVER[ 'SCRIPT_NAME' ] . "', " .
       $stud[ 'record_book' ] .
       ", 'row" . $row . "' )\"" .
       " target=\"_self\">Удалить</a></td>\n";

  // Два параметра: наименование операции (update)
  // и номер зачетной книжки.
  echo "<td><a href=\"" . $_SERVER[ 'SCRIPT_NAME' ] .
       "?operation=update&record_book=" .
       $stud[ 'record_book' ] .
```

```
          "\" target=\"_self\">Изменить</a></td>\n";

    // Два параметра: наименование операции (progress)
    // и номер зачетной книжки.
    echo "<td><a href=\"progress.php" .
         "?operation=progress&record_book=" .
         $stud[ 'record_book' ] .
         "&name=" . $stud[ 'name' ] .
         "\" target=\"_blank\">Экзаменационные оценки" .
         "</a></td>\n";

    echo "</tr>\n";

    $row++;
  }

  echo "</table>\n";

  include 'foot.php';
  return 0;
}

/*
  Сообщение о добавлении новой записи в таблицу students.
*/
function inform_inserted_student( $rec_num )
{
  // Параметр: $rec_num -- число добавленных записей
  //                       (1 или 0).

  // В этом файле содержится информация, на основе
  // которой формируется верхняя часть HTML-страницы.
  include 'head.php';

  if ( $rec_num == 1 )
    echo "<p class=\"messages\">Запись добавлена</p>\n";
  elseif ( $rec_num == 0 )
    echo "<p class=\"messages\">Запись не добавлена</p>\n";

  echo "<a href=\"" . $_SERVER[ 'SCRIPT_NAME' ] .
       "?operation=insert\" target=\"_self\">" .
       "Добавить запись</a><br />\n";

  echo "<a href=\"" . $_SERVER[ 'SCRIPT_NAME' ] .
       "?operation=list\" target=\"_self\">" .
       "Показать список студентов</a><br />\n";

  // В этом файле содержится информация, на основе которой
  // формируется нижняя часть HTML-страницы.
  include 'foot.php';
}
```

```
/*
  Сообщение об обновлении записи в таблице students.
*/
function inform_updated_student( $rec_num )
{
  // Параметр: $rec_num -- число обновленных записей
  //                       (1 или 0).

  include 'head.php';

  if ( $rec_num == 1 )
    echo "<p class=\"messages\">Запись обновлена</p>\n";
  elseif ( $rec_num == 0 )
    echo "<p class=\"messages\">Запись не обновлена</p>\n";

  echo "<a href=\"" . $_SERVER[ 'SCRIPT_NAME' ] .
       "?operation=insert\" target=\"_self\">" .
       "Добавить запись</a><br />\n";

  echo "<a href=\"" . $_SERVER[ 'SCRIPT_NAME' ] .
       "?operation=list\" target=\"_self\">" .
       "Показать список студентов</a><br />\n";

  include 'foot.php';
}

?>
```

Файл **students_db.php**

```
<?php
/*
  Система: "Учет успеваемости студентов"
  Модуль:  работа с базой данных (получение и сохранение
           информации о студентах)
*/

// Строку параметров для подключения к базе данных
// определим в виде константы.
define( "CONNECTION_STRING",
        "host=localhost dbname=edu user=postgres" );

// Поместив подобную конструкцию в каждую процедуру,
// можно выяснить, какие процедуры работают при
// выполнении того или иного запроса.
// $log = fopen( 'log.txt', 'a' );
// fwrite( $log, "delete_student " . $record_book . "\n" );
// fclose( $log );

/*
```

```
  Удаление одной записи из таблицы students.
*/
function delete_student( $record_book )
{
  // Подключаемся к базе данных.
  $dbconn = pg_connect( CONNECTION_STRING )
    or die( 'Не могу подключиться к базе данных: ' .
            pg_last_error() );

  // Сформируем запрос, в котором вместо фактических
  // значений стоят переменные-заменители: $1, $2 и т. д.
  $query = "DELETE FROM students WHERE record_book = $1";

  // Выполняем обращение к базе данных. Фактические
  // значения полей передаем в виде массива.
  $result = pg_query_params( $dbconn, $query,
                             array( $record_book ) );

