第3章 语　句

本章讨论语句。在大多数情况下，语句是程序中可执行的部分，即语句规定了动作。C和C++的语句可分为以下几组：

- 选择语句
- 迭代语句
- 跳转语句
- 标签语句
- 表达式
- 块语句

选择语句包括if和switch (选择语句也称为条件语句)。迭代语句包括while, for和do-while，它们也称为循环语句。跳转语句包括break, continue, goto和return。标签语句包括case，default语句（将与switch语句一起讨论），以及将和goto语句一起讨论的标签语句。表达式语句是由有效表达式组成的语句。块语句简单地说就是代码块（记住，块以{开始，以}结束）。块语句也称为复合语句。

注意： C++增加了两个附加的语句类型：try块（由异常处理使用）和声明语句，这些将在第二部分讨论。

因为许多语句依赖于某些条件测试的结果，所以让我们从复习真值和假值的概念开始。

3.1 C和C++中的真值和假值

许多C/C++语句都依赖于条件表达式来决定下一步执行什么。条件表达式的取值为真或假。在C语言中，真值是任何非0值，包括负数，假值是0。这种处理真假值的方法可以使很多程序得以高效地编码。

C++完全支持刚刚描述的关于真值和假值的0/非0定义，但是C++也定义了一个布尔数据类型bool，它只有真值和假值。正如在第2章中解释的，在C++中，0值被自动转换为假，而非0值自动转换为真。反之亦然：真转换为1，而假转换为0。在C++中，控制条件语句的表达式从技术上讲是bool型的，但是因为任何非0值都转换为真，而任何0值都转换为假，所以在这一点上C和C++没有什么区别。

注意： C99中增加了一个布尔类型_Bool，但是它与C++不兼容。关于在C99的_Bool和C++的bool之间如何兼容的问题，请参阅本书第二部分。

3.2 选择语句

C/C++支持两种类型的选择语句：if和switch。此外，在某些情况下，运算符?是if的一种替换形式。

3.2.1 if语句

if语句的一般形式是：

if (*expression*) *statement*;
else *statement*;

其中，statement可以由单个语句、块语句组成，也可以为空语句，else子句是可选的。

如果expression（表达式）求值为真（除了0以外的任何值），则执行if后的语句或语句块；否则，else后的语句或块执行（如果存在的话）。记住，只执行与if或与else相关联的代码，但不会二者同时执行。

在C语言中，控制if的条件语句必须生成一个标量(scalar)结果。一个标量是一个整数、字符、指针或浮点类型。在C++中，也可以是bool类型。很少使用浮点数来控制条件语句，因为这会大大减慢执行速度（需要用几条指令来执行浮点操作，但执行整数或字符操作时所用的指令相对来说更少一些）。

下列程序包含一个if语句范例，该程序是一个简单的“猜魔数”游戏。当游戏者猜对魔数时，它打印消息** Right **。它使用标准随机数生成器rand()生成魔数，该随机数生成器返回在0到RAND_MAX之间的一个任意数（RAND_MAX定义了32767或一个的更大整数）。rand()要求头文件stdlib.h（C++程序也可以使用新的头文件<cstdlib>）。

```
/* Magic number program #1. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* magic number */

  int guess; /* user's guess */

  magic = rand(); /* generate the magic number */

  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if(guess == magic) printf("** Right **");

  return 0;
}
```

进一步改写魔数程序，下面的程序演示了在游戏者猜错了数时，使用else语句打印出一条消息。

```
/* Magic number program #2. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* magic number */
  int guess; /* user's guess */

  magic = rand(); /* generate the magic number */
```

```
  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if(guess == magic) printf("** Right **");
  else printf("Wrong");
  return 0;
}
```

3.2.2 嵌套的if语句

嵌套的if语句是一条if语句，该if语句是另一条if或else的目标。嵌套的if语句在编程中很常见。在嵌套if语句中，else语句总是与离它最近的if语句相关联，并且这条if语句与else在同一块内且不和别的else相关联。例如，

```
if(i)
{
  if(j) statement 1;
  if(k) statement 2; /* this if */
  else statement 3; /* is associated with this else */
}
else statement 4; /* associated with if(i) */
```

正如注释所说明的，最后一条else语句不与if(j)相关联，因为它们不是在同一块内。最后的else语句与if(i)相关联。还有，里面的else语句与if(k)相关联，它是最近的if语句。

C语言确保至少15层的嵌套。在实际中，大多数编译器允许更多层的嵌套。更重要的是，标准C++建议在C++程序中允许至少256层的嵌套if语句。然而，很少需要层数过多的嵌套，过度嵌套能混淆算法的意义。

通过给游戏者提供猜错情况的反馈信息，可以使用嵌套的if语句来进一步改进魔数程序。

```
/* Magic number program #3. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* magic number */
  int guess; /* user's guess */

  magic = rand(); /* get a random number */

  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if (guess == magic) {
    printf("** Right **");
    printf(" %d is the magic number\n", magic);
  }
  else {
    printf("Wrong, ");
    if(guess > magic) printf("too high\n");
    else printf("too low\n");
  }
```

```
    return 0;
}
```

