第5章

预处理程序

预处理程序的概念来源于C语言，它是C语言程序设计的一部分。早期C语言的一个独特的优点就是提供了预处理功能。C语言的预处理程序为C语言开发的软件能够移植到不同的计算机系统上提供了有效的工具。


视频讲解

5.1 预处理

预处理是在编译程序开始进行第一遍扫描即词法扫描和语法分析之前，对编写的C语言源代码文件进行的文件处理。这种可以对C语言源代码文件进行预先处理的软件称为预处理程序。

预处理程序和编译程序一样都是一个可运行的软件工具。预处理程序读出源代码，对其中内嵌的指示字(directive)进行相应处理，产生源代码的修改版本，修改后的版本紧接着会被编译程序读入进行处理，最后变成二进制目标代码。

预处理程序在很长一段时间内都被认为是C语言特有的内容。最近这些年也被有限地用于预处理其他语言的源代码文件，例如用它实现了FORTRAN和Java的条件编译。

在“2.5 软件安装”里安装的集成开发环境里就有C语言的预处理程序，它被作为GCC(GNU Compiler Collection，GNU编译程序集合)的一部分，安装在指定安装路径C:\Qt\Tools\mingw900_64\bin下面，预处理程序的可执行文件名是cpp.exe。本书所讲的预处理程序指示字内容全部是基于安装的Qt Creator集成开发环境的C语言预处理程序的指示字。使用预处理程序参数选项可以查看预处理结果。

```
cpp -E -dD main.c -o precompiler.txt
```

-E是只预处理，不编译、汇编、链接。

-d后面的字母D、I、M、N对预处理程序有特殊含义，如下所示。

D：与-E选项一同使用时，除了预处理后的正常输出外，还输出所有的宏定义。

I：与-E选项一同使用时，预处理程序输出#include指示字，还输出其他预处理程序的输出。

M：与-E选项一同使用时，预处理程序将在所有的预处理后输出有效的宏定义列表。

N：与-E 选项一同使用时，除了在预处理后的正常输出外，还包含以 # define name 简单形式定义的所有宏的列表。

上述命令行方式将产生预处理后的文件 precompiler. txt，在 Windows 10 下可以用命令行 type precompiler. txt 查看内容；在 CentOS 8. 5(Linux)下可以在终端方式下用命令行 cat precompiler. txt 查看内容。在各种图形化窗口环境下也可以用系统自带的文本编辑器打开查看。图 5. 1 是安装路径 C:\Qt\Tools\mingw900_64\bin 下面的 C 语言预处理程序位置。

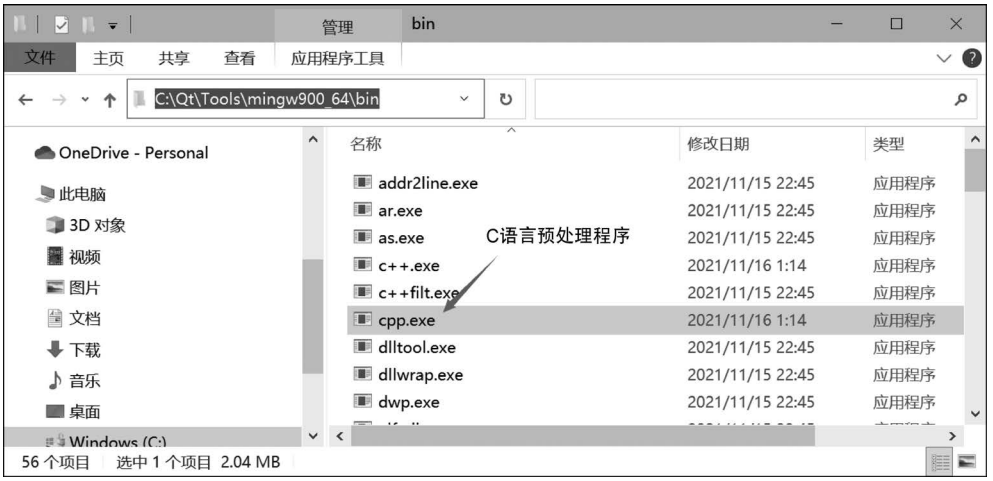

图 5. 1　C 语言预处理程序位置

5.2　指示字

源代码中的预处理指令称为指示字，从源代码中可以很容易地发现，它们以 # 开始，每行第一个非空字符都是 #，这是指示字的标志。# 后面紧跟着指示字的关键字。指示字书写格式如下。

