第3学时　控制程序流

在第2学时中，我们学习了程序的语句、运算符和表达式等内容。该学时中的所有例子有一个共同点，那就是所有的语句都是从上向下执行的，并且只执行一次。

我们之所以使用计算机，原因之一是计算机非常擅长执行重复任务，一次又一次地重复，永不疲倦，也不会诱发手腕综合症。迄今为止，还没有任何办法来告诉 Perl，让它“执行该任务X次”，或者“重复执行该任务，直到任务完成为止”。在本学时中，我们要介绍 Perl的控制结构。使用这些控制结构，就能够将语句组合成所谓的“语句块”，并且重复执行这些语句组，直到它们完成你想要完成的工作。

计算机擅长的另一项工作是能够迅速作出决策。如果计算机每次作出决策时都要询问你，那么你一定会感到非常讨厌，而且也说明计算机太笨了。检索和读取电子邮件，可使你的计算机作出上百万个决策，而这些决策并不是你想要处理的。比如，如何组合网络信息，什么颜色构成屏幕上的每个像素，你收到的电子邮件应该如何分类和显示，当鼠标光标稍稍移动了一点儿时应该怎么办，如此等等，这些都需要迅速作出决策。所有这些决策又是由其他决策构成的，而且其中的一些决策需要每秒钟作出数千次。在本学时中，我们将要介绍条件语句。使用这些语句，你就能够编写代码块，这些代码块是根据你的Perl程序中作出的决策来执行的。

在本学时中，我们将要介绍下列基本概念：

- 语句块。
- 运算符。
- 循环。
- 标号。
- 程序执行后退出Perl。

3.1　语句块

Perl中最简单的语句组合称为块。若要将语句组合在一个块中，只需要用一组匹配的花括号将语句括起来即可，如下所示：

```
{
    statement_a;
    statement_b;
    statement_c;
}
```

在语句块中，语句像前面已经介绍过的情况那样，从上向下执行。在语句块中，你可以拥有另一些语句块，请看下面的例子：

```
{
    statement_a;
    {
        statement_x;
        statement_y;
    }
}
```

语句块的格式与Perl的其他程序格式一样，是自由随意的格式。语句与花括号可以如下面所示放在同一行上，也可以放在若干不同行上，并且可以根据你的需要，采用任何一种对齐方式，只要使用匹配的一组花括号即可：

```
{ statement;  { another_statement; }
       { last_statement;        } }
```

虽然你可以按照你的喜好来安排语句块，但是，如果仅仅将各个语句凑合在一起，那么程序就很难阅读。采用好的缩进方式虽然并不是必须的，但是却能建立便于阅读的 Perl程序。

程序中出现的孤立语句块称为裸语句块。不过大多数情况下，你遇到的语句块是附属在其他Perl语句后面的。

3.2 if语句

若要根据Perl程序中的某个条件来控制语句是否执行，通常可以使用 if语句。下面显示了if语句的句法：

```
if (expression) BLOCK
```

该语句的工作原理是：如果表达式计算的结果是真（ TRUE），那么该代码块运行；如果该表达式的计算结果是假（ FALSE），那么代码块不运行。请记住，该代码块包含了花括号。请看下面这个例子：

```
if ( $r == 5 ) {
    print 'The value of $r is equal to 5.';
}
```

该代码中被测试的表达式是 $r == 5。符号==是个等式运算符。如果等式两边的两个操作数（$r和5）的数值是相等的，那么该表达式被视为真，同时 print语句执行。如果$r不等于5，那么print语句不执行。

如果一个条件是真，那么 if语句也能运行代码，否则，如果该条件不是真，则运行另一个代码。该结构称为if-else语句，它的句法类似下面的样子：

```
if (expression)       #If expression is true...
    BLOCK             # ...this block of code is run.
   else
    BLOCK             # Otherwise this block is run.
```

只有当表达式是真的时候，表达式后面的语句块才运行；如果表达式不是真，那么 else后面的语句块运行。现在请看下面这个例子：

```
$r=<STDIN>;  chomp $r;
if ($r == 10) {
    print '$r is 10';
} else {
    print '$r is something other than 10...';
    $r=10;
    print '$r has been set to 10';
}
```