  // Проверим состояние полученного результата.
  if ( pg_result_status( $result ) != PGSQL_COMMAND_OK )
  {
    // Просто отправим сообщение об ошибке
    // на стандартный вывод. Оно будет отослано
    // web-сервером браузеру и обработано в функции
    // delete_record (см. файл university.js).
    echo pg_last_error( $dbconn );

    // Закроем соединение с сервером баз данных.
    pg_close( $dbconn );

    return;
  }

  // Сформируем объект JSON. Его поле rows будет содержать
  // число фактически удаленных записей (1 или 0). Выведем
  // объект JSON на стандартный вывод. Таким способом
  // web-сервер передаст это значение функции
  // delete_record, написанной на языке JavaScript
  // (см. файл university.js). Эта функция вызывается
  // из функции delete_student при выборе ссылки "Удалить"
  // (см. функцию display_students_list в файле
  // students_view.php).
  echo '{ "rows": "' . pg_affected_rows( $result ) . '" }';

  // Освободим память.
  pg_free_result( $result );

  // Закроем соединение с сервером баз данных.
  pg_close( $dbconn );
}
```

```
/*
  Выборка одной записи из таблицы students.
*/
function get_student_info( $record_book )
{
  // Параметр: $record_book -- номер зачетной книжки

  $dbconn = pg_connect( CONNECTION_STRING ) or
    die( 'Не могу подключиться к базе данных: ' .
         pg_last_error() );

  // Выберем данные для одного студента.
  $query = "SELECT record_book, name, psp_ser, psp_num " .
           "FROM students WHERE record_book = $1";

  $result = pg_query_params( $dbconn, $query,
                             array( $record_book ) );

  // Обратите внимание, что теперь используется статус
  // PGSQL_TUPLES_OK, а не PGSQL_COMMAND_OK.
  if ( pg_result_status( $result ) != PGSQL_TUPLES_OK )
  {
    die ( 'Ошибка: ' . pg_last_error( $dbconn ) );
  }

  // Получим данные из выборки -- это всего одна запись.
  $student_info = pg_fetch_array( $result, NULL,
                                  PGSQL_ASSOC );

  pg_free_result($result);
  pg_close( $dbconn );

  return $student_info;
}

/*
  Выборка всех записей из таблицы students.
*/
function get_students_list()
{
  $students_list = array();

  $dbconn = pg_connect( CONNECTION_STRING ) or
    die( 'Не могу подключиться к базе данных: ' .
pg_last_error() );

  $query = 'SELECT record_book, name, psp_ser, psp_num ' .
           'FROM students ORDER BY name';

  // Поскольку условия WHERE в запросе нет, воспользуемся
  // функцией pg_query.
```

```
  $result = pg_query( $dbconn, $query );

  // Обратите внимание, что теперь используется статус
  // PGSQL_TUPLES_OK, а не PGSQL_COMMAND_OK.
  if ( pg_result_status( $result ) != PGSQL_TUPLES_OK )
  {
    die ( 'Ошибка: ' . pg_last_error( $dbconn ) );
  }

  // Выбираем из полученной выборки строки одну за другой
  // в цикле. Параметр NULL означает, что берется
  // следующая строка из выборки. Параметр PGSQL_ASSOC
  // означает, что строка из таблицы будет представлена
  // в виде хеш-массива (ассоциативного массива).
  while ( $student_info = pg_fetch_array( $result, NULL,
                                          PGSQL_ASSOC ) )
  {
    // Отсутствие индекса (пустые квадратные скобки)
    // означает, что новое значение будет добавлено
    // в конец массива. Получаем двухмерный массив,
    // поскольку $student_info -- это также массив
    // (ассоциативный).
    $students_list[] = $student_info;
  }

  pg_free_result( $result );
  pg_close( $dbconn );

  return $students_list;
}

/*
  Добавление одной записи в таблицу students.
*/
function insert_student()
{
  // Подключаемся к базе данных.
  $dbconn = pg_connect( CONNECTION_STRING )
    or die( 'Не могу подключиться к базе данных: ' .
            pg_last_error() );

  // Сформируем запрос, в котором вместо фактических
  // значений стоят переменные-заменители: $1, $2 и т. д.
  $query = "INSERT INTO students " .
           "( record_book, name, psp_ser, psp_num ) " .
           "VALUES ( $1, $2, $3, $4 )";

