Глава 12. Пакеты, библиотеки, модули

Учебник по Perl

Содержание [скрыть]

Пакеты

В главе 11 мы упомянули о том, что область действия переменных ограничена пакетом. Рассмотрим этот вопрос более подробно.

Итак, *пакет* – это способ создания собственного изолированного пространства имен для отдельного отрезка программы. Каждый фрагмент кода Perl-программы относится к некоторому пакету. Объявление

```
package NAMESPACE;
```

определяет пакет NAMESPACE. Ключевое слово package является именем встроенной функции, в результате обращения к которой компилятору предписывается использовать новое пространство имен. Область действия объявления пакета определяется аналогично области видимости локальных переменных, объявленных при помощи функций ту () или local (). Она распространяется либо до следующего объявления пакета, либо до конца одной из таких единиц программы:

- Подпрограммы;
- блока операторов, заключенного в фигурные скобки;

- строки, переданной на выполнение функции eval ();
- файла.

Область действия зависит от того, в каком месте вызвана для объявления пакета функция package o. Все идентификаторы, встретившиеся внутри этой области, принадлежат к пространству имен текущего пакета. Исключение составляют идентификаторы лексических переменных, созданных при помощи функции ту ().

Объявление пакета может быть сделано несколько раз в разных точках программы. Каждое объявление означает переключение на новое пространство имен. По умолчанию предполагается, что основная программа всегда начинается с объявления пакета

```
package main;
```

Таким образом, те переменные, которые мы называем в языке Perl глобальными, в действительности представляют собой переменные, чьи идентификаторы по умолчанию принадлежат пакету main.

К переменной из другого пакета можно обратиться, указав перед ее именем префикс, состоящий из имени этого пакета, за которым следуют два двоеточия: $PackageName: :name. Такие имена условимся называть *квалифицированными* именами. Если имя пакета отсутствует, предполагается имя main, т. е. записи $: :var и $main: :var обозначают одну и ту же переменную.

Специальная лексема _PACKAGE_ служит для обозначения имени текущего пакета.

```
#!/usr/bin/perl
$x=_PACKAGE_;
print "package $x:\n";
print "\$x= $x\n";
print "\$two::x= $two::x\n";
print "\$three::x= $three::x\n";
eval 'package two; $x=_PACKAGE_; print " package $x:\n"; print "\$x= $x\n"; print "\$main::x=
$main::x\n"; print "\$three::x= $three::x\n";';
print "package $x:\n"; print "\$x= $x\n";
package three;
$x=_PACKAGE_;
print "package $x:\n";
print "\$x= $x\n";
print "\$main::x= $main::x\n";
print "\$two::x= $two::x\n";
package main;
print "package $x:\n";
print "\$x= $x\n";
```

```
print "\$two::x= $two::x\n";
print "\$three::x= $three::x\n";
```

В результате выполнения будут выведены следующие значения:

```
package main: $x= main
$two::x= $three::x=
package two: $x= two
$main::x= main $three::x=
package main: $x= main
package three: $x= three $main: :x= main $two::x= two
package main: $x= main $two::x= two $three::x= three
```

В данном примере используются три пакета, каждый со своим пространством имен: main, two, three. В каждом пакете определена переменная $x, значение которой совпадает с именем пакета. С пакетом main связаны следующие отрезки программы:

- от начала программы до вызова функции evai ();
- после вызова функции evai о до объявления пакета package three;
- после явного объявления пакета package main до конца файла, содержащего данную программу.

Для выражения, выполняемого функцией evai (), определено собственное пространство имен two. Оно действует только в пределах этого выражения. Если бы внутри функции evai о не был определен собственный пакет two, все действия внутри нее были связаны с пакетом, в котором функция evai () была скомпилирована, т. е. с пакетом main.

С пакетом three связана часть программы от объявления package three до объявления package main.

В каждом пакете происходит обращение к переменным из двух других пакетов при помощи указания соответствующего префикса имени переменной.

Компилятор создает для каждого пакета отдельное пространство имен. Переменным $x из разных пакетов присваиваются их значения по мере выполнения соответствующего кода программы. Вот почему при первом обращении из пакета main к переменным two: :$x и $three::x их значения еще не определены.