3.2.3 阶梯型 if–else–if 语句

比较常见的编程结构是阶梯型的 if-else-if 语句，因为其外在的特点，有时称为 if-else-if 阶梯。它的一般形式是：

```
if (expression) statement;
else
 if (expression) statement;
 else
  if (expression) statement;
  .
  .
  .
  else statement;
```

从上往下对条件进行评估。只要发现一个条件为真，就执行与此条件相关的语句且跳过阶梯中其余的部分。如果没有发现条件为真的语句，执行最后一条 else 语句。如果没有 else 语句，而且所有其他条件均为假，则什么也不执行。

虽然从技术上讲，前面 if-else-if 阶梯的缩进格式是正确的，但是可能会导致过度的缩进。因此，通常 if-else-if 阶梯都像下面这样缩进：

```
if (expression)
 statement;
else if (expression)
 statement;
else if (expression)
 statement;
.
.
.
else
 statement;
```

使用 if-else-if 阶梯，魔数程序变为

```
/* Magic number program #4. */
#include <stdio.h>

#include <stdlib.h>

int main(void)
{
  int magic; /* magic number */
  int guess; /* user's guess */

  magic = rand(); /* generate the magic number */
```

```
  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("** Right ** ");
    printf("%d is the magic number", magic);
  }
  else if(guess > magic)
    printf("Wrong, too high");
  else printf("Wrong, too low");

  return 0;
}
```

3.2.4 运算符?的替换

可以使用?运算符来取代一般形式的if-else语句：

if(*condition*) *expression;*
else *expression;*

然而，if和else的目标部分必须是一个表达式，而不是另一条语句。

?运算符称为三元运算符，因为它要求三个操作数。它的一般形式是：

Exp1 ? Exp2 : Exp3

其中，Exp1, Exp2和Exp3是表达式。注意冒号的用法和位置。

?表达式的值是这样决定的：首先对Exp1求值。如果它为真，计算Exp2的值，该值变成了整个?表达式的值。如果Exp1为假，那么计算Exp3，然后它的值变成了这个表达式的值。例如，考虑下面的程序：

```
x = 10;
y = x>9 ? 100 : 200;
```

在这个例子中，y被赋值100。如果x小于9，那么y将取值200。若使用if-else语句，则程序变为：

```
x = 10;
if(x>9) y = 100;
else y = 200;
```

下面的程序使用?运算符来对用户键入的整数值做平方运算，但整数的符号保持不变(10的平方是100，而-10的平方是-100)。

```
#include <stdio.h>

int main(void)
{
  int isqrd, i;

  printf("Enter a number: ");
  scanf("%d", &i);

  isqrd = i>0 ? i*i : -(i*i);
```

```
  printf("%d squared is %d", i, isqrd);

  return 0;
}
```

用?运算符来取代if-else语句并不仅限于赋值的情况。记住，所有的函数（除了用void声明的那些）都返回一个值。因此，在?表达式中可以使用一个或更多的函数调用。遇到这个函数的名称时，这个函数就被执行，以便确定它的返回值。因此，通过把一个或更多的调用放到形成?操作数的表达式中，可以使用?运算符执行一个或更多的函数调用。下面是一个例子。

```
#include <stdio.h>

int f1(int n);
int f2(void);

int main(void)
{
  int t;

  printf("Enter a number: ");
  scanf("%d", &t);

  /* print proper message */
  t ? f1(t) + f2() : printf("zero entered.\n");

  return 0;
}

int f1(int n)
{
  printf("%d ", n);
  return 0;
}

int f2(void)
{
  printf("entered.\n");
  return 0;
}
```

在这个例子中，如果键入了0，就调用printf( )函数并显示消息“键入了0”。如果键入了其他的数，就执行f1( )和f2( )。注意在这个例子中?表达式的值被丢弃了，不需要给它赋任何值。

注意，为了优化目标码，某些C++编译器重新安排表达式的求值顺序。这可能引起构成?运算符操作数的函数以不想要的顺序执行。

使用?运算符，可以再一次重写魔数程序。

```
/* Magic number program #5. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic;
```

```
  int guess;

  magic = rand(); /* generate the magic number */

  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("** Right ** ");
    printf("%d is the magic number", magic);
  }
  else
    guess > magic ? printf("High") : printf("Low");

  return 0;
}
```

这里，?运算符显示了基于 guess > magic 测试结果的适宜的信息。

3.2.5 条件表达式

有时，C/C++的初学者对可以使用任何有效表达式来控制if和?运算符的事实感到困惑，即，你不受限于涉及关系和逻辑运算符的表达式（像在BASIC或Pascal语言中那样）。表达式必须简单地求值为真值或假值（0或非0）。例如，下面的程序从键盘中读取两个整数并显示其商值。它使用一个由第二个数控制的if语句来避免被0除的错误。

```
/* Divide the first number by the second. */

#include <stdio.h>

int main(void)
{
  int a, b;

  printf("Enter two numbers: ");
  scanf("%d%d", &a, &b);

  if(b) printf("%d\n", a/b);
  else printf("Cannot divide by zero.\n");

  return 0;
}
```

