第10章　预处理器和注释

C或C++程序的源代码中可以包括各种编译指令，这些指令称为预处理命令。虽然它们实际上不是C/C++语言的一部分，但却扩展了程序设计环境的范围。本章也将介绍注释。

10.1　预处理器

在开始讨论之前，让我们先从历史角度看看预处理器。虽然预处理器与C++有关，但它主要来源于C语言。此外，C++预处理器实际上与C语言定义的预处理器一样。在这方面C和C++之间的主要区别是它们依赖预处理器的程度。在C语言中，每种预处理器命令是必需的。在C++中，由于更新的、更好的C++语言元素的出现，使某些特征变为多余的了。事实上，C++的一个长期设计目标是完全消除预处理器。但是在现在及可预见的未来，还将广泛使用预处理器。

预处理器包括下列指令：

#define	#elif	#else	#endif
#error	#if	#ifdef	#ifndef
#include	#line	#pragma	#undef

可以看到，所有预处理指令都以符号#开头。此外，每个预处理指令必须单独占用一行。例如

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

是无效的。

10.2　#define

指令#define定义了一个标识符及一个字符序列（即字符集合）。在源程序中每次遇到该标识符时，就用定义的字符序列替换它。标识符被称为宏名，替换过程称为宏替换，指令的一般形式为：

#define *macro-name char-sequence*

注意，该语句没有分号。在标识符和字符序列之间可以有任意个空格，但是一旦字符序列开始，就只能由一新行结束。

例如，如希望用字LEFT代表1，字RIGHT代表0，可以声明这两个#define指令：

```
#define LEFT 1
#define RIGHT 0
```

这使得在源程序中每次遇到LEFT或RIGHT时就用1或0代替。例如，下例在屏幕上显示012。

```
printf("%d %d %d", RIGHT, LEFT, LEFT+1);
```

宏名一旦定义，即可成为其他宏名定义的一部分。例如，下面的代码定义了 ONE，TWO 和 THREE 的值。

```
#define ONE     1
#define TWO     ONE+ONE
#define THREE   ONE+TWO
```

宏替换仅仅是以与之关联的字符序列代替标识符。因此，如果希望定义一个标准错误信息，可编写如下代码：

```
#define E_MS "standard error on input\n"
/* ... */
printf(E_MS);
```

编译器遇到标识符 E_MS 时，就用字符串“standard error on input\n”替换。对于编译器，printf( )语句的实际形式如下：

```
printf("standard error on input\n");
```

如果在字符串中含有标识符，则不进行替换。例如：

```
#define XYZ this is a test

printf("XYZ");
```

不打印“this is a test”，而是“XYZ”。

如果字符序列超过一行，可以在该行末尾用一个反斜杠续行。例如：

```
#define LONG_STRING "this is a very long \
string that is used as an example"
```

C/C++程序员普遍使用大写字母定义标识符。这种约定使程序中的宏替换一目了然。最好是将所有的 #define 放到文件的开始处或独立的文件中，而不是将它们分散到整个程序中。

宏替换的最一般的用途是定义程序中出现的“常量”的名字。例如，某一程序定义了一个数组，而它的几个子程序要访问数组，此时，不应直接以常量定义数组大小，最好是用#define 语句定义它。在需要用数组大小的地方使用宏名。这样在需要改变数组大小时，只需在改变 #define 语句后，再重新编译程序即可。例如：

```
#define MAX_SIZE 100
/* ... */
float balance[MAX_SIZE];
/* ... */
for(i=0; i<MAX_SIZE; i++) printf("%f", balance[i]);
/* ... */
for(i=0; i<MAX_SIZE; i--) x =+ balance[i];
```

因为 MAX_SIZE 定义了 balance 数组的大小，因此，如果将来 balance 的大小需要改变，则只需改变MAX_SIZE的定义即可。在编译程序时，所有引用MAX_SIZE的地方均自动得到修改。

注意：C++ 提供了一个定义常量的更好的方法，即使用 const 关键字，我们将在本书第二部分描述。

10.2.1 定义类似函数的宏

#define命令的另一个有用特性是：宏名可以有变元。每次遇到宏名时，其定义中所用的变元均由程序中的实际变元来代替，这种形式的宏称为类似函数的宏。例如：

```
#include <stdio.h>

#define ABS(a) (a)<0 ? -(a) : (a)

int main(void)
{
  printf("abs of -1 and 1: %d %d", ABS(-1), ABS(1));

  return 0;
}
```

