第5章

任务自动化——预处理

C++ 程序编译的处理过程一般分为预处理、编译(及汇编)和连接,如图 5.1 所示。预处理(preprocess)是在程序编译之前,由预处理器(preprocessor)对源程序中各种预处理命令进行先行处理,处理完毕自动进入对源程序的编译。编译时先分析、后综合,分析是指编译器对源程序进行词法分析和语法分析;综合是指代码优化、存储分配和代码生成。为了完成这些分析综合任务,编译器采用对源程序进行多次扫描的办法,每次扫描集中完成一项或几项任务,也有一项任务分散到几次扫描中完成的。大多数的编译器直接产生机器语言的目标代码,但有的编译器则先产生汇编语言的符号代码,然后调用汇编器(assembler)进行翻译加工处理,产生目标代码。连接器将在不同文件中的目标代码和库函数代码经过重定位处理连接成可执行文件。

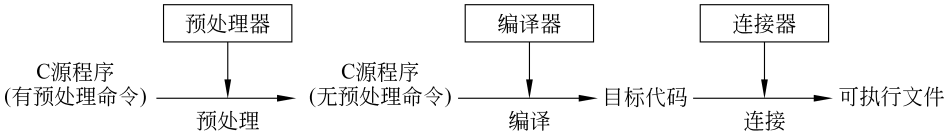

图 5.1　编译、连接处理过程

预处理命令不是 C++ 本身的组成部分,更不是 C++ 语句,它是 C++ 标准规定的可以出现在 C++ 源程序文件中的命令。这些命令必须以“#”开头,结尾不加分号,可以放置在源程序中的任何位置,其有效范围是从出现位置开始到源程序文件末尾。

预处理命令的操作对象是编译器和连接器,用来设置程序编译和连接时的各种参数,当编译工作完成后,预处理命令的作用即告完成,因而它不会出现在目标代码和可执行文件中。预处理命令是 C++ 的一个重要特点,合理地使用预处理功能,可以优化程序设计环境,提高程序的通用性、可读性和可移植性。

C++ 标准提供了多种预处理命令,如宏定义、文件包含和条件编译等,本章介绍常用的预处理命令。

5.1　宏定义

在 C++ 源程序中允许用一个标识符来代表一个字符文本,称为宏,标识符为宏名。宏是由宏定义命令事先定义的。预处理时,对程序中所有后续的宏名实例(称为宏引用),

预处理器都用字符文本去替换,称为宏替换或宏展开。

宏定义通常用于定义程序中的符号常量、类型别名、运算式代换和语句代换等,其命令为#define,分为不带参数的宏定义和带参数的宏定义。

5.1.1 不带参数的宏定义

不带参数的宏定义的命令形式为

```
#define  宏名  字符文本
```

其中:

(1) 宏名按标识符语法取名,习惯上用大写字母,以便与变量等其他名称有所区别。

(2) 字符文本可以为C++允许的标识符、关键字、常量数值、表达式及各种符号等。

(3) 宏名两侧至少用一个空白符间隔,且这个空白符不属于字符文本。

预处理时,程序中所有的宏名将被字符文本完全替换,然后再进行编译。

例如有宏定义

```
#define PI  3.1415926
```

那么程序代码

```
L=2*PI*r;                                    //PI宏引用
```

在预处理时宏替换为

```
L=2*3.1415926*r;
```

又如有宏定义

```
#define M y*y+ 5*y
```

那么程序代码

```
S=3*M+4*M+5*M;                               //M宏引用
```

在预处理时宏替换为

```
S=3*y*y+5*y+4*y*y+5*y+5*y*y+5*y;
```

使用宏定义,可以减少程序中重复书写某些字符文本的工作量,不容易出错;而且程序员习惯性将常量值定义为符号常量,无论是编写程序或是阅读程序都有记忆简单、见名知义的优点。

【例 5.1】 计算半径为 r 的圆周长、圆面积、圆球表面积和圆球体积。

程序代码如下:

```
1    #include <iostream>
2    using namespace std;
3    #define PI  3.1415926                   //不带参数的宏定义
4    int main()
```

```
5    {
6        double r,L,S,SQ,V;
7        cin >>r;                                    //输入半径
8        L=2 * PI * r;                               //计算圆周长
9        S=PI * r * r;                               //计算圆面积
10       SQ=4.0 * PI * r * r;                        //计算圆球表面积
11       V=4.0 * PI * r * r * r/3.0;                 //计算圆球体积
12       cout<<"L="<<L<<",S="<<S<<",SQ="<<SQ<<",V="<<V<<endl;
13       return 0;
14   }
```