Таблицы символов

С каждым пакетом связана *таблица символов.* Она представляет собой хеш-массив, имя которого образовано из имени пакета, за которым следуют два двоеточия. Например, таблица символов пакета main хранится в хеш-массиве %main::. Ключами этого хеш-массива являются идентификаторы переменных, определенных в пакете, значениями – значения типа typeglob, указывающие на гнездо, состоящее из одноименных переменных разных типов: скаляр, массив, хеш-массив, функция, дескриптор файла или каталога.

Тип typeglob, с которым мы уже сталкивались в главе И – это внутренний тип данных языка Perl, который используется для того, чтобы при помощи одной переменной типа typeglob сослаться на все одноименные переменные разных типов. Признаком типа typeglob является символ "*". Если переменной типа typeglob присвоить значение другой переменной типа:

```
typeglob:
*y = *x;
```

то для всех переменных с именем x: $x, @x, %x, &x, будут созданы псевдонимы $y, @y, %y, &y соответственно. Можно создать псевдоним только для переменной определенного типа, например, для скалярной:

```
*y = \$x;
```

Ключами в хеш-массиве таблицы символов являются все идентификаторы, определенные в пакете. Поэтому можно получить данные о переменных всех типов, определенных в пакете, проверяя значения элементов этого хеш-массива. Например, чтобы вывести имена всех переменных, определенных в пакете main, можно использовать следующий код.

```
* !/usr/bin/perl
my ($key, $item) ;
print "Таблица символов пакета main:\n";
for $key (sort keys %main::) {
local *myglob = $main::{$key};
print "определен скаляр \$$key = $myglob\n" if defined $myglob; i^'.Med @myglob)
опт "определен массив \@$key :\n"; for $item (0..$#myglob) {
p> ••••••• \$$key [$item] = $myglob[$item] \n";
} }
if (defined %myglob) {
print "определен хеш-массив \%$key :\n";
for $item (sort keys %myglob) {
print "\$$key {$item} = $myglob{$item}\n";
} } print "определена функция $key()\n" if defined Smyglob;
}
```

При помощи типа typegiob можно создавать скалярные псевдоконстанты. Например, после присваивания

```
*P1 = \3.14159265358979;
```

выражение $PI обозначает операцию разыменования ссылки на константу. Его значением является значение самой константы 3.14159265358979. Значение $PI нельзя изменить, так как это означало

бы попытку изменить константу.

В таблицу символов пакета, отличного от main, входят только идентификаторы, начинающиеся с буквы или символа подчеркивания. Все остальные идентификаторы относятся к пакету main. Кроме того, к нему относятся следующие начинающиеся *с* буквы идентификаторы: STDIN, STDOUT, STDERR, ARGV, ARGVOUT, ENV, INC, sic. Например, при обращении внутри некоторого пакета pack к хеш-массиву %ENV подразумевается специальный хеш-массив %ENV основного пакета main, даже если имя main не используется в качестве префикса для обозначения принадлежности идентификатора ENV.

Конструктор и деструктор пакета *BEGIN и END*

Конструктором в объектно-ориентированном программировании называется специальная подпрограмма, предназначенная для создания объекта. *Деструктором* называется подпрограмма, вызываемая для выполнения завершающих действий, связанных с ликвидацией объекта: закрытие файлов, вывод сообщений и т. д.

Для создания пакета, как мы знаем, требуется только его объявление (в том числе, предполагаемое по умолчанию объявление package main). Вместе с тем, существуют специальные подпрограммы, выполняющие функции инициализации и завершения пакета. По аналогии их можно назвать конструкторами и деструкторами пакета, хотя никаких пакетов они не создают и не удаляют. Это подпрограммы BEGIN и END. При описании этих подпрограмм ключевое слово sub, необходимое при объявлении обычной подпрограммы, можно опустить. Таким образом, /синтаксис, подпрограмм BEGIN, END имеет вид:

```
BEGIN (block) END {block}
```

Подпрограмма BEGIN выполняется сразу после своего определения до завершения компиляции оставшейся части программы. Попробуйте запустить интерпретатор perl в интерактивном режиме. Если ему передать строку

```
print "Привет!";
```

то он напечатает ее только после того, как обнаружит во входном потоке признак конца файла (например, комбинацию <Ctrl>+<D>). Если же в интерактивном режиме определить конструктор пакета

```
BEGIN {print "Привет!"};
```

то вывод строки "Привет!" будет осуществлен немедленно. Это свойство конструктора можно использовать, чтобы в начале пакета определять или импортировать имена из других пакетов *(см. раздел 12.3).* Затем эти имена будут влиять на процесс компиляции оставшейся части пакета.