当编译该程序时，宏定义中的a将被-1和1取代，包含a的括号确保了每种情形下的正确替换。例如，如果围绕a的括号被去掉，表达式

```
ABS(10-20)
```

在宏替换后将被转换成：

```
10-20<0 ? -10-20 : 10-20
```

显然会产生错误的结果。

用类似函数的宏替换函数的一大好处是：宏替换提高了代码的执行速度，因为不存在函数调用开销。然而，如果类似函数的宏很大，则这种速度的提高就会由于重复代码而导致了程序长度的增加。

注意： 尽管带参数的宏很有价值，然而C++中有一种创建内联代码的更好的方法，它用inline关键字。

10.3 #error

#error指令强迫编译器停止编译，它主要用于程序调试。#error指令的一般形式为：

#error error-message

error-message不用双引号括起来。当遇到#error命令时，错误信息随着编译的其他信息一同显示。

10.4 #include

指令#include告诉编译器除了读包含#include的一个源文件外还要读取另一个源文件，被读入的源文件必须用双引号或尖括号括起来，例如：

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

这两行代码指示编译器读入并编译用于处理C I/O文件库函数的头文件。

“包含文件”中可以嵌入#include指令，这种方式称为嵌套包含。嵌套层次依赖于具体实现。标准C规定至少可以得到8层嵌套包含。标准C++推荐至少支持256层嵌套。

文件名是用引号还是用尖括号括起来决定了指定文件的查找方法。如果文件名用尖括号括起来，则文件以编译器的创建者定义的方式进行查找，通常，这意味着从为“包含文件”设置的一些特殊目录中查找。如果文件名用引号括起来，则文件以其他实现所定义的方式进行查找。对许多编译器来说，这意味着查找当前工作目录。如果未找到，则按文件名括在尖括号中的方式继续查找。

通常，许多程序员使用尖括号来包含标准头文件，引号的使用则是为了包含与程序关联的文件，但这两种用法没有硬性的规定。

除了文件外，C++程序可以使用#include指令来包含一个C++头文件。C++定义了一组标准头文件，这些头文件提供了各种C++库所需的信息。一个头文件是一个标准标识符，可以但不必映射到一个文件名。因此，一个头文件只是一个抽象，该抽象保证包含了程序所要求的合适的信息。与头文件有关的各种问题在第二部分讨论。

10.5　条件编译指令

有几个指令允许你对程序源代码的各部分有选择地进行编译，这个过程称为条件编译。商业软件公司广泛应用条件编译来提供和维护某一程序的许多定制版本。

10.5.1　#if，#else，#elif 和 #endif

通常，最常用的条件编译指令为#if，#else，#elif 和 #endif。这些指令允许你根据常量表达式的结果，有条件地选择代码指令。

#if 指令的一般形式为：

```
#if constant-expression
 statement sequence
#endif
```

如果#if后面的常量表达式为true，则编译它与#endif之间的代码，否则跳过这些代码。指令 #endif 标识一个 #if 块的结束。例如：

```
/* Simple #if example. */
#include <stdio.h>

#define MAX 100

int main(void)
{
#if MAX>99
  printf("Compiled for array greater than 99.\n");
#endif

  return 0;
}
```

由于MAX大于99，所以程序在屏幕上显示一条消息。该例说明了一个重点：跟在#if后面的表达式在编译时求值，因此，它必须仅包含常量及已定义过的标识符，不能使用变量。

#else 的工作方式与C语言中的else相同：当#if失败时，#else建立起另一选择。因而，上面的例子可扩充为：

```
/* Simple #if/#else example. */
#include <stdio.h>
#define MAX 10

int main(void)
{
#if MAX>99
  printf("Compiled for array greater than 99.\n");
#else
  printf("Compiled for small array.\n");
#endif

  return 0;
}
```