请注意，在上面这个例子中，为了将一个值赋予 $r，使用一个赋值运算符=。为了测试$r的值，使用等式运算符 ==。这两个运算符有着很大的差别，其作用是不同的。在你的程序中不要混淆它们的用法，因为调试非常困难。请记住，运算符=用于赋值，而==则用于测试一个等式。

建立if语句的另一种方法是使用多个表达式，然后根据哪个表达式是真，来运行代码：

```
if (expression1)      # If expression1 is true ...
    BLOCK1            # ...run this block of code.
elsif (expression2)   # Otherwise, if expression2 is true...
    BLOCK2            # ...Run this block of code.
else
    BLOCK3            # If neither expression was true, run this.
```

可以像下面这样来读取上面这个语句块：如果标号为 expression1的表达式是真，那么语句块BLOCK1就运行。否则，控制权转给 elsif，对expression2进行测试，如果该表达式是真，则运行BLOCK2。如果expression1和expression2都不是真，那么BLOCK3运行。下面是一个实际的Perl代码的例子，用于显示这个句法的实际情况：

```
$r=10;
if ($r==10) {
    print '$r is 10!';
} elsif ($r == 20) {
    print '$r is 20!';
} else {
    print '$r is neither 10 nor 20';
}
```

3.2.1 其他关系运算符

到现在为止，我们都是使用等式运算符 ==来比较if语句中的数字的量值。实际上 Perl还有一些运算符可以用来进行数字值的测试，表 3-1列出了这种运算符的大部分。

表3-1 数字关系运算符

运 算 符	举 例	说 明
==	$x == $y	如果$x等于$y，则为真
>	$x > $y	如果$x大于$y，则为真
<	$x < $y	如果$x小于$y，则为真
>=	$x >= $y	如果$x大于或者等于$y，则为真
<=	$x <= $y	如果$x小于或者等于$y，则为真
!=	$x != $y	如果$x不等于$y，则为真

若要使用这些运算符，只需要将它们放在你的程序中需要测试数字值之间的关系的任何位置，就像程序清单3-1中所示的if语句那样。

程序清单3-1 一个小型猜数字游戏

```
1:   #!/usr/bin/perl -w
2:
3:   $im_thinking_of=int(rand 10);
4:   print "Pick a number:";
5:   $guess=<STDIN>;
6:   chomp $guess;    # Don't forget to remove the newline!
7:
8:   if ($guess>$im_thinking_of) {
9:                print "You guessed too high!\n";
10:   } elseif ($guess < $im_thinking_of) {
11:               print "You guessed too low!\n";
12:   } else {
13:              print "You got it right!\n";
14:   }
```

第1行：这一行是Perl程序第一行的标准格式，它指明你想要运行的解释程序和用于激活警告特性的-w开关。与第1学时的内容比较，你会发现那里的第一行与这里的第一行稍有不同。

第3行：函数（rand 10）取出0至10 之间的一个数字， int（）对该数字范围进行舍位，这样只有0到9的整数被赋予$im_thinking_of。

第4~6行：这一行让用户进行猜测，将它赋予 $guess，并删除结尾处的换行符。

第8~9行：如果$guess大于$im_thinking_of中的数字，那么这些行将输出一个相应的消息。

第10~11行：否则，如果$guess小于$im_thinking_of中的数字，那么这些行便输出该消息。

第12~13行：剩下的惟一选择是用户猜测该数字。

表3-1中的运算符只能用于测试数字值。使用这些运算符可以测试你可能不想要的运行特性中的非字母数据的结果。请看下面这个例子：

```
$first="Simon";
$last="simple";
if ($first == $last) {     # == is not what you want!
    print "The words are the same!\n";
}
```

$first和$last中存放的两个值实际上是要测试它们之间是否相等。对它们进行测试的原因在第1学时中已经做了说明。当 Perl期望数字值的时候，如果使用了非数字字符串，那么这些字符串的计算结果将是 0。因此，上面这个 if表达式在Perl看来就像是：if （ 0 == 0）。这个表达式的计算结果是真，这可能不是你想要的结果。

如果程序中的警告特性被打开，那么当程序运行时，用 ==运算符来测试两个字母字符值（上面代码段中的 simple和Simon），就会产生一个警告消息，以提醒你存在这个问题。