Можно определить несколько блоков BEGIN внутри файла, они будут выполняться один за другим в порядке определения.

Подпрограмма END выполняется настолько поздно, насколько это возможно, т. е. при завершении работы интерпретатора. Можно указать несколько блоков END, при этом они будут выполняться в порядке, обратном определению.

```
END {
print "Завершаем работу, до свидания\n";
} . . BEGIN {
print "Привет, начинаем работу\n"; }
print "Это тело программы\n"; BEGIN {
print "Еще один блок BEGIN после блока END\n"; }
```

Здесь сознательно выбран не совсем естественный порядок следования конструкторов и деструкторов BEGIN и END в тексте программы, чтобы подчеркнуть, в каком порядке они будут вызываться. Вывод выглядит так:

```
Привет, начинаем работу
Еще один блок BEGIN после блока END
Это тело программы
Завершаем работу, до свидания
```

Автозагрузка

При попытке обратиться к функции из некоторого пакета, которая в нем не определена, интерпретатор завершает работу с выдачей сообщения об ошибке. Если же в этом пакете определить функцию с именем AUTOLOAD, то при вызове из пакета несуществующей функции вместо нее будет вызвана функция AUTOLOAD с параметрами, переданными при вызове несуществующей функции. При этом интерпретатор perl продолжает выполнение программы. Полное имя несуществующей функции с указанием имени пакета сохраняется в переменной $AUTOLOAD из того же пакета, что и функция AUTOLOAD. Например, для основного пакета main можно определить функцию AUTOLOAD, как в следующем примере.

```
#!/usr/bin/perl sub AUTOLOAD {
print "Функция $AUTOLOAD не определена\n"; } print "Начало работы\n";
f();
print "Конец работы\n";
```

Функция f (), в отличие от функции AUTOLOAD, не определена в пакете main, поэтому в результате выполнения данной программы будут выведены сообщения:

```
Начало работы
Функция main::f не определена
Конец работы
```

Этот пример достаточно тривиальный, но он дает представление об использовании функции AUTOLOAD. В состав дистрибутива Perl входят стандартные модули, многие из которых содержат собственные, достаточно сложные определения функции AUTOLOAD, которые можно рассмотреть в качестве более сложного примера.

Библиотеки

Подпрограммы, как мы отметили в цачале главы, служат для объединения группы операторов с целью их повторного использования. Следующим шагом является объединение группы подпрограмм и их сохранение в отдельном файле для последующего использования другими программами. Для реализации этой задачи в языке Perl имеются два механизма: библиотеки и модули.

Исторически первым средством являются библиотеки, появившиеся в версии Perl 4. Библиотека представляет собой пакет, областью действия которого является отдельный файл. Иными словами, библиотека – это файл, содержащий в качестве первой строки объявление пакета.

```
package package_name;
```

Имя файла библиотеки обычно имеет расширение pi.

> **Замечание**
>
> После появления модулей (см. *раздел 12,3)* термин "библиотека" часто используют в широком смысле для обозначения всего множества модулей в составе Perl, содержащего разнообразные коллекции подпрограмм. Если не оговорено противное, мы будем использовать слово "библиотека" для обозначения файла библиотеки в соответствии с версией Perl 4.

Для использования библиотеки в основной программе, ее следует подключить к последней при помощи директивы крмпилятора require.

Ключевое слово require служит для обозначения встроенной функции Perl. Фактически обращение к функции require () используется в качестве директивы компилятора. Поэтому дальше мы будем использовать применительно к ключевому слову require оба термина: "функция" и "директива-компилятора". Выясним, какой смысл имеет эта директива.

Функция *require()*

```
require [EXPR]
```

загружает внешние функции из библиотеки Perl во время выполнения. Она используется для того, чтобы сделать библиотеку подпрограмм доступной для любой Perl-программы.

Если параметр EXPR отсутствует, вместо него используется специальная переменная $_.

Если параметр является числом, это соответствует требованию, что для выполнения данного сценария необходим интерпретатор perl с номером версии, не меньшим, чем значение параметра.

Таким образом, сценарий, который требует Perl версии 5.005, может иметь в качестве первой строки:

```
require 5.005;
```

Более ранние версии Perl вызовут немедленное завершение интерпретатора с выдачей сообщения об ошибке.