程序运行情况如下：

```
2.0↙
L=12.5664,S=12.5664,SQ=50.2655,V=33.5103
```

使用不带参数的宏定义需要注意以下 6 点。

(1) 宏定义用宏名来代表一个字符文本，在宏替换时又以该字符文本取代宏名，这只是一种简单的替换。字符文本中可以包含任何字符，预处理器对它不作任何语法检查，即使有错误，也只有在编译已经宏替换后的源程序时才会发现。因此不要在字符文本中放置任何多余的字符，如在行末加分号，否则它们也将作为字符文本的组成部分。例如：

```
#define PI  3.1415926;
S=2 * PI * r;                          //错误，宏替换为 S=2 * 3.1415926; * r;
```

特别地，C++ 标准允许在宏定义的字符文本后面出现 C++ 注释，例如：

```
#define E  2.718281828                 //自然对数
#define G  9.8                         //重力加速度
```

预处理时会忽略所有这样的注释。

(2) 宏替换时使用完整的字符文本替换宏名，既不会少，也不会增加额外的字符。因此在一些运算式代换中，字符文本中有无括号对宏替换的结果是有影响的。例如：

```
#define M1  a+b
#define M2  (a+b)
L=M1 * M1;                             //宏展开为 L=a+b * a+b;
L=M2 * M2;                             //宏展开为 L=(a+b) * (a+b);
```

(3) 源程序中的字符串常量、注释或标识符的一部分若有与宏名相同的字符，不会进行宏替换。假定已定义 PI 宏，则

```
cout<<"PI="<<xPI * y<<endl;            //xPI 是变量名
```

不进行任何宏替换，因为代码中出现的 PI 均不是宏名。

(4) 一个宏名不要重复定义两次以上，否则引用的宏总是最后一次的宏定义；若要进行新的宏定义，应该先使用 # undef 命令。

#undef 命令的作用是取消已有的宏定义,一般形式为

```
#undef  宏名
```

取消以后的宏名不再有定义,程序中若继续引用它将导致错误。例如:

```
#define MXY (x*x+y*y)
#define MXY (x*x+2*x*y+y*y)          //重复宏定义
cout<<MXY<<endl;                     //MXY 宏替换为(x*x+2*x*y+y*y)
#define MAB  a+b
cout<<MAB<<endl;                     //MAB 宏替换为 a+b
#undef MAB                           //取消 MAB 宏定义
cout<<MAB-2<<endl;                   //错误,MAB 未定义
#define MAB  a*b                     //重新定义 MAB
cout<<MAB<<endl;                     //MAB 宏替换为 a*b
```

(5) 一个宏的作用范围是从定义位置开始到#undef 命令结束,如果没有对应的#undef 命令,则宏的作用范围到源程序文件末尾结束。通常将宏定义写在源程序文件的开头或头文件中。

(6) 宏定义允许嵌套,即在宏定义的字符文本中可以引用已经定义的宏名,在宏替换时由预处理器层层替换。例如:

```
#define WIDTH 80
#define LENGTH (WIDTH+10)
L=WIDTH*a;                           //宏展开为 L=80*a;
S=LENGTH*b*WIDTH ;                   //宏展开为 S=(80+10)*b*80;
```

5.1.2 带参数的宏定义

带参数的宏定义的命令形式为

```
#define  宏名(参数表)  字符文本
```

带参数的宏的引用形式为

```
宏名(引用参数表)
```

其中:

(1) 参数表允许多个参数,用逗号分隔,称为形式参数(不同于函数的形参概念)。

(2) 字符文本中包含所指定的参数文本,出现次序和数目没有任何限制。

(3) 引用参数表与宏定义的形式参数要求一一对应。

预处理时,预处理器先将宏引用的引用参数文本对应地替换宏定义字符文本中的参数,接下来进行宏替换,然后再进行编译。

例如,有以下宏定义:

```
#define max(a,b) (((a) >(b))?(a) : (b))
```

形式参数按顺序为 a 和 b，而程序代码

```
L=max(x-y,x+y);                    //max 宏引用
```

引用参数文本按顺序为 x－y 和 x＋y。

先将引用参数文本对应地替换字符文本中的参数，字符文本中其他内容不变，则字符文本置换为

```
(((x-y) >(x+y))?(x-y) : (x+y))
```