#关键字　其他信息

C 语言预处理程序的常用指示字及其描述如表 5. 1 所示。

表 5. 1　C 语言预处理程序的常用指示字及其描述

指　示　字	描　　述
# define	定义宏名字，预处理程序会把这个宏扩展到使用该名字的位置
# undef	删除前面用 # define 指示字创建的定义
# error	产生出错消息，挂起预处理程序
# warning	由预处理程序创建一个警告信息
# include	查找指示字列表，直到找到指定的文件，然后将文件内容插入，就好像在文本编辑器中插入一样
# include_next	和 # include 一样，但该指示字从找到当前文件的目录之后的目录开始查找
# if	如果计算算术表达式的结果为非 0 值，就编译指示字和它匹配的 # endif 之间的代码
# ifdef	如果已经定义了指定的宏，就编译指示字和它匹配的 # endif 之间的代码

续表

指　示　字	描　　述
#ifndef	如果没有定义指定的宏，就编译指示字和它匹配的#endif之间的代码
#else	如果#if、#ifdef或#ifndef为假，则提供一个用于编译的可选代码集合
#elif	由#if指示字提供一个用于计算的可选表达式
#endif	与#if、#ifdef、#ifndef配对的、不可缺少的指示字
#line	指出行号及可能的文件名，报告给编译程序，用于创建目标文件中的调试信息
#pragma	提供额外信息的标准方法，可用来指出一个编译程序或一个平台
##	连接运算符，可用于宏内，将两个字符串连接成一个

根据指示字指令要求，预处理程序是可以修改源代码的，而不是修改指示字。预处理程序总是对分散在C语言源程序代码中的特定指示字（用#作前缀的指令）进行识别和处理。

C语言提供了多种预处理指示字（预处理命令）功能，如文件包含、宏定义、条件编译等，合理地使用它们会使编写的程序便于阅读、修改、移植和调试，也有利于模块化程序设计。下面会详细介绍这些预处理命令。

5.3　宏定义

#define称为宏定义指示字，它也是C语言预处理程序指示字的一种。所谓宏定义，就是用一个用户标识符来表示一个字符串，如果在后面的代码中出现了该用户标识符，那么就全部替换成该用户标识符后面指定的字符串。在宏定义中可以指定参数作为扩展宏的一部分。

大多数宏定义在效果上为常数。这些名字传统上为大写字母。例如，下面的指示字定义了宏PI和MAXINT，无论何时在C语言源代码中出现该宏的地方都会替换为字符串"3.1415926"和"2147483647"。

```
#define  PI  3.1415926
#define  MAXINT  2147483647
```

在C语言源代码中定义PI、MAXINT之后，凡是源代码中出现PI、MAXINT的地方，都会被宏后面定义的字符串"3.1415926"和"2147483647"替换。通过运行“5.1 预处理”里的cpp预处理程序，带预处理参数选项来看这种效果。完整的示例程序代码如下。

```
#include <stdio.h>
#include <math.h>
#define  PI  3.1415926
#define  MAXINT  2147483647
int main()
{
    double dx = 0.0f;
    dx = sin(PI/6); printf("sin(PI/6) = %E\n",dx);
    dx = sin(PI*1.5/2); printf("sin(PI*1.5/2) = %E\n",dx);
    if(dx<PI) printf("%E比PI小\n",dx);
    else printf("%E大于或等于PI\n",dx);
    printf("最大的整数是%d\n",MAXINT);
    return 0;
}
```