Если параметр является строкой, функция require включает в основную программу библиотечный файл, задаваемый параметром EXPR. Логика работы функции require соответствует следующему коду:

```
sub require {
my($filename) = @_; return 1 if $INC{$filename}; . my($realfilename,$result); ITER: {
foreach $prefix (@INC) {
$realfilename = "$prefix/$filename"; if (-f $realfilename) {
$result = do $realfilename; last ITER; } }
die "Can't find $filename in \@INC"; }
die $@ if $@;
die "$filename did not return true value" unless $result; $INC{$filename} = $realfilename;
return $result; }
```

Замечание

Специальный встроенный массив OINC содержит имена каталогов, в которых следует искать сценарии Perl, подлежащие выполнению в конструкциях do *filename,* require **или use.**

Первоначально содержит:

- имена каталогов, переданные при запуске интерпретатору perl в качестве параметра ключа -i;
- имена библиотечных каталогов по умолчанию (зависят от операционной системы);
- символическое обозначение текущего каталога ".".

Специальный встроенный хеш-массив % INC содержит по одному элементу для каждого файла, включенного при помощи do или require. Ключом является имя файла в том виде, как оно указано в качестве аргумента функций do или require, а значением – его полное маршрутное имя.

Встретив директиву require *myfile,* интерпретатор perl просматривает специальный хеш-массив %INC, проверяя, не был ли файл (rayfile уже включен ранее при помощи функций require или do. Если да, то выполнение функции require завершается. Таким образом файл под одним именем может быть включен только один раз. В противном случае интерпретатор просматривает каталоги, имена которых содержатся в специальном массиве @INC, в поисках файла *ту file.* Если файл найден, он выполняется, иначе директива require завершает выполнение с выдачей сообщения об ошибке:

```
Can't find myfile in @INC
```

Замечание

Замечание об использовании имени файла В Директиве require EXPR. Обычно имена библиотечных файлов имеют суффикс ".pi", например, myfile.pl. Интерпретатор perl воспринимает точку"." в качестве знака операции конкатенации двух строк "myfile" и "pi" и пытается найти файл myfiiepl. Во избежание подобных ошибок имя файла в директиве require следует заключать в кавычки:

```
require "myfile.pl";
```

Если аргумент EXPR является словом без суффиксов, не заключенным в кавычки, то директива require предполагает, что оно имеет суффикс ".рт", чтобы облегчить загрузку стандартных модулей, имеющих расширение " .рт".

Создание и подключение библиотечного файла

Для создания собственной библиотеки следует выполнить следующие шаги. П Создать каталог для хранения библиотечных файлов.

- Сохранить наборы подпрограмм в отдельных файлах-библиотеках. Переместить библиотечные файлы в специально созданный для них каталог.
- В конец каждого библиотечного файла поместить строку " i;". Смысл этого действия заключается в следующем. Как видно из приведенного текста, включение библиотечного файла в основную программу осуществляется через его выполнение функцией do:

```
$result = do $realfilename;
```

Значение $resuit, возвращаемое функцией require, должно быть ненулевым, что является признаком успешного выполнения кода инициализации. Простейший способ получить ненулевое значение — добавление в конец каждого библиотечного файла строки "1,-".

- В основной программе использовать директиву require, указав в качестве ее аргументов имена требуемых библиотечных файлов.
- Добавить в массив @INC имя каталога, содержащего библиотечные файлы, либо при запуске основной программы передать это имя интерпретатору perl при помощи ключа *-i.* \ '

Создадим библиотечный файл myiib.pl и поместим его в каталог /usr/temp/periiib. Файл myiib.pl содержит единственную подпрограмму Numof Args (), которая выводит число аргументов, переданных ей при вызове.

```
# библиотечный файл /usr/temp/perllib/mylib.pl sub NumOfArgs {
return $#_+!; }
```

```
I;
```

Создадим файл основной программы mymain.pi:

```
#!/usr/bin/perl
unshift (@INC, "/usr/temp/perllib");
require "mylib.pl";
print "Число аргументОв=", NumOfArgs(1,2,3,4), "\n";
```

В результате выполнения файла mymain.pi будет выведена строка

```
Число аргументов=4
```

Обратите внимание на выполнение всех шагов, необходимых для создания и подключения библиотеки.