这个方法能够工作，因为如果b为0，控制if语句的条件为假，else语句得以执行。否则，条件为真（非0），除法操作得以完成。

要注意的是，按下面这样写if语句

```
if(b != 0) printf("%d\n", a/b);
```

是冗余低效的，并且可看做是不好的编程风格。因为b的值足以控制if语句，所以不需要用0测试它。

3.2.6 switch语句

C/C++具有内嵌的多分支选择语句，称为switch，它根据一个整数或字符常量列表连续地测试一个表达式的值。当发现一个匹配时，与那个常量相关联的语句执行。switch语句的一般

形式是：

```
switch (expression) {
 case constant1:
  statement sequence
  break;
 case constant2:
  statement sequence
  break;
 case constant3:
  statement sequence
  break;
 .
 .
 .
 default
  statement sequence
}
```

expression必须对一个字符或整数值进行计算，例如，浮点表达式就是不允许的。依照case语句中指定的常量值，按顺序测试expression的值。当发现一个匹配时，与那个case相关联的语句序列执行，直到遇到break语句或switch语句的末尾。如果没有发现匹配，则执行default语句。default语句是可选的，如果它不存在，那么若所有的匹配都失败，则不会发生任何动作。

在C语言中，一条switch语句可以有至少257条case语句。标准C++建议支持至少16384条case语句。在实践中，为了效率的原因，你会想要限制case语句的数目到更少的数量。尽管case是一条标签语句，在switch语句外，它不能单独存在。

break语句是C/C++中的一个跳转语句，它既可用在循环语句中，也可用在switch语句中（参见“迭代语句”一节）。如果在switch语句中遇到break语句，程序就“跳转”执行switch语句后面的那行代码。

关于switch语句，有三个要点需要知道：

- switch语句与if语句的不同之处在于：switch语句只能进行相等测试，而if语句可以对任何关系或逻辑表达式求值。
- 在同一个switch语句中，不允许出现两个case常量有相同值的情况。当然，一个switch语句中的case常量，可以和包围这个switch语句的另一个switch语句中的case常量有相同的值。
- 如果在switch语句中使用字符常量，它们将自动转换为整型值。

switch语句经常用于处理键盘命令，如菜单选择命令。下面给出的是一个拼写检查程序，函数menu( )显示了一个菜单，并调用相应的过程：

```
void menu(void)
{
  char ch;

  printf("1. Check Spelling\n");
  printf("2. Correct Spelling Errors\n");
  printf("3. Display Spelling Errors\n");
```

```
  printf("Strike Any Other Key to Skip\n");
  printf("      Enter your choice: ");

  ch = getchar(); /* read the selection from
                     the keyboard */

  switch(ch) {
    case '1':
      check_spelling();
      break;
    case '2':
      correct_errors();
      break;
    case '3':
      display_errors();
      break;
    default :

    printf("No option selected");
  }
}
```

从技术上讲，switch语句中的break语句是可选的，它们终止了与每个常量相关联的语句序列。如果省略了break语句，就继续在下一个case语句中执行，直到遇到break或到达switch语句的末尾。例如，下面的函数使用case的“向下传”的性质来简化设备驱动器输入处理程序的代码：

```
/* Process a value */
void inp_handler(int i)
{
  int flag;

  flag = -1;

  switch(i) {
    case 1: /* These cases have common */
    case 2: /* statement sequences. */
    case 3:
      flag = 0;
      break;
    case 4:
      flag = 1;
    case 5:
      error(flag);
      break;
    default:
      process(i);
  }
}
```

这个例子说明了switch语句的两个方面。首先，case语句中可以没有语句序列。如果是这样，执行简单地跳到下一个case语句。在这个例子中，前三个case语句都执行同样的语句，即

```
flag = 0;
break;
```

第二，如果没有break语句，一个语句序列的执行就会继续到下一个case语句。如果i的值等于4，那么flag设置为1。因为在那条case语句的末尾没有break语句，执行继续并且对error(flag)的调用也被执行。如果i等于5，error(flag)将以flag的值为-1而被调用（不是1）。

如果没有break语句，几个case语句可以同时运行，这一事实防止了不必要的代码重复，提高了代码效率。

3.2.7 嵌套的switch语句

一个switch语句可以作为另一个外层switch语句序列的一部分。即使外层和内层的switch语句中的case常量包含相同的值，也不会发生冲突。例如，下面的代码段是完全合法的：

```
switch(x) {
  case 1:
    switch(y) {
      case 0: printf("Divide by zero error.\n");
              break;
      case 1: process(x,y);
    }
    break;
  case 2:
    .
    .
    .
```