经过预处理后，根据设置的参数，预处理程序保留了所有的宏定义和处理后的正常输出。可以看到这两个宏被扩展到源代码中。预处理后的C语言程序代码如图5.2所示。

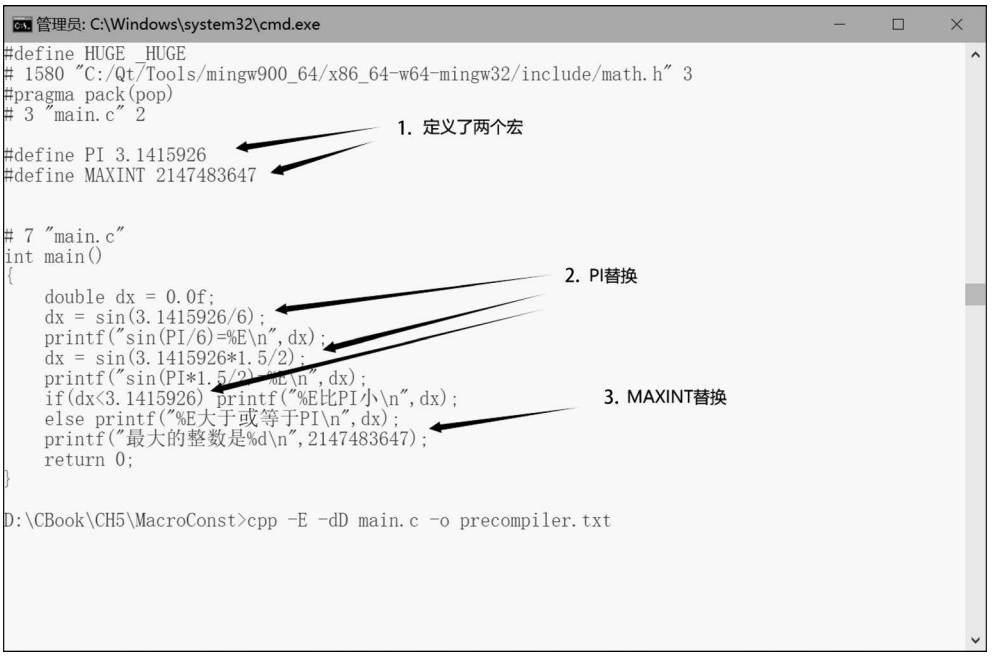

图5.2 常量宏定义预处理后的C语言程序代码

宏定义还可以不带字符串，也就是只有宏标识符。这种方式主要是给#indef和#ifndef提供测试的标记，在头文件的编写方面有特殊的效果，在后面的“5.14 指示字使用技巧”中会介绍这部分内容。只有宏标识符的定义规则如下所述。

```
#define  HUFMANHEADER_H
```

宏定义也遵循C语言的先定义、声明后使用的原则，因此在宏定义之前使用宏，不会出现宏替换。只有在定义之后才会在使用宏的地方替换成宏定义的字符串，例如下面的示例代码。

```
int  nSum = 0, B = 29;
#define  SB  10
    nSum = SB + B;                    //这个地方会保留 B,会使用变量 B
    printf("nSum = SB + B;%d  %d  %d\n",nSum,SB,B);
#define  B  99
    nSum = SB + B;                    //这个地方 SB 和 B 全部被替换
    printf("nSum = SB + B;%d  %d  %d\n",nSum,SB,B);
```

上述宏定义示例代码经过预处理程序处理之后，变成如下源代码，看看有什么变化。

```
int nSum = 0, B = 29;
#define SB 10
    nSum = 10 + B;                    //还会用变量 B
    printf("nSum = SB + B;%d  %d  %d\n",nSum,10,B);
#define B 99
    nSum = 10 + 99;                   //变量 B 已经无影无踪了
    printf("nSum = SB + B;%d  %d  %d\n",nSum,10,99);
```