  // Выполняем обращение к базе данных. Фактические
  // значения полей передаем в виде массива.
  $result =
    pg_query_params( $dbconn, $query,
```

```
                    array( $_POST[ 'record_book' ],
                           $_POST[ 'name' ],
                           $_POST[ 'psp_ser' ],
                           $_POST[ 'psp_num' ] )
                  );

  // Проверим состояние полученного результата.
  if ( pg_result_status( $result ) != PGSQL_COMMAND_OK )
  {
    die ( 'Ошибка: ' . pg_last_error( $dbconn ) );
  }

  // Число фактически добавленных записей (1 или 0).
  $num_rows = pg_affected_rows( $result );

  // Освободим память.
  pg_free_result( $result );

  // Закроем соединение с сервером баз данных.
  pg_close( $dbconn );

  return $num_rows;
}

/*
  Обновление одной записи в таблице students.
*/
function update_student()
{
  $dbconn = pg_connect( CONNECTION_STRING )
    or die( 'Не могу подключиться к базе данных: ' .
pg_last_error() );

  $query = "UPDATE students " .
           "SET ( record_book, name, psp_ser, psp_num ) = " .
           "( $1, $2, $3, $4 ) " .
           "WHERE record_book = $5";

  $result =
    pg_query_params( $dbconn, $query,
                    array( $_POST[ 'record_book' ],
                           $_POST[ 'name' ],
                           $_POST[ 'psp_ser' ],
                           $_POST[ 'psp_num' ],
                           $_POST[ 'current_record_book' ] )
                  );

  if ( pg_result_status( $result ) != PGSQL_COMMAND_OK )
  {
    die ( 'Ошибка: ' . pg_last_error( $dbconn ) );
  }
```

```
  // Число фактически обновленных записей (1 или 0).
  $num_rows = pg_affected_rows( $result );

  pg_free_result( $result );
  pg_close($dbconn);

  return $num_rows;
}

?>
```

Наша учебная программа содержит еще два модуля: «Справочник учебных дисциплин» и «Экзаменационные оценки». Они так же, как и модуль «Информация о студентах», разделены на три файла каждый:

– **disciplines.php**, **disciplines_view.php**, **disciplines_db.php**;

– **progress.php**, **progress _view.php**, **progress _db.php**.

С целью уменьшения объема учебного пособия мы не приводим здесь тексты этих файлов. Они включены в электронное приложение к пособию.

Следующий шаг в нашей программе изучения основ web-разработки – ознакомление с языком JavaScript. Важным понятием, которое широко используется при программировании на JavaScript, является **Document Object Model** (**DOM**) – объектная модель документа. Эта модель является также интерфейсом программирования для HTML. Каждый HTML-элемент является объектом, для которого определены свойства, методы и события. Когда HTML-документ загружается в браузер, создаваемые объекты объединяются в древовидную структуру. Для доступа к ним предусмотрены специальные методы, одним из которых является **getElementById**. Получив доступ к объекту, можно изменить его свойства, например, записать в абзац новый текст:

```
document.getElementById( "demo" ).innerHTML = "Hello World!";
```

С помощью кода на JavaScript можно изменять свойства HTML-объектов, удалять их и создавать новые. В программах на языке JavaScript, которые мы покажем, также используется HTML DOM. Файлы, содержащие код на JavaScript, как правило, имеют расширение **.js**.

В файле **students.js** содержатся функции, вызываемые непосредственно из HTML-документа, который будет сформирован при выполнении файла **students.php**.

Файл **students.js**

```
/*
  Система: "Учет успеваемости студентов"
  Модуль:  функции на языке JavaScript, используемые
           в модуле "Студенты"
*/

/*
  Проверка значений, введенных в поля формы.
*/
function check_student_form()
{
  // Абзац (параграф), в который будут записываться
  // сообщения об ошибках при вводе значений.
  var err = document.getElementById( "error" );

  // Доступ у значениям полей формы можно получить
  // по идентификатору этой формы и идентификаторам ее полей.
  var r_book = document.forms[ "student_form" ][ "r_book" ];
  var f_name = document.forms[ "student_form" ][ "f_name" ];
  var p_ser  = document.forms[ "student_form" ][ "p_ser" ];
  var p_num  = document.forms[ "student_form" ][ "p_num" ];

  var ckeck_ok = true;