3.3 迭代语句

在C/C++和所有其他高级编程语言中，迭代语句（也称为循环）允许一组指令重复的执行，直到达到一定的条件。这个条件可以被预先定义（像在for循环中），也可以是居后定义的（像在while和do-while循环中）。

3.3.1 for循环

在所有的过程化编程语言中，for循环的设计通常具有不同的形式。然而，在C/C++中，它提供了意想不到的灵活性和高效性。

for语句的一般形式是：

for(*initialization; condition; increment*) *statement;*

for循环有许多变种，但是它的最常用的形式是这样工作的：initialization是赋值语句，用于设置循环控制变量。condition是关系表达式，决定何时退出循环。increment定义每次重复循环时循环变量如何改变。必须用分号把这三个部分分开。只要条件为真，for循环就继续执行。一旦条件为假，程序就在for循环后面的语句处恢复执行。

在下面的程序中，使用for循环来在屏幕上显示从1到100的数字：

```
#include <stdio.h>

int main(void)
{
```

```
  int x;

  for(x=1; x <= 100; x++) printf("%d ", x);

  return 0;
}
```

在这个循环中，x的初值被设为1，然后和100进行比较。因为x小于100，所以调用printf()，之后循环重复。x增1并再次进行测试，看其是否仍然小于等于100。如果是，则调用printf()。重复这个过程，直到x大于100，此时循环终止。在这个例子中，x是循环控制变量，每次循环重复时就改变并检查它。下面的例子是一个迭代多条语句的for循环。

```
for(x=100; x != 65; x -= 5) {
  z = x*x;
  printf("The square of %d, %f", x, z);
}
```

x的平方和对printf()的调用都被执行，直到x等于65。注意这个循环是以降序方式进行的：x被初始化为100，每次循环重复时，从中减5。

在for循环中，条件测试总是在循环顶部执行。这意味着如果开始时条件为假，在循环内部的代码可能根本不会执行。例如，在下面的代码段中：

```
x = 10;
for(y=10; y!=x; ++y) printf("%d", y);
printf("%d", y); /*  this is the only printf()
                     statement that will execute */
```

循环永远不会执行，因为当进入循环时x和y是相等的。这会引起对条件表达式的求值为假，所以循环体和循环的增量部分都不会执行。因此，y的值仍然为0，代码产生的惟一输出是数10在屏幕上显示了一次。

3.3.2 for循环的变种

前面的讨论描述了for循环的最常见的形式。然而，在某些编程环境下，为了增强for循环的性能、灵活性和适用性，也允许for循环的几个变种。

一个最常见的变种使用逗号(,)运算符来允许两个或更多的变量控制循环(记住，必须使用逗号把若干表达式以“do this and this”的形式串到一起。参见第2章)。例如，变量x和y控制下面的循环，并且二者都在for语句内被初始化：

```
for(x=0, y=0; x+y<10; ++x) {
  y = getchar();
  y = y - '0'; /* subtract the ASCII code for 0,
                  from y */
    .
    .
    .
}
```

逗号把两个初始化语句分开。每次循环重复时，x增1，y的值由键盘输入设置。要使循环终止，x和y都必须是正确的值。即使y的值是由键盘输入决定的，y也必须初始化为0，以便它的值在对条件表达式第一次求值时就定义完成(如果没有定义y，有时它可能包含10这个值，

使条件测试为假，循环终止运行）。

下面所示的函数converge( )，演示了多个循环控制变量正在工作。通过把字符从两端移动，向中间会聚，converge( ) 函数把一个字符串的内容复制到另一个字符串中。

```
/* Demonstrate multiple loop control variables. */
#include <stdio.h>
#include <string.h>

void converge(char *targ, char *src);

int main(void)
{

  char target[80] = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXX";

  converge(target, "This is a test of converge().");
  printf("Final string: %s\n", target);

  return 0;
}

/* This function copies one string into another.
   It copies characters to both the ends,
   converging at the middle. */
void converge(char *targ, char *src)
{
  int i, j;

  printf("%s\n", targ);
  for(i=0, j=strlen(src); i<=j; i++, j--) {
    targ[i] = src[i];
    targ[j] = src[j];
    printf("%s\n", targ);
  }
}
```