5.4 带参数的宏定义

在C语言中宏的用处很多,也很强大。很多学习C语言编程的人员很少接触到预处理程序方面的内容,在工作以后,开发软件时也很少在源代码中使用预处理指示字。使用预处理指示字编写C语言源代码后才会理解预处理指示字的强大功能。

前面讲述了定义普通的带字符串值和不带字符串值的宏的定义规则,在宏定义规则中,还有一种很强大的带参数的宏定义规则。更进一步,在C99标准中宏不但可以带确定个数的参数,还可以带不确定数目的参数。

下面是一个大家都熟悉的宏,它使用参数来创建一个表达式,可以返回两值中的较大值。

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

在C语言源代码中出现max(a,b)的任何地方,都会被宏后面定义的字符串((a)>(b) ? (a) : (b))替换。在源代码中使用宏时,其中参数a,b可以用其他标识符(变量名,或表达式都可以)替换。

需要特别注意的是,宏名标识符内不能出现空格,带参数的宏名字与(号之间也不能出现空格,一旦这些地方出现空格,后面的内容都会被认为是宏的字符串值。

通过“5.1 预处理”里的cpp预处理程序,带参数命令来看这种效果。完整的示例程序代码如下。

```
#include <stdio.h>
#define  max(a,b)  ( (a) > (b) ? (a) : (b) )
int main()
{
    int   x = 12, y = 18;
    float fx = 8.9f, fy = 3.2f;
    double dx = 3.9E+123, dz = 9.2E+99;
    printf("x = 12, y = 18 最大值是:%d\n",max(x,y));
    printf("fx = 8.9f, fy = 3.2f 最大值是:%E\n",max(fx,fy));
    printf("dx = 3.9E+123, dz = 9.2E+99 最大值是:%E\n",max(dx,dz));
    printf("宏定义太方便了!\n");
    return 0;
}
```

源代码中printf输出函数参数使用了已经定义的宏max(a,b),经过预处理后,根据设置的参数,预处理程序保留了所有的宏定义和处理后的正常输出。可以看到这个宏被扩展到源代码中printf函数里了,宏参数也都正确处理了。处理后的C语言程序代码如图5.3所示。

1. 参数个数不确定的宏定义

在C99标准中定义宏时,参数个数不确定的情况可以用省略号代替,这些参数被保存在字符串中作为变量__VA_ARGS__,它会在宏内部进行扩展。参数个数不确定的宏有如下两种形式。

```
#define MYOUT(…)  printf(__VA_ARGS__)
#define YOUROUT(a,b,…) printf("Line %d : Row %d ",a,b); \
                           printf(__VA_ARGS__)
```

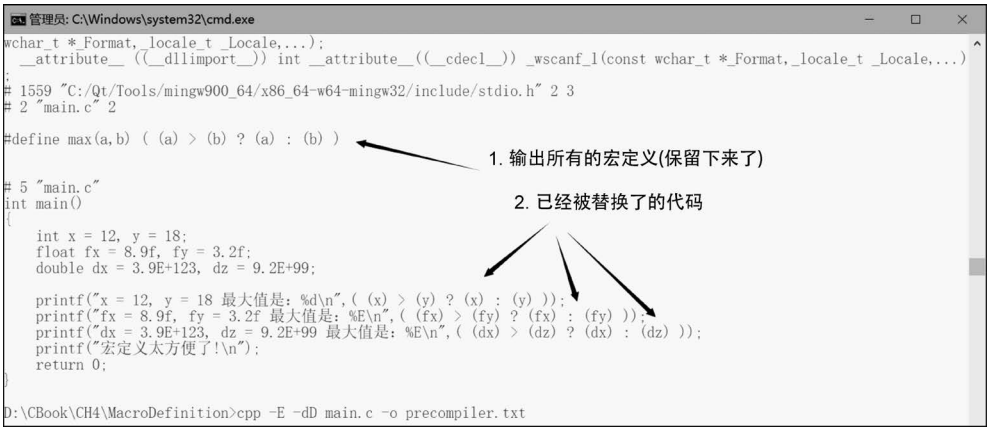

图 5.3 处理后的 C 语言程序代码

第二种宏定义中,确定的参数是两个,不确定的参数放在确定参数后面,宏定义一行写不完可以在行尾用\说明有续行,还可以多次续行写完宏定义。

在进行了上述的宏定义之后,在源代码中如果出现如下代码。

```
MYOUT("%-s %d\n","出现异常的门牌号是",5);
YOUROUT(76,nSum*3, "号武汉服装仓库\n");
```