在此例中，因为MAX小于99，不编译#if块，而编译#else块。因此，屏幕上显示“Compiled for small array”这一消息。

注意，#else被使用来标志#if块的末尾和#else块的开始。这是必需的，因为任何#if仅有一个#endif与之关联。

#elif意指“else if”，它形成一个if-else-if嵌套语句，用于多种编译选择。#elif后面跟一个常量表达式。如果表达式为true，则编译其后的代码块，不对其他#elif表达式进行测试，否则，顺序测试下一块。#elif的一般形式为：

```
#if expression
  statement sequence
#elif expression 1
  statement sequence
#elif expression 2
  statement sequence
#elif expression 3
  statement sequence
#elif expression 4
.
.
.
#elif expression N
  statement sequence
#endif
```

例如，下面的程序利用ACTIVE_COUNTRY的值定义货币符号。

```
#define US 0
#define ENGLAND 1
#define JAPAN 2

#define ACTIVE_COUNTRY US

#if ACTIVE_COUNTRY == US
  char currency[] = "dollar";
#elif ACTIVE_COUNTRY == ENGLAND
  char currency[] = "pound";
#else
```

```
  char currency[] = "yen";
#endif
```

标准C指定#if及#elif可以有8层嵌套，标准C++建议至少支持256层嵌套。嵌套时，每个#endif，#else及#elif与最近的#if或#elif配对，例如，下面的程序是完全有效的：

```
#if MAX>100
  #if SERIAL_VERSION
    int port=198;
  #elif
    int port=200;
  #endif
#else
  char out_buffer[100];
#endif
```

10.5.2　#ifdef和#ifndef

条件编译的另一种方法使用#ifdef和#ifndef指令，它们分别表示“如果有定义”和“如果无定义”。#ifdef的一般形式为：

```
#ifdef macro-name
statement sequence
#endif
```

如果宏名在前面#define语句中已定义过，则该语句后面的代码块被编译。

#ifndef的一般形式为：

```
#ifndef macro-name
 statement sequence
#endif
```

如果宏名没有用#define语句定义，则编译该代码块。

#ifdef与#ifndef都可使用#else或#elif语句。例如：

```
#include <stdio.h>

#define TED 10

int main(void)
{
#ifdef TED
  printf("Hi Ted\n");
#else
  printf("Hi anyone\n");
#endif
#ifndef RALPH
  printf("RALPH not defined\n");
#endif

  return 0;
}
```

上述代码打印“Hi Ted”和“RALPH not defined”。如果TED没有定义，则显示“Hi anyone”，

后面是“RALPH not defined”。

在标准C中，可以嵌套#ifdef与#ifndef到至少8层，标准C++支持至少256层嵌套。

10.6 #undef

#undef指令取消其后那个前面已定义过的宏名，也就是不定义宏。#undef的一般形式为：

#undef *macro-name*

例如：

```
#define LEN 100
#define WIDTH 100

char array[ LEN][ WIDTH];

#undef LEN
#undef WIDTH
/* at this point both LEN and WIDTH are undefined */
```

直到遇到#undef语句之前，LEN与WIDTH均有定义。

#undef的重要目的是将宏名局限于仅需要它们的代码段中。

10.7 使用defined

除了#ifdef外，还有一种方法可以确定宏名是否已定义。可以将#if指令和defined编译时运算符一起使用。defined运算符的一般形式如下：

defined *macro-name*

如果macro-name已被定义，则表达式为真，否则为假。例如，要确认宏MYFILE是否已被定义，可以用下面两种预处理命令之一：

```
#if defined MYFILE
```

或

```
#ifdef MYFILE
```

可以在defined前面加上!来颠倒条件。例如，仅当DEBUG没有定义时，才编译以下程序段：

```
#if !defined DEBUG
  printf("Final version!\n");
#endif
```

使用defined语句的一个原因是允许由#elif语句确认宏名的存在。

10.8 #line

#line指令改变__LINE__和__FILE__的内容，它们是在编译器中预先定义的标识符。__LINE__标识符包含已编译的代码的行数。__FILE__标识符是一字符串，包含已编译的源

文件名。#line 的一般形式是：

#line *number "filename"*

其中，number 为任何正整数，是__LINE__ 的新值。filename 为任意有效的文件标识符，是__FILE__ 的新值。#line 主要用于调试及其他特殊应用。