下面是程序生成的输出。

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
TXXXXXXXXXXXXXXXXXXXXXXXXXXXX
ThXXXXXXXXXXXXXXXXXXXXXXXXXXX.
ThiXXXXXXXXXXXXXXXXXXXXXXXXXX).
ThisXXXXXXXXXXXXXXXXXXXXXXXX().
This XXXXXXXXXXXXXXXXXXXXe().
This iXXXXXXXXXXXXXXXXXXge().
This isXXXXXXXXXXXXXXXXrge().
This is XXXXXXXXXXXXXerge().
This is aXXXXXXXXXXXXverge().
This is a XXXXXXXXXXnverge().
This is a tXXXXXXXXonverge().
This is a teXXXXXXconverge().
This is a tesXXXX converge().
This is a testXXf converge().
This is a test of converge().
Final string: This is a test of converge().
```

在converge( )函数中，for循环使用两个循环控制变量i和j，从相反的两端给字符串加下标。当循环迭代时，i增加而j减小。当i比j大时，循环终止，从而保证复制了所有的字符。

条件表达式不必涉及根据一些目标值来测试循环控制变量的工作。事实上，条件可以是任何关系或逻辑语句，这意味着可以测试几个可能的终止条件。例如，可以使用下面的函数把用户登录到远程系统。用户有三次机会键入密码。当用完三次或用户键入了正确的密码时，循环终止。

```
void sign_on(void)
{
  char str[20] = "";
  int x;

  for(x=0; x<3 && strcmp(str, "password"); ++x) {
    printf("Enter password please:");
    gets(str);
  }

  if(x==3) return;
  /* else log user in ... */
}
```

这个函数使用strcmp( ),它是一个标准库函数，用于比较两个字符串，如果匹配，则返回0。

记住，for循环三个部分中的每一个都可以由任何有效的表达式组成，实际上表达式不必与通常所用的部分相关联。考虑下面的例子：

```
#include <stdio.h>

int sqrnum(int num);
int readnum(void);
int prompt(void);

int main(void)
{
  int t;

  for(prompt(); t=readnum(); prompt())
    sqrnum(t);

  return 0;
}

int prompt(void)
{
  printf("Enter a number: ");
  return 0;
}

int readnum(void)
{
  int t;

  scanf("%d", &t);
  return t;
```

```
}

int sqrnum(int num)
{
  printf("%d\n", num*num);
  return num*num;
}
```

仔细看一下 main( )中的for循环。注意for循环的每一部分都是由函数调用组成的，函数调用提示用户并读取从键盘键入的一个数字。如果键入的数是0，循环终止，因为条件表达式将为假。否则，平方此数。因此，这个 for 循环在非传统但是完全有效的意义上使用初始化和增量部分。

for循环的另一个有趣的品质是，循环定义块的三个部分是可选的。事实上，其中的任何部分都不必一定是表达式，因为表达式是可选的。例如，下面这个循环将会运行，直到用户键入了 123：

```
for(x=0; x!=123; ) scanf("%d", &x);
```

注意 for 循环定义的增量部分是空的。这意味着每次循环重复时，测试 x 看看它是否等于123，但是没有进一步的操作。然而，如果在键盘上键入123，循环条件将变为假并且循环终止。

循环控制变量的初始化可以出现在 for 语句的外面。当循环控制变量的初始条件必须用某些复杂的方法进行计算时，常发生这种情况，如下例所示：

```
gets(s); /* read a string into s */
if(*s) x = strlen(s); /* get the string's length */
else x = 10;

for( ; x<10; ) {
  printf("%d", x);
  ++x;
}
```

初始部分为空，且 x 在进入循环前已被初始化。

3.3.3 无限循环

尽管可以使用任何循环语句创建一个无限循环，for循环通常用于这个目的。因为不要求形成 for 循环的三个表达式的任何一个，所以可以通过使条件表达式为空来创建一个无限循环：

```
for( ; ; ) printf("This loop will run forever.\n");
```

当没有条件表达式时，我们假定它为真。虽然可以有一个初始化和增量表达式，但是C++程序员更常使用 for(;;) 结构来表示一个无限循环。

实际上，for(;;) 结构并不能保证一个无限循环，因为在循环体内遇到的break语句会立即终止循环（break 将在本章后面详细讨论）。程序控制在循环后面的代码行处恢复，如下面所示：

```
ch = '\0';

for( ; ; ) {
  ch = getchar(); /* get a character */
  if(ch=='A') break; /* exit the loop */
```

```
}
printf("you typed an A");
```

这个循环会一直运行，直到用户在键盘上键入了A为止。

3.3.4 没有循环体的for循环

一条语句可以是空语句，这意味着for循环体（或者任何其他循环体）也可以为空。可以使用这个事实来改进某些算法的效率并创建时间延迟循环。

从输入流中除去空格是一项常见的任务。例如，一个数据库程序可以允许这样的查询："显示小于400的所有收支平衡情况"。这个数据库要求输入每个字，每个字前面没有空格。即，数据库输入处理器识别"show"而不是" show"。下面的循环显示了完成此工作的一种方法，它去掉了由str所指向的字符串中的前导空格。

```
for( ; *str == ' '; str++) ;
```

可以看到，这个循环没有循环体——也不需要。

时间延迟（Time delay）循环经常用在程序中。下面的代码显示了如何使用for循环创建一个时间延迟循环：