  // Запишем пустую строку в этот абзац.
  err.innerHTML = "";

  // Сделаем проверки только на предмет отсутствия
  // введенных значений.
  if ( r_book.value == "" )
  {
    err.innerHTML +=
      "Поле 'Номер зачетной книжки' не заполнено<br />";
    ckeck_ok = false;
  }

  if ( f_name.value == "" )
  {
    err.innerHTML += "Поле 'Ф. И. О.' не заполнено<br />";
    ckeck_ok = false;
  }

  if ( p_ser.value == "" )
  {
    err.innerHTML +=
      "Поле 'Серия паспорта' не заполнено<br />";
    ckeck_ok = false;
  }
```

```
  if ( p_num.value == "" )
  {
    err.innerHTML +=
      "Поле 'Номер паспорта' не заполнено<br />";
    ckeck_ok = false;
  }

  return ckeck_ok;
}

/*
  Удаление записи из таблицы students.
*/
function delete_student( script_name, record_book, row )
{
  // Параметры: script_name -- имя скрипта, вызвавшего
  //                           эту функцию;
  //            record_book -- номер зачетной книжки;
  //            row -- идентификатор строки в таблице
  //                   студентов на экране.

  var script_params;

  // Строка-параметр содержит номер зачетной книжки
  // для той записи, которую нужно удалить.
  script_params = "record_book=" + record_book;

  // Вызываем функцию, которая выполнит обращение
  // к web-серверу.
  delete_record( script_name, script_params, row );
}
```

Кроме рассмотренного файла **students.js** в нашей учебной программной системе предусмотрен еще файл **university.js**, который используется и с другими модулями, а не только с **students.php**.

Файл **university.js**

```
/*
  Система: "Учет успеваемости студентов"
  Модуль:  функции на языке JavaScript, используемые
           в разных модулях
*/

/*
  Выполнение удаления одной записи из таблицы.
  Здесь используется технология AJAX, позволяющая получить
  данные из базы данных и на их основе обновить лишь часть
  HTML-страницы, не переформируя ее заново всю полностью.
*/
```

```
function delete_record( script_name, script_params, row )
{
  // Параметры:
  // script_name -- имя скрипта, который будет выполнять
  // удаление записи из таблицы;
  // script_params -- строковая переменная, содержащая
  // параметры скрипта;
  // row -- идентификатор строки в таблице на экране.

  // Создаем стандартный объект XMLHttpRequest.
  var xhttp = new XMLHttpRequest();

  // Сформируем функцию-обработчик для события
  // onreadystatechange.
  // Эта функция вызывается не в момент ее создания,
  // а при изменении состояния объекта XMLHttpRequest,
  // т. е. при отправке запроса web-серверу и при
  // получении ответа от него.
  xhttp.onreadystatechange = function()
  {
    // Если получен ответ от сервера и его статус --
    // 200 (OK)...
    if ( this.readyState == 4 && this.status == 200 )
    {
      try
      {
        // Попытаемся разобрать информацию, полученную
        // от web-сервера, и сформировать из нее объект
        // JavaScript. В случае штатной работы она должна
        // быть в формате JSON.
        var server_response =
            JSON.parse( this.responseText );
      }
      // Если web-сервер отправил неструктурированную
      // информацию, то JSON.parse выдаст ошибку.
      // Обработаем эту ситуацию.
      catch ( err )
      {
        // Выведем сообщение, полученное от web-сервера,
        // во всплывающем окне. Это будет сообщение
        // об ошибке, которое возвращает PostgreSQL.
        display_message( this.responseText );

        return;
      }

      // В функциях delete_discipline (см. файл
      // disciplines_db.php), delete_student (см. файл
      // students_db.php), delete_exam (см. файл
      // progress_db.php) мы с помощью команды echo
      // выводили на стандартный вывод JSON-объект,
```