例如，下面的代码规定程序中行计数从 100 行开始，printf( )语句显示数 102，因为它是语句 #line 100 后的第 3 行。

```
#include <stdio.h>

#line 100                       /* reset the line counter */
int main(void)                  /* line 100 */
{                               /* line 101 */
  printf("%d\n",__LINE__);      /* line 102 */

  return 0;
}
```

10.9　#pragma

#pragma 指令是一个实现时定义的指令，它允许向编译器传送各种指令。例如，编译器可能有一种选择：支持对程序执行的跟踪。可用 #pragma 语句指定一个跟踪选项。编译器的用户手册中有更详细的说明。

10.10　# 和 ## 预处理器运算符

有两种预处理器运算符：# 和 ##。这些运算符与 #define 语句一起使用。

运算符，通常称为 stringize 运算符，使得在它后面的变元转换成带引号的串。例如，考虑下面的程序：

```
#include <stdio.h>

#define mkstr(s) # s

int main(void)
{
  printf(mkstr(I like C++));

  return 0;
}
```

预处理器把下一行

```
printf(mkstr(I like C++));
```

转换成：

```
printf("I like C++");
```

操作符，称为 pasting 运算符，用于连接两个符号。例如：

```
#include <stdio.h>

#define concat(a, b) a ## b
```

```
int main(void)
{
  int xy = 10;

  printf("%d", concat(x, y));

  return 0;
}
```

预处理器把下一行

```
printf("%d", concat(x, y));
```

转换成：

```
printf("%d", xy);
```

如果这些运算符对你来说很陌生，请记住，大多数程序并不需要或不常用这些运算符。它们之所以存在，主要是允许预处理器去处理一些特殊的情况。

10.11 预定义的宏名

C++规定了六个内嵌的预定义的宏名，它们是：

```
__LINE__
__FILE__
__DATE__
__TIME__
__STDC__
__cplusplus
```

C语言定义了前五个，下面将依次讨论它们。

__LINE__和__FILE__宏指令在本章有关#line的部分中已讨论过。简单地讲，编译时，它们包含程序的当前行数和文件名。

__DATE__宏指令含有形式为月/日/年的字符串，表示源文件被翻译为目标码时的日期。

__TIME__宏指令包含程序编译的时间。时间用字符串表示，其串的形式为时：分：秒。

__STDC__宏的意义是编译时定义的。一般来讲，如果__STDC__已定义，编译器将仅接受不包含任何非标准扩展的标准C/C++代码。

与标准C++一致的编译器将把__cplusplus定义为一个包含至少6位的数值。与标准C++不一致的编译器将使用具有5位或更少位的数值。

10.12 注释

C89只定义了一种注释，这种注释以字符/*开始，以*/结束。在星号与斜杠之间不允许有空格。编译器忽略注释开始符到注释结束符之间的任何文本。例如，下列程序在屏幕上只打印“hello”。

```
#include <stdio.h>

int main(void)
```

```
{
  printf("hello");
  /* printf("there"); */

  return 0;
}
```

这种注释通常称为多行注释，因为注释文本散布于两行或更多行上。例如：

```
/* this is a
multi-line
comment */
```

注释可出现在程序的任意位置，但它不能出现在关键字或标识符中。例如以下的注释：

```
x = 10+ /* add the numbers */5;
```

是有效的。而

```
swi/*this will not work*/tch(c) {  ...
```

是无效的，因为关键字中不能含有注释，然而，必须认识到，在表达式中插入注释会使程序含混不清。

注释不可嵌套，即一个注释不能含有另一个注释。例如，该代码在编译时出错。

```
/* this is an outer comment
  x = y/a;
  /* this is an inner comment - and causes an error */
*/
```

10.12.1　单行注释

C++ (和 C99)支持两种类型的注释。第一种是多行注释，第二种是单行注释。单行注释由 // 开始，在行结束时结束。例如，

```
// this is a single-line comment
```

当需要短的、一行一行的描述时，单行注释是特别有用的。尽管从技术上讲 C89 不支持单行注释，但是许多编译器接受它们，C99 把单行注释添加到了 C 语言中。还有，单行注释可以嵌套在多行注释内。

当需要解释程序时就利用注释。除了最简单和最直观的函数外，都应有注释，在函数开始处说明其功能、如何调用以及返回何处。