Модули

Дальнейшим развитием понятия библиотеки явилось понятие модуля, возникшее в версии Perl 5. Модуль представляет собой библиотеку подпрограмм, обладающую дополнительными свойствами по сравнению с библиотеками Perl 4. Он позволяет управлять экспортом своих имен в другие программы, объявляя, какие из них экспортируются по умолчанию, а какие должны быть явно указаны в соответствующем операторе вызывающей программы.

Под *экспортом* мы здесь понимаем предоставление возможности другим модулям импортировать символы из пространства имен данного модуля. Соответственно под *импортом* мы понимаем включение в собственное пространство имен символов, экспортируемых другим модулем.

Для целей управления экспортом каждый модуль должен располагать методом import о и определить специальные массивы @EXPORT и @EXPORT_OK.

(Понятие "метод" используется в объектно-ориентированном программировании, которое обсуждается в главе 13.)

Вызывающая программа обращается для импорта символов^ к методу import () экспортирующего модуля.

Специальный массив @ EXPORT содержит идентификаторы, экспортируемые по умолчанию.

Специальный массив @EXPORT_OK содержит идентификаторы, которые будут экспортироваться только в том случае, если они явно указаны в списке импорта вызывающей программы.

С появлением модулей появилась новая директива для их подключения к основной программе. Эта директива реализуется функцией use ().

Функция *use()*

Относительно ключевого слова use можно сказать то же самое, что и относительно ключевого слова require. Оно служит для обозначения встроенной функции Perl. Фактически же обращение к функции use о используется в качестве директивы компилятора, поэтому мы также будем использовать применительно к ключевому слову use оба термина: "функция" и "директива компилятора".

Функция

```
use ()
use Module [LIST] use VERSION
```

служит для загрузки модуля во время компиляции.

Директива use автоматически экспортирует имена функций и переменных в основное пространство имен текущего пакета. Для этого она вызывает метод import () импортируемого модуля. Механизм экспорта имен устроен таким образом, что каждый экспортирующий модуль должен иметь свой метод import (), который используется программой, импортирующей имена. Метод import () должен быть определен в самом экспортирующем модуле или наследован у модуля Exporter. Большинство модулей не имеют своих собственных методов import (), вместо этого они экспортируют его из модуля Exporter.

Логику работы директивы use можно описать одной строкой:

```
BEGIN { require Module; import Module LIST; }
```

Здесь значением параметра Module должно быть слово без суффиксов, не заключенное в кавычки.

Если первый аргумент директивы use является числом, он обозначает номер версии интерпретатора perl. Если номер версии текущего интерпретатора perl меньше, чем значение VERSION, интерпретатор выводит сообщение об ошибке и завершает работу.

Конструктор пакета BEGIN вызывает немедленное выполнение подпрограммы require {) и метода import о до завершения компиляции оставшейся части файла.

Выше мы рассмотрели логику работы функции require(). Она загружаете память файл Module.pm, выполняя его при помощи функции dot). Затем метод import() модуля Module.pm импортирует в вызывающую программу имена, определенные в Module.pm, в соответствии со списком LIST.

Если список импорта LIST отсутствует, из Module будут импортированы те имена, которые перечислены в специальном массиве @EXPORT, определенном В самом Module.

Если список импорта задан, то в вызывающую программу из модуля Module будут импортированы только имена, содержащиеся в списке LIST.

Создание и подключение модуля

Для создания модуля MyModuie следует создать пакет и сохранить его в файле MyModuie. pm. Расширение .pm является признаком того, что данный файл является модулем Perl.

В следующем примере мы создадим собственный модуль MyModuie, содержащий одну функцию MyArgs o , одну скалярную переменную $MyArgs, один массив емуАгдз и один хеш-массив %муАгдз. Затем создадим файл основной программы MyMain.pi, экспортирующий модуль MyModuie, используя директиву use.

Файл модуля MyModuie. pm:

```
package MyModuie;
require Exporter;
@ISA = 'qw(Exporter) ;
SEXPORT = qw(MyArgs) ;
@EXPORT_OK = qw($MyArgs @MyArgs %MyArgs);
sub MyArgs {
my ($x, $i);
SMyArgs = @_;
$MyArgs = $#MyArgs + 1;
foreach $x (@MyArgs) {
$MyArgs{$x}=++$i ; }
}
```

Файл основной вызывающей программы MyMain.pi: ^

```
tt!/usr/bin/perl
use MyModuie qw(:DEFAULT $MyArgs @MyArgs %MyArgs);
MyArgs one, two, three, four;
print "число аргументов=$MyArgдз\n";
print "массив аргументов: @MyArgs\n";
print "хеш-массив аргументов:\n";
foreach $k (keys %MyArgs) {
print "\$MyArgs{$k}=$MyArgs{$k} "; }
```

Первые пять строк файла MyModuie.pm являются стандартными для определения модуля Perl. Их можно использовать в качестве образца при создании собственных модулей.