```
for(t=0; t<SOME_VALUE; t++) ;
```

3.3.5 while循环

C/C++中第二个循环语句是while循环。它的一般形式是：

while(*condition*) statement;

其中statement可以是一条空语句、一条语句或语句块。condition可以是任何表达式，真值是任何非0值。当条件为真时，循环重复。当条件为假时，程序控制传递到循环后面第一个代码行处。

下面的例子显示了一个键盘输入例程，这个例程简单地循环，直到用户键入了A为止：

```
char wait_for_char(void)
{
  char ch;

  ch = '\0'; /* initialize ch */
  while(ch != 'A') ch = getchar();
  return ch;
}
```

首先，ch初始化为空值。作为一个局部变量，它的值在执行wait_for_char( )时是未知的。然后while循环检查看ch是否不等于A。

因为ch被初始化为空，测试为真，循环开始。每次按下一个键时，就再次测试条件。一旦键入了A，条件为假，因为ch等于A，此时循环终止。

像for循环一样，while循环在循环顶部检查测试条件，这意味着如果条件在开始时就为假，循环体不会执行。这个特征可以消除在循环之前执行分离的条件测试的需要。pad( )函数很好地演示了这一点，它在字符串的末尾处添加空格，以把字符串填充到预先定义的长度。如果字符

串已经是要求的长度，就不添加空格。

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

void pad(char *s, int length);

int main(void)
{
  char str[80];

  strcpy(str, "this is a test");
  pad(str, 40);
  printf("%d", strlen(str));

  return 0;
}

/* Add spaces to the end of a string. */
void pad(char *s, int length)
{
  int l;

  l = strlen(s); /* find out how long it is */

  while(l<length) {
    s[l] = ' '; /* insert a space */
    l++;
  }
  s[l] = '\0'; /* strings need to be
                  terminated in a null */
}
```

pad( )有两个参数，一个是s，指向要延长字符串的一个指针，另一个是length，表示s应该有的字符数。如果字符串长度s已经等于或大于length，那么在while循环内的代码就不会执行。如果s小于length，pad( )添加要求数量的空格。strlen( )函数（标准库的一部分）返回字符串的长度。

如果需要几个不同的条件来终止while循环，那么通常仅用一个变量作为条件表达式，这个变量的值可在循环的各个点处设置。在下面这个例子中，

```
void func1(void)
{
  int working;

  working = 1; /* i.e., true */

  while(working) {
    working = process1();
    if(working)
      working = process2();
    if(working)
      working = process3();
  }
}
```

三个例程中的任何一个都可以返回假，并导致循环退出。

在while循环体内不需要任何语句。例如，

```
while((ch=getchar()) != 'A') ;
```

将简单地循环，直到用户键入了A为止。如果不愿意把赋值放到while条件表达式内，记住等号仅是一个运算符，用于计算右边操作数的值。

3.3.6 do-while循环

不像for和while循环（它们是在循环的顶部测试循环条件），do-while循环在循环的底部检查它的条件，这意味着do-while循环总是至少执行一次。do-while循环的一般形式是：

```
do {
 statement;
} while(condition);
```

尽管在只有一条语句时不需要花括号，它们常常被用来避免和while相混淆（对你，而不是对编译器来说）。do-while循环重复，直到条件为假。

下面的do-while循环将读取来自键盘的数字，直到它找到一个小于或等于100的数为止。

```
do {
  scanf("%d", &num);
} while(num > 100);
```

或许do-while循环最常用的情况是在菜单选择函数中。当用户键入一个有效的响应时，它作为函数值返回。无效响应会引发一个重新提示。下面的代码显示了本章前面开发的一个拼写检查程序的改进版本：

```
void menu(void)
{
  char ch;

  printf("1. Check Spelling\n");
  printf("2. Correct Spelling Errors\n");
  printf("3. Display Spelling Errors\n");
  printf("      Enter your choice: ");

  do {
    ch = getchar(); /* read the selection from
                       the keyboard */
    switch(ch) {
      case '1':
        check_spelling();
        break;
      case '2':
        correct_errors();
        break;
      case '3':
        display_errors();
        break;
    }
  } while(ch!='1' && ch!='2' && ch!='3');
```

```
}
```

这里，do-while循环是一个很好的选择，因为你将总会想要一个菜单函数显示此菜单至少一次。在显示了选项后，程序进入循环，直到选择了有效的选项。

3.4 在选择和迭代语句内声明变量

在C++（不是C89）中，在if或switch语句的条件表达式内、在while循环的条件表达式内、或在for循环的初始化部分声明一个变量是可能的。在这些地方声明的变量其作用域限定到由那条语句控制的代码块内。例如，在for循环内声明的变量将限定在那个循环内。

下面是一个在for循环的初始化部分声明一个变量的例子：

```
/* i is local to for loop; j is known outside loop. */
int j;
for(int i = 0; i<10; i++)
  j = i * i;

/* i = 10; // *** Error *** -- i not known here! */
```

这里，i在for循环的初始化部分内声明并用来控制循环。在循环外面，i是未知的。

因为在for循环中的循环控制变量经常仅被那个循环所需要，所以在for循环的初始化部分声明变量就变成了一个常见的做法。然而，要记住，C89不支持这种做法（C99去除了这种限制）。