预处理之后替换成如下代码。

```
printf("%-s %d\n","出现异常的门牌号是",5);
printf("Line %d : Row %d ",76,nSum*3); printf("号武汉服装仓库\n");
```

2. 字符串宏参数的特殊替换

程序中如果字符串内出现宏的名字,这时预处理程序不会进行替换。但是字符串宏参数是可以加入字符串中的。#加宏的参数可以将宏参数字符串化,从而做到修改 C 语言源代码字符串的效果。例如下面的例子中定义的宏。

```
#define INSERTSTR(EMBEDDED_STR) \
    printf("现在要插入宏参数:"#EMBEDDED_STR" 插入结束.\n")
#define GRADE(STR,NAME)  "欢迎"#STR"年级学生:"#NAME
```

在进行了上述的宏定义之后,在源代码中如果出现如下代码。

```
INSERTSTR(我是宏参数 EMBEDDED_STR);
printf("%s\n",GRADE(九,大黄蜂));
```

经过预处理程序处理之后,替换成如下代码。

```
printf("现在要插入宏参数:""我是宏参数 EMBEDDED_STR"" 插入结束.\n");
printf("%s\n","欢迎""九""年级学生:""大黄蜂");
```

5.5 #undef 指示字

#undef 指示字用于将之前由#define 指示字定义的宏取消。在不需要使用已经定义的宏,或者需要用这个宏标识符重新定义成别的新值时,就可以用#undef 指示字,例如:

```
#define DESKCOLOR  COLOR
```

```
#define  COLOR  4009242
    int  wudesk = DESKCOLOR;
#undef COLOR
#define  COLOR  3153936
    int  sundesk = DESKCOLOR;
```

经过预处理程序处理之后，上述代码会被替换成如下代码。

```
#define DESKCOLOR COLOR
#define COLOR 4009242
    int wudesk = 4009242;
#undef COLOR
#define COLOR 3153936
    int sundesk = 3153936;
```

C语言源代码中虽然wudesk、sundesk都是用DESKCOLOR赋值，但是它的字符串值是COLOR，于是取消宏COLOR之后再定义给宏COLOR不同的值就可以不改变C语句而实现语句变化。这就是嵌套的宏定义与重新定义宏的组合用处。

另外源代码中用到宏，但是却没有给出宏定义，例如：

```
int wudesk = COLOR1;
int sundesk = COLOR2;
```

这种源代码需要在编译时用－D参数选项传入宏COLOR1和宏COLOR2，命令行参数如下。

```
gcc -DCOLOR1=100 -DCOLOR2=900 main.c -o abc.exe
```

5.6　#error与#warning指示字

#error指示字会引起预处理程序报告致命错误或中断。它可以用来捕获尝试按照某种不可能工作的形式进行编译的条件。例如，下面的例子只有在定义了__MAC_10_15的情况下才能成功预处理并编译、链接。如果没有定义宏__MAC_10_15，不管是在集成开发环境构建，还是在gcc命令行方式下编译，更或是在cpp命令行方式预处理，编译都过不去，也不可能产生可执行程序。

```
#include <stdio.h>
int main()
{
#ifndef __MAC_10_15
#error "这个程序只能在 macOS 10.15 上运行"
#endif
    printf("Hello World!\n");
    return 0;
}
```