如果你想要测试非数字值，你可以使用另一组 Perl运算符，表3-2列出了这些运算符。

表3-2 字母关系运算符

运 算 符	举 例	说 明
eq	$s eq $t	如果$s等于$t，则为真
gt	$s gt $t	如果$s大于$t，则为真
lt	$s lt $t	如果$s小于$t，则为真
ge	$s ge $t	如果$s大于或者等于$t，则为真
le	$s le $t	如果$s小于或者等于$t，则为真
ne	$s ne $t	如果$s不等于$t，则为真

这些运算符通过从左到右观察每个字符，然后按照 ASCII的顺序对它们进行比较，来确定“ 大于 ”和“ 小于 ”。这意味着字符串按照升序进行排序，大多数标点符号放在最前面，然后是数字，接着是大写字母，最后是小写字母。例如， 1506大于Happy，而Happy又大于happy。

3.2.2 “ 真 ”对于Perl意味着什么

到现在为止，我们已经介绍了“ 如果该表达式是真 ..... ”或者“ .....计算的结果为真.....，”等情况，但是尚未介绍 Perl认为的“ 真 ”的正式定义。关于什么是真，什么不是真， Perl有几个简短的规则，当你认真思考一下这些规则之后，就会认识到这些规则的意义。下面就是这些规则：

- 数字0为假。
- 空的字符串（“ ”）和字符串“ 0 ”为假。
- 未定义值undef为假。
- 其他东西均为真。

明白了吗？需要注意的另一个问题是：当你测试一个表达式，看它是真还是假时，该简化表达式，调用函数，使用运算符，数学算式简化为表达式，如此等等，然后转换成标量值，以便进行计算，确定它是真还是假。

请考虑这些规则，然后看一下表3-3。在查看答案之前，猜测一下该表达式是真还是假。

表3-3 真还是假的例子

表 达 式	真 还 是 假
0	假。数字0为假
10	真。这是个非0数字，因此是真
9>8	真。关系运算符将根据你的期望返回真或假
-5+5	假。这个表达式计算后得出的结果是0，而0是假
0.00	假。这个数字是0的另一种表示法，0x0，00，0b0和0e00也是一样
""	假。在上面介绍的规则中明确讲到这个表达式是假
""	真。引号中有一个空格，这意味着它们并不是完全空的
"0.00"	真。真奇怪！它已经是个字符串了，而不是“ 0 ”或“ ”。因此它是真
"00"	也是真，原因与“ 0.00 ”相同
"0.00" + 0	假。在这个表达式中，对0.00+0进行了计算，结果是0，因此这是假

到现在为止，我们只介绍了if语句的关系运算符。实际上你可以使用任何表达式，按照你想要使用的方法来计算它是真或假：

```
# The scalar variable $a is evaluated for true/false
if ($a) {  ...  }

# Checks the length of $b.  If nonzero, the test is true.
if (length($b)) { ....  }
```

undef这个值在Perl中是个特殊值。尚未设置的变量均拥有undef这个值，并且有些函数在运行失败时也返回undef。它不是0，也不是一个普通的标量值，它有几分特殊。当在一个if语句中测试为真时，undef总是计算为假。如果你试图使用数学表达式中的undef值时，它将被视为0。

使用尚未设置的变量，往往是程序出错的标志。如果你运行的程序激活了警告特性，那么在一个表达式中使用undef值或者将未定义的值传递给某个函数，就会使Perl发出一个警告，即Use of uninitialized value（使用了未经初始化的值）。

3.2.3 逻辑运算符

当你编写程序时，有时可能需要类似下面这样的某种代码，即如果$x是真，并且$y是真，那么执行这项操作，但是，如果$z是真，则不要执行该操作。可以将这个例子的代码编写成一系列的if语句，不过这并不是个出色的代码：

```
if ($x) {
    if ($y) {
        if ($z) {
       # do nothing
   } else {
```

```
            print "All conditions met.\n";
        }
    }
}
```

Perl拥有一套完整的运算符，可以将真和假的语句组合在一起，这些运算符称为逻辑运算符。表3-4显示了各个逻辑运算符。

表3-4 逻辑运算符一览表

运 算 符	替 代 名	举 例	分 析
&&	and	$s && $t	只有当$s和$t都是真时，才是真
		$q and $p	只有当$q和$p都是真时，才是真
||	or	$a || $b	如果$a或$b是真，则为真
		$c or $d	如果$c或$d是真，则为真
!	not	! $m	如果$m不是真，则为真
		not $m	如果$m不是真，则为真