> 提示：一个在for循环的初始化部分声明的变量是否是局限于那个循环的，这一点会随着时间的推移而改变。最初，这个变量在for循环后是可用的。然而，标准C++限制这个变量到for循环的范围，正如刚才所描述的。

如果编译器完全用标准C++进行编译，那么也可以在任何条件表达式内声明一个变量，如被if或while循环所用的那些。例如，下面这个代码段，

```
if(int x = 20) {
  x = x - y;

  if(x>10) y = 0;
}
```

声明x并给它赋值20。因为这是一个真值，所以执行if语句的目标。在条件语句内声明的变量其作用域限定到由那个语句控制的代码块中。因此，在这个例子中，在if外面，x是未知的。坦率地讲，并不是所有的程序员都相信在条件语句内声明变量是好的编程实践，本书中将不使用这种技术。

3.5 跳转语句

C/C++有四种执行无条件分支的语句：return, goto, break和continue。当然，可以在程序的任何地方使用return和goto语句。可以与任何循环语句一起使用break和continue语句。正如本章前面讨论的，也可以在switch语句中使用break语句。

3.5.1 return 语句

使用 return 语句来从一个函数中返回。可把 return 语句看做跳转语句，因为它使得执行返回（跳转回）到调用这个函数的地方。return 可以有、也可以没有与它关联的值。如果它有一个与它关联的值，那个值就变为这个函数的返回值。在 C89 中，从技术上讲，非 void 函数不必返回一个值。如果没有指定返回值，将返回一个无用值。然而，在 C++（和 C99）中，一个非 void 函数必须返回一个值。即，在 C++ 中，如果函数被指定为返回一个值，在其中的任何 return 语句都必须有一个与它相关联的值（甚至在 C89 中，如果函数声明为返回一个值，好的编程实践是确实返回一个值）。

return 语句的一般形式是：

return *expression*;

仅在函数声明为返回一个值时，expression 才会出现。在这种情况下，expression 的值将变成函数的返回值。

在函数内，只要喜欢，可以使用多条 return 语句。然而，只要遇到第一条 return 语句，函数就停止运行。结束函数的大括号}也会引起函数返回，它与没有任何指定值的 return 语句一样。如果这出现在非 void 函数中，那么函数的返回值是未定义的。

声明为 void 的函数可能不包含指定一个值的 return 语句。因为 void 函数没有返回值，在 void 函数内没有 return 语句能够返回一个值就很有意义了。关于 return 语句的更多信息，请参见第 6 章。

3.5.2 goto 语句

因为 C/C++ 有丰富的控制结构并允许使用 break 和 continue 进行附加的控制，所以就不需要 goto 语句了。关于 goto 语句，大多数程序员关心的是：它倾向于使程序不可读。然而，尽管 goto 语句在几年前就不受人欢迎了，它有时有自己的用途。不存在要求 goto 语句的编程环境，它仅是为了方便的缘故。如果广泛使用，在某些编程环境中可能有利，比如跳出深层嵌套的循环。在本节以外，不使用 goto 语句。

goto 语句在进行操作时要求一个标签，标签（label）是后跟一个冒号的有效标识符。还有，标签必须和使用它的 goto 语句在同一函数中——不能在函数间跳转。goto 语句的一般形式是：

goto *label*;
.
.
.
label:

其中，label 是在 goto 之前或之后的任何有效的标签。例如，可以使用 goto 和一个标签，创建一个从 1 到 100 的循环，如下所示：

```
x = 1;
loop1:
  x++;
  if(x<100) goto loop1;
```

3.5.3 break语句

break语句有两种用途。可以使用它来终止switch语句(在本章前面关于switch语句的一节中讨论过)中的case语句,也可以使用它来强迫立即退出一个循环,跳过正常的循环条件测试。

当在循环中遇到break语句时,循环立即终止,程序控制在循环后的下一条语句处恢复。例如,

```
#include <stdio.h>

int main(void)
{
  int t;

  for(t=0; t<100; t++) {
    printf("%d ", t);
    if(t==10) break;
  }

  return 0;
}
```

上面的程序在屏幕上显示从0到10的数字。然后,循环终止,因为break语句导致立即退出循环,而没有考虑条件测试t<100。

程序员经常在循环中使用break语句,在循环中一个特定条件可能引起循环的立即终止。例如,在下面的例子中,一个按键可以停止look_up( )函数的执行:

```
void look_up(char *name)
{
  do {
    /* look up names ... */
    if(kbhit()) break;
  } while(!found);
  /* process match */
}
```

如果没有按键,kbhit( )函数返回0;否则,它返回一个非0值。因为计算机环境间的区别,标准C和标准C++都没有定义kbhit( ),但是编译器一定会提供它(其名称可能不同)。一条break语句只能退出最里面的循环,例如,

```
for(t=0; t<100; ++t) {
  count = 1;
  for(;;) {
    printf("%d ", count);
    count++;
    if(count==10) break;
  }
}
```