```
      // в котором в поле rows содержалось число удаленных
      // записей. Этот JSON-объект в результате попадает
      // от web-сервера в эту процедуру.
      if ( server_response.rows == '1' )
      {
        // Используя иерархию DOM, удалим из таблицы
        // на экране строку с идентификатором, значение
        // которого равно параметру row. Таблица
        // на экране будет автоматически перерисована.
        var table_row = document.getElementById( row );
        // Здесь атрибут parentNode означает родительский
        // объект для строки, т. е. таблицу.
        table_row.parentNode.removeChild( table_row );

        display_message( "<p class=\"messages\">Запись
удалена</p>\n" );
      }
      else  // server_response.rows == '0'
      {
        display_message( "<p class=\"messages\">Запись не уда-
лена</p>\n" );
      }
    }
  };

  // Готовим запрос к web-серверу. Параметр "POST"
  // означает название метода, используемого для отправки
  // запроса. Параметр script_name означает имя скрипта,
  // который будет запущен web-сервером для получения
  // ответа на наш запрос. Параметр true означает, что
  // запрос выполняется асинхронно, т. е. наша функция
  // delete_record не ожидает ответа, а "надеется"
  // на созданный ею обработчик событий: когда ответ от
  // web-сервера придет, тогда он и будет обработан.
  xhttp.open( "POST", script_name, true );
  xhttp.setRequestHeader( "Content-Type",
                          "application/x-www-form-urlencoded"
);

  // Непосредственная отправка запроса методом POST.
  // Строка-параметр содержит имена и значения ключевых
  // полей таблицы, из которой удаляется запись.
  xhttp.send( "operation=delete&" + script_params );
}

/*
  Вывод сообщения во всплывающем окне.
*/
function display_message( message )
{
  // Параметры: message -- текст сообщения.
```

```
  // Создадим окно. Не будем указывать URL и имя окна,
  // поэтому первые два параметра будут пустыми строками.
  var err_win =
    window.open( "", "",
                 "menubar=no, status=no, titlebar=no, " +
                 "width=800, height=200" );
  // Начинаем создание документа в новом окне.
  err_win.document.open();

  // "Пишем" HTML-теги в созданном окне.
  // Используем символ "\n" -- в конце, а пробелы --
  // в начале символьных строк, содержащих HTML-теги,
  // чтобы форматирование полученного исходного текста
  // HTML-документа было аккуратным.
  err_win.document.write( "<!DOCTYPE html>\n" );
  err_win.document.write( "<html
xmlns=\"http://www.w3.org/1999/xhtml\">\n" );
  err_win.document.write( "<head>\n" );
  err_win.document.write( "  <meta http-equiv=\"Content-Type\"
" +
                          "content=\"text/html; charset=utf-
8\" />\n" );
  err_win.document.write( "  <link rel=\"stylesheet\" " +
                          "href=\"university.css\"
type=\"text/css\" />\n" );
  err_win.document.write( "</head>\n" );
  err_win.document.write( "<body>\n" );
  err_win.document.write( message );
  err_win.document.write( "\n  <div>\n" );
  err_win.document.write( "    <input type=\"button\"
value=\"Закрыть\" " +
                          "onclick=\"self.close()\" />\n" );
  err_win.document.write( "  </div>\n" );
  err_win.document.write( "</body>\n" );
  err_win.document.write( "</html>\n" );

  err_win.document.close(); // Завершаем создание документа.
}
```

Скопируйте в каталог **/usr/local/apache2/htdocs** следующие файлы: **university.php**, **university.css**, **university.js**, **head.php**, **foot.php**, **students.php**, **students_view.php**, **students_db.php**, **students.js**, **progress.php**, **progress_view.php**, **progress_db.php**, **progress.js**, **disciplines.php**, **disciplines_view.php**, **disciplines_db.php**, **disciplines.js**.

Запустите web-сервер:

```
/usr/local/apache2/bin/apachectl  start
```

К этому моменту у вас уже должна быть создана база данных edu. В адресной строке браузера введите:

```
localhost/university.php
```

Для изучения работы программы удобно воспользоваться возможностью браузера демонстрировать исходный текст HTML-страницы. Поэтому, активизировав тот или иной модуль и получив в окне браузера новую страницу, вы можете с помощью контекстного меню выбрать режим, который называется приблизительно так: «Показать исходный код страницы». Сопоставление полученной страницы с ее исходным кодом может помочь лучше разобраться в работе программ как на языке PHP, так и на языке JavaScript.

Схема взаимодействия PHP-программ с web-сервером 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 задана в теге <form> с помощью атрибута action. Полученные данные web-сервер передает этой программе на ее стандартный ввод. Программа должна уметь принять такие данные. Язык PHP сконструирован таким образом, что программа, написанная на нем, получает эти данные автоматически, без специальных действий со стороны программиста. В результате программа получает данные, отправленные с помощью метода GET, в глобальном массиве $_GET, а данные, отправленные с помощью метода POST, в глобальном массиве $_POST. Затем эта программа, в нашем случае, например, **students_view.php**, 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 программам на языках PHP и JavaScript, которые должны взаимодействовать между собой. Конфигурация web-сервера Apache по умолчанию такова, что файлы перечисленных видов должны находиться в его подкаталоге **htdocs**.

В процессе работы web-программ могут возникать ошибки. Поскольку web-сервер Apache ведет файлы-журналы **access_log** и **error_log** (на компьютерном жаргоне они называются «логи»), которые находятся в

подкаталоге **logs**, то в случае возникновения ошибок рекомендуем вам обращаться к этим файлам-журналам.

Контрольные вопросы и задания

1. С помощью документации изучите назначение всех опций команды apachectl. В чем различие между опциями restart и graceful?

2. Для чего предназначены каскадные таблицы стилей CSS? Каким образом можно подключить внешнюю таблицу стилей к HTML-странице?

3. Каким образом можно включить код на языке JavaScript непосредственно в HTML-страницу? А как можно подключить внешний файл с кодом на языке JavaScript?

4. Более детально изучите технологию AJAX, позволяющую получать данные от web-сервера без полного переформирования web-страницы. За информацией можно обратиться, например, на сайт www.w3schools.com.

5. Каким образом можно посмотреть исходный код HTML-страницы, представленной в окне браузера?

6. Основательно разберитесь с такими CSS-свойствами, как margin и padding. Как они влияют на визуализацию HTML-элементов?

7. Посмотрите примеры использования HTML, CSS, JavaScript на сайте www.w3schools.com.

8. С помощью документации по языку PHP детально изучите функции для взаимодействия с СУБД PostgreSQL. Познакомьтесь также и с технологией PDO (PHP Data Objects).

9. В представленной версии учебной программы все проверки введенных пользователем данных выполняются в браузере средствами языка JavaScript. Добавьте проверки данных, выполняемые на стороне сервера средствами языка PHP.

10. Добавьте какой-либо модуль к нашей учебной программе, например, модуль «Преподаватели». Соответственно измените структуру таблицы «Экзаменационные оценки» (progress), добавив в нее столбец «Идентификатор преподавателя».

10. Средства создания интерфейса пользователя

Заключительная глава пособия посвящена средствам разработки интерфейса пользователя. В настоящее время преобладающей тенденцией становится использование web-интерфейсов. Однако настольные интерфейсы также используются. Поэтому мы представим одну из наиболее известных свободно распространяемых библиотек, предназначенных для создания кроссплатформенных приложений с графическим интерфейсом, – wxWidgets. С другой стороны, консольные приложения также существуют и иногда 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

Использование библиотеки **ncurses** продемонстрируем на примере ОС FreeBSD. При установке этой операционной системы данная библиотека также устанавливается, если вы инсталлируете исходные тексты программ и утилит операционной системы. Располагается она в каталоге **/usr/src/contrib/ncurses**.

Если в вашей системе этой библиотеки по какой-то причине нет, то можно ее установить. Получить исходные тексты можно с помощью ftp-клиента:

```
ftp -a ftp.gnu.org
```

Подключившись к серверу, выполните ряд команд в среде ftp-клиента:

```
cd gnu/ncurses
ls
bin
get ncurses-6.0.tar.gz
close
quit
```

Если команда **ls** покажет, что есть более новая версия библиотеки, то загрузите ее.

Поскольку вы уже приобрели некоторый опыт установки программных продуктов в среде FreeBSD, то мы опишем процедуру компиляции библиотеки **ncurses** очень кратко.

Войдите в систему с правами пользователя root. Извлеките файлы из архива:

```
tar xzvf ncurses-6.0.tar.gz
cd ncurses-6.0
```

Для обеспечения возможности использования отладчика **gdb** при изучении библиотеки **ncurses** служит параметр **-g** компилятора языка C. Рекомендуем также отключить оптимизацию объектного кода с помощью параметра **-O0** (заглавная буква «O» и цифра 0).