使用表3-4中的运算符，可以将前面这个代码段改写得更加简洁，如下所示：

```
if ($x and $y and not $z ) {
    print "All conditions met.\n";
}
```

用逻辑运算符连接起来的表达式将自左向右进行计算，直到能够为整个表达式确定一个真或假的值。请看下面的代码段：

```
1:   $a=0; $b=1; $c=2; $d="";
2:
3:   if ($a and $b) {  print '$a and $b are true'; }
4:   if ($d or $b) { print 'either $d or $b is true'; }
5:   if ($d or not $b or $c)
6:         { print '$d is true, or $b is false or $c is true'; }
```

第1行：这一行代码为变量赋予一个默认值。

第3行：$a首先被计算。它的结果是假，因此 and表达式不可能是真。 $b从来不计算，它也不必计算，因为表达式的真是在计算 $a之后知道的。 Print没有执行。

第4行：$d首先被计算。它的计算结果是假。即使 $d是假，该表达式仍然可能是真，因为它包含一个逻辑 or，因此下一步要观察 $b。$b的结果是真，因此该表达式是真，同时 print开始运行。

第5行：$d首先被计算。它的结果是假。即使 $d是假，该表达式可能仍然是真，正如第 4行中的情况一样，因为它包含一个逻辑 or。接着， $b的真（为1，因此是真）被求反，因此该表达式变成假。 or语句的真尚不能确定，因此 $c被计算。 $c的计算结果是真，因此整个表达式是真，print开始运行。

这个运行特性（即一旦确定表达式是真，立即停止逻辑表达式的计算）称为短路。整个特性可供Perl程序员用来借助逻辑运算符创建简单的流控制语句，而完全不使用 if语句：

```
$message="A and B are both true."
($a and $b) or $message="A and B are not both true.";
```

在上面这个例子中，如果 $a或$b中有一个是假，那么 or右边的表达式必须计算，并且消息被改变。如果 $a和$b都是真，那么 or必须是真，并且不必计算右边的表达式。整个表达式的真值根本不使用。这个例子使用 and和or运算符的短路副作用来操作 $message。

运算符 || 和or并不完全相同。它们的差别在于 || 的运行优先级要高于or。这意味着在一个表达式中，|| 要比or更早地进行计算。这与数学表达式中乘法的计算要先于加法的情况是一样的。这个规则也适用于 &&/and和！/not。如果你对计算顺序没有把握，可以使用括号来确保表达式的计算顺序的正确。

Perl的逻辑运算符有一个有趣的属性，那就是它们不仅仅返回真或假，它们实际上返回计算得出的最后值。例如，表达式5 && 7的计算结果并不只是返回真，而是返回7。这样你就可以创建下面这个代码：

```
# Set $new to old value if $old is true,
# otherwise use the string "default".
$new=$old || "default";
```

这比下面这个代码更加简洁：

```
$new=$old;
if (! $old) {  # was $old empty (or false)?
   $new="default";
}
```

3.3 循环

在本学时的开头我们说过，仅仅根据条件来进行决策和运行代码是不够的。在许多情况下，需要一次又一次重复运行一段代码。程序清单 3-1中显示的程序练习并没有太大的趣味，因为只是进行一次猜测，它是个毫无意义的游戏。如果你希望能够进行多次猜测，那么必须按照一定的条件重复执行一些代码段，这就是循环所要达到的目的。

3.3.1 用while进行循环

最简单的一种循环是 while循环。只要表达式是真的， while循环就会重复执行该代码段。while循环的句法类似下面的形式：

```
while (expression) BLOCK
```

当Perl遇到while语句时，它就计算该条件。如果条件计算的结果是真，代码块就运行。当运行到代码块的结尾时，表达式被重新计算，如果结果仍然是真，代码块重复执行，如程序清单3-2所示：

程序清单3-2 while循环示例

```
1:   $counter=0;
2:   while ($counter < 10 ) {
3:       print "Still counting...$counter\n";
4:       $counter++;
5:   }
```