Первая строка служит определением пакета.

Вторая строка осуществляет включение встроенного модуля Exporter. Так предоставляется возможность наследовать метод import, реализованный в этом модуле, и использовать стандартные соглашения для задания списка импорта в вызывающей программе.

Третья строка определяет массив @ISA, состоящий из одного элемента, содержащего название пакета Exporter. С каждым пакетом ассоциируется свой массив @ISA, включающий имена других пакетов, представляющих классы. Иногда интерпретатор встречает обращение к методу, не определенному в текущем пакете. Он ищет этот метод, просматривая пакеты, определенные в массиве @ISA текущего пакета. Таким образом в языке Perl реализован механизм наследования.

В четвертой и пятой строках определяются имена, которые будут экспортироваться за пределы модуля. Специальный массив OEXPORT содержит имена, экспортируемые по умолчанию. В четвертой строке указывается, что из данного модуля по умолчанию будет экспортировано имя функции MyArgs.

В пятой строке специальный массив @EXPORT_OK содержит имена, которые будут экспортироваться только в том случае, если они явно указаны в списке импорта вызывающей программы. В примере это имена переменной $MyArgs, **массива** @MyArgs **и ассоциативного массива** %MyArgs.

Функция MyArgs подсчитывает число своих аргументов и запоминает его в переменной $MyASrgs. Затем она помещает аргументы в массив @MyArgs и формирует ассоциативный массив %MyArgs, в котором ключами являются имена аргументов функции MyArgs, а значениями – их порядковые номера.

К основной программе MyMain.pl модуль MyModuie подключается при помощи директивы use. Директива use содержит список импорта

```
qw(:DEFAULT $MyArgs @MyArgs %MyArgs)
```

Обычно список импорта включает в себя имена переменных и функций. Кроме того, он может содержать некоторые управляющие им спецификации. Спецификация : DEFAULT означает включение в список импорта *всех* элементов специального массива @EXPORT. В нашем случае это значит, что в список импорта будет добавлено имя функции MyArgs, содержащееся в списке @EXPORT. Кроме того, список импорта явно содержит имена $MyArgs, емуАгдз и %MyArgs. Экспорт этих имен по явному запросу вызывающей программы разрешен модулем MyModuie путем их включения в список

```
@EXPORT_OK.
```

В результате выполнения основной программы MyMain.pi будет получен вывод:

```
число аргументов=4
массив аргументов: one two three four хеш-массив аргументов: .
$MyArgs{one}=l $MyArgs{three}=3 $MyArgs{two}=2 $MyArgs{four}=4
```

Функция *по()*

Существует функция no (), противоположная по своему назначению функции use o и также выполняющая роль директивы компилятора

```
no Module LIST
```

Директива по отменяет действия, связанные с импортом, осуществленные ранее директивой use, вызывая метод unimport () Module LIST.

Стандартные модули Perl

В данном разделе мы рассмотрели вопросы, связанные с созданием модулей. В состав дистрибутива Perl входит большой набор стандартных модулей, предназначенных для выполнения определенных функций) Помимо них, существует огромная коллекция модулей, разработанных программистами всего мира, известная как коллекция модулей CPAN (Cbmprehensive Perl Archive Network). Ее копия поддерживается в сети Internet на многих анонимных ftp-серверах. В качестве отправной точки можно обратиться по адресу http://www.perl.com/CPAN/modtiles/. Список стандартных модулей и категорий модулей CPAN приведен в приложении 2. Здесь же мы в заключение рассмотрим специальный вид стандартных модулей – прагма-библиотеки.

Прагма-библиотеки

Многие языки программирования позволяют управлять процессом компиляции посредством директив компилятора. В языке Perl эта возможность реализована при помощи так называемых прагма-библиотек. В современной терминологии, связанной с программированием, слово "pragma" используется для обозначения понятия, смысл которого в русском языке выражается сочетанием "директива компилятора". В языке Perl термин "pragma" обозначает модуль, содержащий коллекцию подпрограмм, используемых на этапе компиляции. Его назначение – передать компилятору информацию о том, как модифицировать процесс компиляции. Поскольку сочетание "библиотека директив компилятора" звучит несколько тяжеловато, мы используем для обозначения таких модулей название "прагма-библиотека".