在屏幕上显示从1到10的数字100次。每次执行时遇到break语句,控制返回到外面的for循环。

在switch语句中使用的break语句仅会影响那个switch语句,它对switch碰巧所属的循环没有影响。

3.5.4 exit( )函数

尽管exit( )不是程序控制语句，但是在这里讨论它还是很有必要的。正如用break可以跳出循环一样，使用标准库函数exit( )也可以退出程序。这个函数可以使整个程序终止运行，强迫返回到操作系统。事实上，exit( )的作用相当于它正在从整个程序中退出。

exit( )函数的一般形式是：

```
void exit(int return_code);
```

return_code的值被返回到调用进程，通常为操作系统。习惯上，返回0表示正常的程序终止，返回其他参数则表示某种错误。也可以使用宏EXIT_SUCCESS和EXIT_FAILURE作为返回码。exit( )函数要求头文件stdlib.h。C++程序也可以使用C++风格的头文件<cstdlib>。

当程序执行的强制条件不满足要求时，程序员经常使用exit( )。例如，想像一个虚拟的计算机游戏，它要求一个专用的图形适配卡。这个游戏的main( )函数像下面这样：

```
#include <stdlib.h>

int main(void)
{
  if(!virtual_graphics()) exit(1);
  play();
  /* ... */
}
/* .... */
```

其中virtual_graphics( )是用户定义的函数，如果虚拟图形适配卡存在，则此函数返回真。如果不存在，则此函数返回假且程序终止。

下面的程序使用exit( )退出程序并返回到操作系统：

```
void menu(void)
{
  char ch;

  printf("1. Check Spelling\n");
  printf("2. Correct Spelling Errors\n");
  printf("3. Display Spelling Errors\n");
  printf("4. Quit\n");
  printf("     Enter your choice: ");

  do {
    ch = getchar(); /* read the selection from
                       the keyboard */
    switch(ch) {
      case '1':
        check_spelling();
        break;
      case '2':
        correct_errors();
        break;
      case '3':
        display_errors();
```

```
        break;
      case '4':
        exit(0); /* return to OS */
    }
  } while(ch!='1' && ch!='2' && ch!='3');
}
```

3.5.5 continue语句

continue语句有点像break语句。然而，代替强迫程序终止，continue跳过循环结束前的语句，强迫循环的下一个迭代发生。在for循环语句中遇到continue后，首先进行条件测试，然后执行循环的增量部分。在while和do-while循环中遇到continue后，程序控制直接回到条件测试部分。例如，下面的程序可以计算用户键入的字符串中的空格数：

```
/* Count spaces */
#include <stdio.h>

int main(void)
{
  char s[80], *str;
  int space;

  printf("Enter a string: ");
  gets(s);
  str = s;

  for(space=0; *str; str++) {
    if(*str != ' ') continue;
    space++;
  }
  printf("%d spaces\n", space);

  return 0;
}
```

对每个字符进行测试，看它是否是一个空格。如果不是，continue语句强迫for循环重新开始。如果是一个空格，则space加1。

下面的程序显示怎样使用continue语句，通过强迫尽快执行条件测试来加速退出循环：

```
void code(void)
{
  char done, ch;

  done = 0;
  while(!done) {
    ch = getchar();
    if(ch=='$') {
      done = 1;
      continue;
    }
    putchar(ch+1); /* shift the alphabet one
                      position higher */
  }
}
```

这个函数通过把你键入的所有字符移向下一个字符来完成编码信息。例如，A变成了B。这个函数在你键入$时终止。在输入$后，不会出现进一步的输出，因为由于continue的作用，条件测试时会发现done为真，从而引起循环退出。

3.6 表达式语句

第2章详细讨论了表达式，然而，这里要提及一些特殊的要点。记住，表达式语句只是一个有效的表达式后跟一个分号，如下所示：

```
func();  /* a function call */
a = b+c; /* an assignment statement */
b+f();   /* a valid, but strange statement */
;        /* an empty statement */
```

第一个表达式语句执行函数调用。第二个是一个赋值语句。第三个表达式，尽管很奇怪，仍然由C++编译器求值并且调用了函数f()。最后一个例子显示一条语句可以为空（有时称为空语句）。

3.7 块语句

块语句是一组相关的语句，被作为一个单元对待。组成块的语句在逻辑上绑定在一起。块语句也称为复合语句，一个块语句以{开始，以}结束。程序员经常使用块语句来创建其他语句，如if语句的多语句目标。然而，可以把块语句放到任何地方，就像放置其他语句一样。例如，下面是完全合法（尽管有点不寻常）的C/C++代码：

```
#include <stdio.h>

int main(void)
{
  int i;

  { /* a block statement */
    i = 120;
    printf("%d", i);
  }

  return 0;
}
```