第1行：$counter被初始化为0。

第2行：表达式$counter < 10被计算。如果计算的结果是真，该语句块中的代码就运行。

第4行：$counter的值递增1。

第5行：花括号}给第2行上以 {为开始的代码块做上结束标号。这时， Perl返回到while循

环的顶部，并重新计算该条件表达式。

3.3.2 使用for循环

for语句是Perl循环结构中最复杂和最有用的语句。它的句法类似下面的形式：

```
for ( initialization; test; increment) BLOCK
```

for语句分为3个部分，即initialization、test和increment，它们之间用分号隔开。当 Perl遇到一个for循环时，便出现下面这个操作顺序：

- 初始化表达式被计算。
- 测试表达式被计算。如果它的计算结果的真，代码块就运行。
- 当该代码块执行结束后，便执行递增操作，并再次计算测试表达式。如果该测试表达式的计算结果仍然是真，那么代码块再次运行。这个进程将继续下去，直到测试表达式的计算结果变为假为止。

下面是for循环的一个例子：

```
for( $a=0; $a<10; $a=$a+2 ) {
    print "a is now $a\n";
}
```

在上面这个代码段中， $a设置为0，执行测试表达式 $a<10，发现其结果为真。循环的本身输出了一条消息。然后递增语句 $a=$a+2开始运行，它将$a的值递增2。测试语句再次执行，循环重复运行。这个特殊的循环将重复运行，直到 $a的值达到10为止。这时测试语句变为假，for循环后面的程序将继续运行。

不必使用for语句来进行计数，它只是进行重复操作，直到测试表达式变为假为止。要记住，for语句的3个组成部分中的每一个都是可有可无的，但是两个分号是必不可少的。下面这个for语句漏掉了某些元素，不过它仍然是有效的：

```
$i=10;            # initialization
for( ; $i>-1; ) {
    print "$i..";
    $i--;           # actually, a decrement.
}
print "Blast off!\n";
```

3.4 其他流控制工具

用循环和条件语句来控制你的程序运行方式是不错的，但是还需要其他的流控制语句，以便提高程序的可读性。例如， Perl有一些语句可以用来提前退出 while循环，跳过for循环的某些部分，在代码块结束之前退出 if语句，或者甚至在不到结束的时候就退出程序。使用本节中介绍的某些结构，就能够使你的 Perl程序变得更加简洁和便于阅读。

3.4.1 奇特的执行顺序

if语句还可以使用另一种句法。如果在 if语句块中只有一个表达式，那么该表达式实际上可以放在if语句的前面。因此不要写成下面这个语句：

```
if (test_expression) {
    expression;
}
```

可以写成:

```
expression if (test_expression);
```

下面是该语句变形的两个例子:

```
$correct=1 if ($guess == $question);
print "No pi for you!"  if ( $ratio != 3.14159);
```

在Perl代码中使用该句法通常是为了清楚起见。有时，如果在条件之前看到它的作用，那么阅读代码就会更加容易。 if前面的表达式必须是个单一表达式。 if语句也必须后跟一个分号。

3.4.2 明细控制

除了使用语句块、for、while、if以及其他流控制语句来控制代码块以外，还可以使用 Perl 语句来控制语句块中的流程。

为你提供这种控制能力的最简单的语句是 last。last语句能够使当前正在运行的最里面的循环块退出。请看下面这个例子：

```
while($i<15) {
    last if ($i==5);
    $i++;
}
```

last语句能够在$i的值是5时使while循环退出，而不是在通常while测试的结果是假时退出。当你拥有多个嵌套的循环语句时， last将退出当前正在运行的循环。

程序清单3-3用于找出其乘积等于140的所有小于100的两个数，比如2与70、4与35等，不过查找的效率不太高。这里需要注意的是 last语句。当找到一个乘积时，其结果被输出，里面的循环（在$i上重复运行的循环）退出，外面的循环继续运行（通过递增 $i），并返回里面的循环。

程序清单3-3

```
1:   for($i=0; $i<100; $i++) {
2:       for($j=0; $j<100; $j++) {
3:           if ($i * $j == 140) {
4:               print "The product of $i and $j is 140\n";
5:               last;
6:           }
7:       }
8:   }
```

next语句使得控制权被重新传递给循环的顶部，同时下一个循环的重复运行则开始进行，如果该循环尚未结束的话：

```
for($i=0; $i<100; $i++) {
    next if (not $i % 2);
    print "An odd number=$i\n";
}
```