```
./configure CFLAGS="-g -O0" > conf_log 2>&1 &
make > make_log 2>&1 &
```

Перейдите в каталог с тестовыми примерами и запустите какую-нибудь программу, например, **xmas**:

```
cd test
./xmas
```

Если при запуске программ-примеров вы получаете ошибку (вместо xterm может быть указан другой терминал)

```
Error opening terminal: xterm
```

тогда добавьте при запуске утилиты **configure** параметр **--enable-termcap** и повторите компиляцию:

```
./configure --enable-termcap CFLAGS="-g -O0" > conf_log 2>&1 &
make > make_log 2>&1 &
cd test
./xmas
```

В состав пакета **ncurses** входит большое число программ-примеров, которые находятся в подкаталоге **test**. Мы рекомендуем вам изучить эти примеры и использовать их в качестве учебного пособия.

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 нужно использовать 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 подкаталоге **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
```

10.2. Библиотека wxWidgets

Войдите в систему с правами пользователя root.

Чтобы установить эту библиотеку в среде ОС Debian, вам необходимо предварительно установить пакеты **pkg-config** и **libgtk-3-dev**, если они еще не установлены:

```
apt-get install pkg-config
apt-get install libgtk-3-dev
```

Загрузите последнюю стабильную версию библиотеки с сайта http://www.wxwidgets.org.

1. Распакуйте архивный файл в каталог, например, **/home/DISTRIB**.

ПРИМЕЧАНИЕ. Как и прежде, вместо конкретных цифр, отражающих номер версии в имени архивного файла, мы будем использовать символы «x».

```
tar xjvf wxWidgets-3.x.x.tar.bz2
```

2. Перейдите в каталог **wxWidgets-3.x.x**:

```
cd wxWidgets-3.x.x
```

ПРИМЕЧАНИЕ. В подкаталоге **docs/gtk** есть файл **install.txt**, в котором процедура установки описана подробно.

3. В каталоге **wxWidgets-3.x.x** 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_gtk
```

4. Перейдите в этот подкаталог и запустите из него утилиту **configure** со следующими параметрами (команду необходимо вводить в одну строку):

```
cd build_gtk

../configure --with-gtk=3 --enable-shared --enable-debug
--enable-debug_gdb --enable-debugreport --enable-debug_info
--enable-debug_flag --enable-monolithic
--with-libiconv-prefix=/usr/local --disable-optimise
--enable-intl --enable-unicode --enable-utf8
CFLAGS="-g -O0" CXXFLAGS="-g -O0" > conf_log.txt 2>&1 &
```

Обратите внимание на две точки перед именем утилиты configure – 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 поддержкой средств интернационализации и стандарта Unicode. 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. 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
./minimal
```

7. Возвратитесь в каталог, из которого вы запускали утилиту **configure**, т. е. **build_gtk**, и выполните установку библиотеки в системный каталог:

```
cd ../..
make install > make_install_log 2>&1 &
```

8. Обновите конфигурацию динамического загрузчика **ld.so** (**ld-linux.so**), чтобы он «увидел» новую библиотеку:

```
ldconfig
```

Библиотека **wxWidgets** сопровождается хорошей документацией, которую можно загрузить с того же сайта.

Контрольные вопросы и задания

1. На основе библиотеки **ncurses** модифицируйте интерфейс программы, выполняющей выборку информации из базы данных под управлением СУБД PostgreSQL.

2. Изучите работу библиотеки **ncurses** с помощью отладчика **gdb**.

3. Скомпилируйте все примеры программ, содержащиеся в подкаталоге **samples** установочного комплекта библиотеки **wxWidgets**, и ознакомьтесь с их работой.

4. С помощью утилиты **ldd** посмотрите список разделяемых библиотек, от которых зависит программа **minimal** в среде ОС Debian.

5. На основе программы **minimal** напишите свою программу.

Заключение

Итак, вы познакомились с рядом инструментальных средств и технологий, используемых в среде операционных систем Debian и FreeBSD. 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.

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!