因此预处理时宏替换为

```
L=(((x-y) >(x+y))?(x-y) : (x+y))
```

使用带参数的宏定义需要注意以下 4 点。

(1) 宏名与“(参数表)”之间不能有空白符，否则命令形式会被理解为不带参数的宏定义，而“(参数表) ”是字符文本的一部分。例如：

```
#define AREA  (r) PI * r * r
```

AREA 被理解为不带参数的宏，“(r) PI * r * r”组成了它的字符文本，因此：

```
S=AREA(x);                         //AREA 宏引用
```

展开为

```
S=(r) PI * r * r(x)
```

显然是不对的。

(2) 字符文本中的参数必须是由各种符号、空白符分隔出来的独立字符文本。例如：

```
#define MSET1(arg) x=Aarg+2;
#define MSET2(arg) x=A+arg;
MSET1(1)                           //宏替换为 x=Aarg+2; arg 没有被替换
MSET2(1)                           //宏替换为 x=A+1; arg 已被替换
```

Aarg 字符文本不会进行 arg 参数替换，因为 Aarg 是一个不可分割的整体。

(3) 引用参数文本替换字符文本中的参数时，只是简单地做文本替换，某些表达式的宏定义中，这种简单处理可能会得到不符合原意的替换结果。例如：

```
#define POWER(a) a * a             //计算 a 的平方
```

如果宏引用为

```
S=POWER(x);                        //宏替换为 S=x * x
```

这是正确的，但如果宏引用为

```
S=POWER(x+y);                      //宏替换为 S=x+y * x+y
```

得到的宏扩展“x＋y * x＋y”显然与“(x＋y) * (x＋y)”原意不符。

解决这个问题有两种方法。

一是给引用参数文本加上括号。例如：

```
S=POWER((x+y));                    //宏替换为 S=(x+y)*(x+y)
```

二是在宏定义时给字符文本中的参数加上括号。例如：

```
#define POWER(a) (a)*(a)           //计算 a 的平方
S=POWER(x+y);                      //宏替换为 S=(x+y)*(x+y)
```

在实际编程中,第二种方法更稳妥。

(4) 无论是带参数的宏定义或是不带参数的宏定义,均可以使用行连接符"\"得到多行宏定义,进而得到具有复杂功能的宏。例如：

```
1   #define PRINTSTAR(n) { \
2       int i,j; \
3       for(i=1;i<=n;i++) { \
4           for (j=1;j<=i;j++) \
5               cout<<"*"; \
6           cout<<endl; \
7       } \
8   } \
9
```

那么 PRINTSTAR(5)宏引用的结果实际上是如下程序代码：

```
1   {
2       int i,j;
3       for(i=1;i<=5;i++) {
4           for (j=1;j<=i;j++)
5               cout<<"*";
6           cout<<endl;
7       }
8   }
9
```

这里外加一对大括号({ })的目的是形成一个复合语句,则"int i,j;"变量定义就是局部区域的变量,它们与外部不会有任何冲突。需要注意,宏定义的最后要连接一个空行(例如第 9 行),这样宏替换时才会有相应的换行。PRINTSTAR(5)的运行结果如下：

```
*
**
***
****
*****
```

可以用不同的参数引用宏 PRINTSTAR,得到数目不同的星号输出,例如：

```
PRINTSTAR(5)                       //宏替换为一段程序代码
```

```
PRINTSTAR(8)                    //宏替换为一段程序代码
PRINTSTAR(10)                   //宏替换为一段程序代码
```

从这个例子可以看出，如果善于利用宏定义，可以实现程序的简化。

带参数的宏定义的引用与函数调用在语法上比较相似，例如，在调用函数时，在函数名后的括号内写实参，要求实参与形参的顺序对应和数目相等。但它们基本含义不同，主要区别如下。

（1）函数调用时会先计算实参表达式的值，然后参数值传递给形参，程序指令会转到函数内部开始执行。而带参数的宏定义只是参数文本替换，不存在计算实参、参数传递、跳转执行等。