该循环将输出从0到98之间的所有偶数。如果 $i不是偶数，那么next语句将使该循环通过它的下一个迭代运行过程。表达式 $i % 2是$i除以2的余数。在这个例子中， print语句被跳过了。编写这个循环时使用的一种更加有效的方法是按 2这个值来递增 $i，但是这将无法展示next的作用，是不是？

3.4.3 标号

Perl允许你给语句块和某些循环语句（for、while）加上标号。也就是说，可以在语句块或语句的前面放置一个标识符：

```
MYBLOCK: {
}
```

上面这个语句块的标号是MYBLOCK。标号名使用的约定与变量名基本相同，不过有一个很小的差别，那就是标号名不像变量，它不带%、$和@之类的标识符。应该确保标号名与Perl的内置关键字不能冲突。就样式而言，如果标号名全部使用大写字母，那么这是最好的。不应该使它与目前和将来的Perl关键字的形式发生任何冲突。for和while语句也都可以带有标号。

```
OUTER: while($expr ) {
    INNER: while($expr) {
        statement;
    }
}
```

last、redo和next语句都可以带有一个标号，作为参数。这样就可以退出一个特殊的语句块。在程序清单3-4中，使用一对嵌套的for循环，找到两个140的因子。如果想在找到一个因子后立即退出该循环，需要在两个循环之间对标志变量和if语句进行复杂的安排。问题是不能从内循环中退出外循环。

```
OUTER: for($i=0; $i<100; $i++) {
    for($j=0; $j<100; $j++) {
        if ($i * $j == 140) {
            print "The product of $i and $j is 140\n";
            last OUTER;
        }
    }
}
```

现在，最后一个语句可以用来设定它想要退出哪个循环，在这个例子中，要退出的是OUTER循环。这个代码段只输出它找到的140的第一个因子。

3.4.4 退出Perl

exit语句是最后的一个流控制工具。当Perl遇到exit语句时，程序就停止执行，Perl将一个退出状态返回给操作系统。这个退出状态通常用来指明程序已经成功地完成运行。第11学时我们将要更加详细地介绍各种退出状态。现在，我们看到退出状态0意味着一切运行正常。下面是exit的一个例子：

```
if ($user_response eq 'quit') {
    print "Good Bye!\n";
    exit 0;        # Exit with a status of 0.
}
```

exit语句具有某些对你的操作系统非常重要的副作用。当一个exit执行时，所有打开的文件均被关闭，文件锁被解开，Perl分配的内存被释放给系统，Perl解释程序执行清楚的关闭操作。

3.5 练习：查找质数

在这个练习中，我们将要观察一个小程序，它用来查找和输出质数。质数是只能被1和它

本身整除的数。例如，2是个质数，3也是个质数，而4不是质数（因为它可以被1、4和2整除），如此等等。质数的数量是无限的，它们需要占用大量的计算机功能来查找。

使用文本编辑器，键入程序清单 3-4的程序，并将它保存为 Primes。不要键入行号。根据你在第1学时学习到的方法，使该程序成为可执行程序。

当你完成上面的操作时，键入下面这个命令行，设法使该程序启动运行：

```
Perl Primes
```

程序清单3-4 用于查找质数的完整源代码

```
1:   #!/usr/bin/perl -w
2:
3:   $maxprimes=20;    # Stop when you've found this many
4:   $value=1;
5:   $count=0;
6:   while($count < $maxprimes) {
7:              $value++;
8:              $composite=0;
9:   OUTER: for ($i=2; $i<$value; $i++) {
10:           for($j=$i; $j<$value; $j++) {
11:               if (($j*$i)==$value) {
12:                   $composite=1;
13:                   last OUTER;
14:               }
15:           }
16:       }
17:       if (! $composite) {
18:           $count++;
19:           print "$value is prime\n";
20:       }
21:  }
```