Библиографический список

1. Брайант, Р. Компьютерные системы: архитектура и программирование [Текст] : пер. с англ. / Р. Брайант, Д. О'Халларон. – СПб. : БХВ-Петербург, 2005. – 1104 с.

2. Грофф, Дж. Р. SQL. Полное руководство [Текст] : пер. с англ. / Джеймс Р. Грофф, Пол Н. Вайнберг, Эндрю Дж. Оппель. – 3-е изд. – М. : Вильямс, 2015. – 960 с.

3. Дейт, К. Дж. Введение в системы баз данных [Текст] : пер. с англ. / Крис Дж. Дейт. – 8-е изд. – М. : Вильямс, 2005. – 1328 с.

4. Лав, Р. Linux. Системное программирование [Текст] : пер. с англ. / Р. Лав. – 2-е изд. – СПб. : Питер, 2014. – 448 с.

5. Митчелл, М. Программирование для Linux. Профессиональный подход [Текст] : пер. с англ. / М. Митчелл, Дж. Оулдем, А. Самьюэл. – М. : Вильямс, 2002. – 288 с.

6. Прата, С. Язык программирования С. Лекции и упражнения [Текст] : пер. с англ. / Стивен Прата. – 6-е изд. – М. : Вильямс, 2015. – 928 с.

7. Реймонд, Э. Искусство программирования для Unix [Текст] : пер. с англ. / Э. Реймонд. – М. : Вильямс, 2005. – 544 с.

8. Рочкинд, М. Программирование для UNIX [Текст] : пер. с англ. / М. Рочкинд. – М. : Русская редакция ; СПб. : БХВ-Петербург, 2005. – 704 с.

9. Скляр, Д. PHP. Сборник рецептов [Текст] : пер. с англ. / Д. Скляр, А. Трахтенберг. – 3-е изд. – СПб. : Питер, 2015. – 784 с.

10. Шварц, Р. Изучаем Perl [Текст] : пер. с англ. / Р. Шварц, Т. Феникс, б. де фой. – 5-е изд. – СПб. : Символ-Плюс, 2009. – 384 с.

11. Шварц, Р. Perl: изучаем глубже [Текст] : пер. с англ. / Р. Шварц, Б. Фой, Т. Феникс. – 2-е изд. – СПб. : Символ-Плюс, 2007. – 320 с.

12. Шилдт, Г. Полный справочник по C [Текст] : пер. с англ. / Герберт Шилдт. – 4-е изд. – М. : Вильямс, 2002. – 704 с.

13. PostgreSQL [Электронный ресурс] : официальный сайт / The PostgreSQL Global Development Group. – Электрон. дан. – Б. м., [1996–]. – Режим доступа: http://www.postgresql.org. – Загл. с экрана.

14. Postgres Professional [Электронный ресурс] : российский производитель СУБД Postgres Pro : официальный сайт / Postgres Professional. – Электрон. дан. – Б. м., [2015–2017]. – Режим доступа: http://postgrespro.ru. – Загл. с экрана.

15. W3 Schools [Электронный ресурс] : the world's largest web development site. – Электрон. дан. – [Б. м.] : [Б. и.], 2015. – Режим доступа: http://www.w3schools.com, свободный. – Загл. с экрана. – Яз. англ.

Учебно-теоретическое издание

МОРГУНОВ Евгений Павлович
МОРГУНОВА Ольга Николаевна

Технологии разработки программ в среде операционных систем Linux и FreeBSD

Вводный курс

Учебное пособие

Редактор *Х. Х. Хxxxxxxxx*
Оригинал-макет и верстка *Х. Х. Хxxxxxxxxxx*

Подписано в печать __.__.2018. Формат 60×84/16. Бумага офсетная.
Печать плоская. Усл. п. л. _,_. Уч.-изд. л. _,_. Тираж __ экз.
Заказ . С __/__.

Санитарно-эпидемиологическое заключение
№ 24.04.953.П.000032.01.03. от 29.01.2003 г.

Редакционно-издательский отдел
Сиб. гос. ун-та науки и технологий им. М. Ф. Решетнева.
Опечатано в отделе копировально-множительной техники
Сиб. гос. ун-та науки и технологий им. М. Ф. Решетнева.
660014, г. Красноярск, просп. им. газ. «Красноярский рабочий», 31.