（2）函数调用是在程序运行时执行的，它会为形式参数分配临时的内存单元。而宏在预处理阶段替换，不会为形式参数分配内存单元，而且也没有返回和返回值的概念。

（3）函数调用对实参和形参都要定义类型，且要求二者的类型一致，如果不一致，会进行类型转换。而宏定义不存在类型问题，它的形式参数和引用参数都只是一个文本记号，宏替换时进行文本置换。

（4）无参数函数调用必须包含括号，无参数宏定义引用时不需要括号。例如：

```
#define PI  3.1415926           //宏定义
int fun();                      //函数原型
x=fun();                        //函数调用
x=PI;                           //宏引用
```

（5）每一次宏引用，宏替换后都会使源程序增长，相当于将宏定义的字符文本“粘贴”到源程序中一次，而函数调用代码是复用的。宏替换会占用编译时间，函数调用则会占用运行时间。

（6）宏定义与第 4 章的内联函数非常相似。两者区别在于：宏是由预处理器对宏进行替换，它是在代码处不加任何检验的简单替换；而内联函数是通过编译器来实现的，它有函数的特性，只是在需要用到的时候，内联函数像宏一样地展开，取消了函数的参数入栈，减少了调用的开销。内联函数要做参数类型检查，这是内联函数跟宏相比的优势。

【例 5.2】　宏引用和函数调用的区别。

程序代码如下：

```
1    #include <iostream>
2    using namespace std;
3    int M1(int y)
4    {
5        return((y)*(y));
6    }
7    #define M2(y) ((y)*(y))
8    int main()
9    {
```

```
10      int i,j;
11      for (i=1,j=1;i<=5;i++) cout<<M1(j++)<<" ";          //函数调用处理
12      cout<<endl;
13      for (i=1,j=1;i<=5;i++) cout<<M2(j++)<<" ";          //宏引用处理
14      cout<<endl;
15      return 0;
16  }
```

程序运行结果如下：

```
1 4 9 16 25
1 9 25 49 81
```

例子中函数 M1 计算的表达式和宏 M2 计算的表达式均为(y) * (y)，且函数调用为 M1(j++)，宏引用为 M2(j++)，形式也是相同的。从输出结果来看，却大不相同。原因是循环 5 次函数调用，使得每次 j 自增 1，故输出 1～5 的平方值。而宏引用展开为“((j++) * (j++))”，循环 5 次宏引用，使得 j 每次增加 2，故输出 1、3、5、7、9 的平方值。

从上述分析中可以看出函数调用和宏引用在形式上相似，在本质上是完全不同的。

5.1.3 # 和 ## 预处理运算

C++ 标准为预处理命令定义了两个运算符：# 和 ##，它们在预处理时被执行。

运算符的作用是文本参数“字符串化”，即出现在宏定义字符文本中的 # 把跟在后面的参数转换成一个 C++ 字符串常量。例如：

```
#define PRINT_MSG1(x) cout<<#x;
#define PRINT_MSG2(x) cout<<x;
PRINT_MSG1(Hello World);              //正确,宏替换为 cout<<"Hello World";
PRINT_MSG1("Hello World");            //正确,宏替换为 cout<<"\"Hello World\"";
PRINT_MSG2(Hello World);              //错误,宏替换为 cout<<Hello World;
PRINT_MSG2("Hello World");            //正确,宏替换为 cout<<"Hello World";
```

简单来说，# 参数的作用就是对这个参数替换后，再加双引号(“”)括起来，变为“参数”。

运算符的作用是将两个字符文本连接成一个字符文本，如果其中一个字符文本是宏定义的参数，连接会在参数替换后发生。例如：

```
#define SET1(arg) A##arg=arg;
#define SET2(arg) Aarg=arg;
SET1(1);                              //宏替换为 A1=1;
SET2(1);                              //宏替换为 Aarg=1;
```