第1行：这一行包含到达解释程序的路径（可以修改该路径，使它适合你的系统的需要）和开关-w。请始终使警告特性处于激活状态。

第3行：$maxprimes是你想要查找的质数的最大数量。

第4行：$value是你将要测试其质数特性的值。

第5行：$count是迄今为止找到的质数的数量。

第6行：只要程序没有找到足够数量的质数， while循环就继续运行。

第7行：$value被递增，因此，经过检查符合质数要求的第一个数是 2。

第8行：$composite是for循环中使用的一个标志，用于指明找到的数是合数，不是质数。

第9~10行：for循环重复运行通过$value的所有可能的因子。如果 $value是4，那么这些循环将产生2和2、2和3、3和3。

第11~14行：$i与$j的值相乘；如果乘积是 $value，那么$value是个合数。 $composite标志被设置，同时，各个for循环均退出。

第17~20行：在for循环之后，检查$composite标志。如果它是假，那么这数是质数。然后这些行输出一个消息，同时计数器的数字递增。

这里用来查找质数的算法其运行的速度并不特别快，效率也不高，但是它很好地展示了循环的运行情况。在更好的关于数值算法的著作中，你会找到更好的方法。

3.6 课时小结

在本学时中，我们介绍了Perl的许多流控制结构。有些结构，比如if和逻辑运算符，可以用于根据真或假的值来控制程序的各个部分是否运行。其他的结构，如while、until和for，用于根据需要的次数循环运行代码段。我们还介绍了Perl的“真”究竟是什么概念，Perl中的所有测试条件实际上都使用这个概念。

3.7 课外作业

3.7.1 专家答疑

问题：我熟悉另一种编程语言C，它有一个switch（或case）语句。Perl的switch语句在哪里？

解答：Perl没有这个语句。Perl提供了各种各样的测试方法，它确定switch语句的最佳句法是非常可怕的。下面是仿真switch语句的最简单的方法：

```
if ($variable_to_test == $value1) {
    statement1;
} elsif ($variable_to_test == $value2) {
    statement2;
} else {
    default_statement;
}
```

如果在命令行提示符后面键入perldoc perlsyn，你会看到一个在线句法手册页，它包含了许多关于如何在Perl中仿真switch语句的出色方法，其中有些配有很像switch的句法。

问题：在for（while、if）语句块中我能够嵌套多少个这样的语句块？

解答：如果你的系统的内存允许的话，嵌套的语句数目是不受限制的。但是，通常情况下，如果循环中嵌套的语句太多，那么就要使用不同的方法来处理这个问题。

问题：Perl向我显示了一条消息，即Unmatched right bracket或Missing right bracket（括号不匹配，或者括号遗漏）。报告的行号是文件的结尾。我该怎么办?

解答：在你的程序的某个位置，使用了左花括号({)，但是没有右括号，或者有了右括号，没有左括号。有时Perl能够猜到你的程序中的某个位置出现了键入错误，但是有时无法猜到这样的错误。由于控制结构可以人工嵌套许多层，因此，直到Perl到达文件的结尾但没有找到对应的括号时，它才知道你产生了键入错误。好的程序编辑器（如vi、Emacs或UltraEdit）配有一个特性，可以帮助你查找不匹配的括号，你可以选择使用。

3.7.2 思考题

1) 只要条件为真，while语句就始终循环运行。当条件是假时，什么语句将循环运行？

   a. if (not) {}

   b. while (! condition) {}

2) 下面的表达式是真还是假？

```
(0 and 5) || ("0" or 0 or "") and (6 and "Hello")) or 1
```

   a. 真

   b. 假

3) 下面这个循环运行之后，$i的值是什么？

```
for($i=0; $i<=10; $i++) {     }
```

a. 10

b. 9

c. 11

3.7.3 解答

1) 答案是b。while (! condition) {}语句将循环运行，直到条件变为假为止。

2) 答案是a。该表达式可以使用下面的步骤加以简化：

（假）||（(假）and（真））or真

假||真or真

真

3) 答案是c。测试表达式是$i<=10，因此当测试的条件最后变为假时，$i必须是11。如果你在这个问题上出错，请不要担心。这是个非常常见的错误，在程序员中甚至有一个专门的名字来表示它，即篱笆桩错误或一步之差的错误。

3.7.4 实习

- 修改程序清单3-1，使游戏继续进行，直到做出一次正确的猜测。
- 程序清单3-4在查找质数时的效率实际上是非常低的。例如，它要分析 2以上的所有偶数，而这些偶数不可能是质数。请对这种方法进行修改，使这个程序运行起来更加有效。