Как и остальные модули, прагма-библиотека подключается к основной программе при помощи директивы use и выполняет функцию директивы компилятора. Область действия большинства таких директив ограничена, как правило, блоком, в котором они встречаются. Для отмены соответствующей директивы используется функция по.

Например, для ускорения выполнения некоторых отрезков программы можно заставить компилятор использовать целочисленную арифметику вместо принятой по умолчанию арифметики с плавающей точкой, а затем снова вернуться к последней.

```
#!/usr/bin/perl
print "Арифметика с плавающей точкой: 2/3= ", 2/3, "\n";
use integer;
print "Целочисленная арифметика: 2/3= ",2/3, "\n";
no integer;
print "Возврат к арифметике с плавающей точкой: 2/3= ", 2/3, "\n";
```

В результате выполнения данного примера будет получен вывод

```
Арифметика с плавающей точкой: 2/3= 0.666666666666667
Целочисленная арифметика: 2/3= 0
Возврат к арифметике с плавающей точкой: 2/3= 0.666666666666667
```

В дистрибутивный комплект Perl входит стандартный набор прагма-библиотек. Некоторые из них представлены в табл. 12.1.

Таблица 12.1, *Некоторые прагма-библиотеки*

Прагма-библиотека	Назначение
diagnostics	Включает режим диагностики с выдачей подробных сообщений
integer	Применение целочисленной арифметики вместо арифметики с плавающей точкой
lib	Позволяет добавлять элементы в специальный массив @INC во время компиляции
overload	Режим переопределения операций Perl, например, директива package Number; use overload "+" => \&add; определяет функцию Number : : add ( ) в качестве операции сложения
sigtrap	Директива, позволяющая управлять обработкой сигналов в UNIX
strict	Режим ограниченного использования "опасных" конструкций Perl use strict "refs"; генерирует ошибку выполнения при использовании символических ссылок use strict "vars"; генерирует ошибку компиляции при попытке обращения к переменной, которая не была объявлена при помощи директивы use vars, локализована при помощи функции ту() или не является квалифицированным именем use strict "subs";

	генерирует ошибку компиляции при попытке использовать идентификатор, который не заключен в кавычки, не имеет префикса типа и не является именем подпрограммы, за исключением тех случаев, когда он заключен в фигурные скобки, или стоит слева от символа => use strict; эквивалентно заданию всех трех рассмотренных выше ограничений
subs	Служит для предварительного объявления подпрограмм, указанных в списке: use subs qw(subl sub2 sub3);
vars	Служит для предварительного объявления переменных, указанных в списке use vars qw($scal @list %hash) ; после чего их можно использовать при включенной директиве use strict, не опасаясь возникновения ошибки компиляции

Вопросы для самоконтроля

1. Что такое пакет?
2. Верно ли, что пакет должен всегда занимать отдельный файл?
3. Что такое таблица символов?
4. Сколько таблиц символов могут быть связаны с одним файлом, функцией, блоком операторов, заключенным в фигурные скобки?
5. Какие функции выполняют конструктор и деструктор пакета BEGIN и END?
6. Как определить имя текущего пакета?
7. Для чего нужна функция AUTOLOAD?
8. Что такое библиотека?
9. Назовите действия, необходимые для создания библиотечного файла.
10. Что такое модуль? В чем разница между модулем и библиотекой?
11. Объясните назначение массивов SEXPORT и @EXPORT_OK.
12. Чем похожи и чем отличаются функции use () и require () ?
13. Объясните, как создать модуль и подключить его к вызывающей программе.
14. Объясните назначение функции по ().
15. Что такое прагма-библиотека?

Упражнения

1. Напишите программу, которая выводит таблицу символов для пакета, заданного ее аргументом.
2. Создайте два модуля. Модуль Modi должен содержать подпрограмму reverselist o, которая переставляет в обратном порядке элементы в массиве, переданном ей в качестве параметра. Модуль Mod2 должен содержать массив eiist. Используйте оба модуля в основной программе, которая при помощи подпрограммы reverseiist o переставляет элементы в массиве eiist.