A 字符与 ## arg 参数连接在一起形成了 A1，而对于 Aarg 字符文本，不会进行 arg 替换。

5.1.4 预定义宏

C++ 标准中预先定义了一些有用的符号常量，主要是编译信息，如表 5.1 所示。

表 5.1　标准预定义符号常量

符号常量	类　　型	说　　明
__DATE__	字符串常量	编译程序日期(形式为“MM DD YYYY”,例如“May 4 2006”)
__TIME__	字符串常量	编译程序时间(形式为“hh:mm:ss”,例如“10:20:05”)
__FILE__	字符串常量	编译程序文件名
__LINE__	int 型常量	当前源代码的行号
__STDC__	int 型常量	若为 1 说明此程序兼容 ANSI C 标准
__cplusplus	int 型常量	若编译的是 C++ 程序,值定义为 197711L

其中,“__”为两个下画线,__DATE__和__TIME__用于指明程序编译的时间,__FILE__和__LINE__用于调试目的,__cplusplus 检测编译文件是否支持 C++ 。

5.2　文件包含

文件包含命令的作用是把指定的文件插入该命令所处的位置上取代该命令,然后再进行编译处理,相当于将文件的内容“嵌入”当前的源文件中一起编译。

文件包含命令为#include,有两种命令形式:

```
#include <头文件名>
```

或

```
#include "头文件名"
```

说明:

(1) 一个#include 命令只能包含一个头文件,包含多个头文件要用多个#include 命令,且每个文件包含命令占一行。通常,头文件的扩展名为.h 或.hpp。

(2) 第一种形式与第二种形式的区别是编译器查找头文件的搜索路径不一样。第一种形式仅在编译器 INCLUDE 系统路径中查找头文件,第二种形式先在源文件所处的文件夹(用户路径)中查找头文件,如果找不到,再在系统路径中查找。

一般地,如果调用标准库函数或者专业库函数包含头文件时,使用第一种形式;包含程序员自己编写的头文件时,将头文件放在源文件所处的文件夹中且使用第二种形式。

(3) 头文件的内容通常是函数声明、全局性常量、数据类型声明和宏定义等信息,一般不包括定义,如函数定义和变量定义等。所起的作用就是为其他程序模块提供声明性信息,而定义、函数实现代码等应放在源文件中。

在实际编程中,如果程序是由多个源文件组成的,一般采用工程方式来集合,而不是使用文件包含命令。

1. 文件包含的路径问题

文件包含命令中的头文件名可以写成绝对路径的形式。例如:

```
#include "C:\DEV\GSL\include\gsl_linalg.h"
#include <C:\DEV\SDL\include\SDL.h>
```

这时直接按该路径打开头文件,此时第一种形式或第二种形式的命令没有区别。请注意,由于文件包含命令不属于C++语法,因此"C:\DEV\GSL\include\gsl_linalg.h"不能理解为字符串,其中的"\"不要写成"\\"。

头文件名也可以写成相对路径的形式。例如:

```
#include <cmath>
#include <zlib\zlib.h>
#include "user.h"
#include "share\a.h"
```

这时的文件包含命令是相对系统INCLUDE路径或用户路径来查找头文件的。

假设编译器系统INCLUDE路径为"C:\DEV\MinGW\include",则

```
#include <cmath>                //cmath在C:\DEV\MinGW\include
#include <zlib\zlib.h>          //zlib.h在C:\DEV\MinGW\include\zlib
```

假设用户路径为"D:\Devshop",则

```
#include "user.h"        //user.h在D:\Devshop或C:\DEV\MinGW\include
#include "share\a.h"     //a.h在D:\Devshop\share或C:\DEV\MinGW\include\share
```

如果在上述路径中找不到头文件,会出现编译错误。

2. 文件包含的重复包含问题

头文件有时需要避免重复包含(即多次包含),例如一些特定声明不能多次声明,而且重复包含增加了编译时间。这时可以采用以下两个办法之一。

(1) 使用条件编译。例如:

```
#if !defined(_FILE1_H_C6793AB5__INCLUDED_)
#define _FILE1_H_C6793AB5__INCLUDED_
…                                       //头文件内容
#endif
```

即将头文件内容放在一个条件编译块中,第1次编译时编译条件成立,故继续往下编译,第2行使编译条件为假,这样再次编译头文件时,头文件内容就不会编译了。条件中的宏定义"_FILE1_H_C6793AB5__INCLUDED_",为了与其他编译条件相区别,故意写得很长、很怪。

(2) 使用特殊预处理命令#pragma。例如:

```
#pragma once
…                                       //头文件内容
```

即在头文件第1行增加这个预处理命令,它的意思是:在编译一个源文件时,只对该文件