#warning指示字和#error指示字的工作原理一样，它的条件不是致命性错误，而是预处理程序在发出消息之后会继续进行下去，不会像#error那样停止。

```
#include <stdio.h>
int main()
{
```

```
#ifndef __MAC_10_15
#warning "这个程序生成 macOS 10.15 代码,\
          需要在 macOS 10.15 上运行"
#endif
    printf("Hello World!\n");
    return 0;
}
```

程序构建之后运行,可以打印出 Hello World!。程序运行结果如图 5.4 所示。

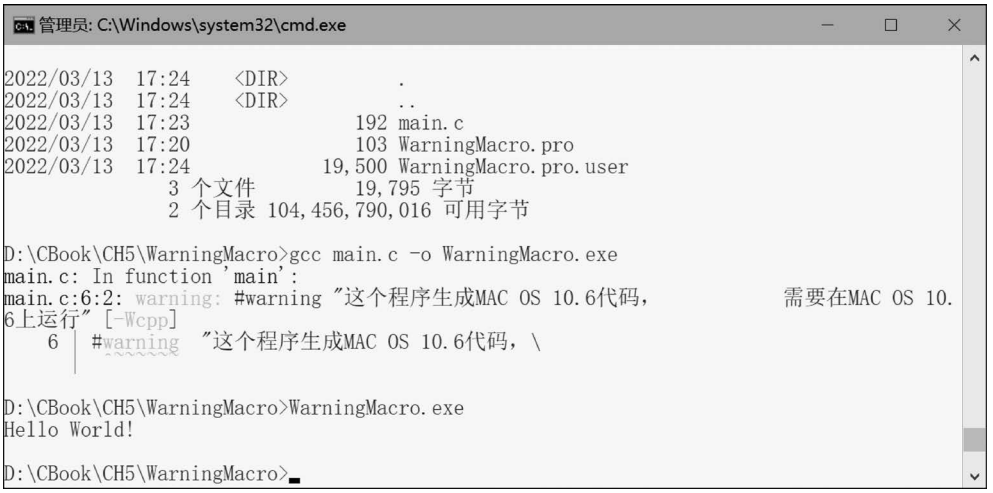

图 5.4 #warning 指示字的运行结果

5.7 #include 指示字

#include 指示字又称为文件包含命令,用来引入对应的头文件(.h 文件)。#include 指示字会查找指定的文件,将文件的内容插入当前源代码文件,从而把头文件和当前源文件连接成一个源文件,这与复制、粘贴的效果相同。这种方式包含的文件通常是头文件(.h 文件),扩展名为 h,但其实可以是任意名字和扩展名的文本文件。

#include 指示字有 3 种形式:第 1 种是#include 系统头文件,头文件名用尖扩号括起来;第 2 种是#include 自定义头文件,头文件名用双引号括起来;第 3 种是#include 预处理助记符,预处理助记符是预先用#define 指示字定义的宏,用于指定头文件的名字。这 3 种#include 头文件的代码示例如下所示。

```
#include <stdio.h>                    //#include <可带路径的标准库头文件>
#include <sys/socket.h>               //#include <可带路径的标准库头文件>
#include "..\declare\myHeader.h"      //#include "可带路径的自定义头文件"
#define LONGJMP <setjmp.h>            //#include 预处理助记符
#include LONGJMP                      //#include 预处理助记符
```

gcc 编译程序的-I 选项(字母 i 的大写字母)可用来修改指定的查找目录。使用此选项参数后,头文件查找由使用-I 选项说明的目录开始,然后继续查找系统目录的标准集合。例如在 D:\CBook\CH11\PtrDescription 目录下有 main.c、variable.c、variable.h 三个 C 语言源代码文件。现在将 main.c、variable.c 文件复制到 D:\WorkDir\Ipara 目录下,然后将

variable.h 文件复制到 D:\WorkDir\myster 目录下，这样.h 文件就分散在不同的目录下，如果直接用图 5.5 中的命令编译会出错，因为找不到 variable.h 文件。运行 gcc 时如使用如图 5.6 中-ID:\WorkDir\myster 参数选项就能顺利找到 variable.h 文件，通过编译生成 lpara.exe 可执行文件。

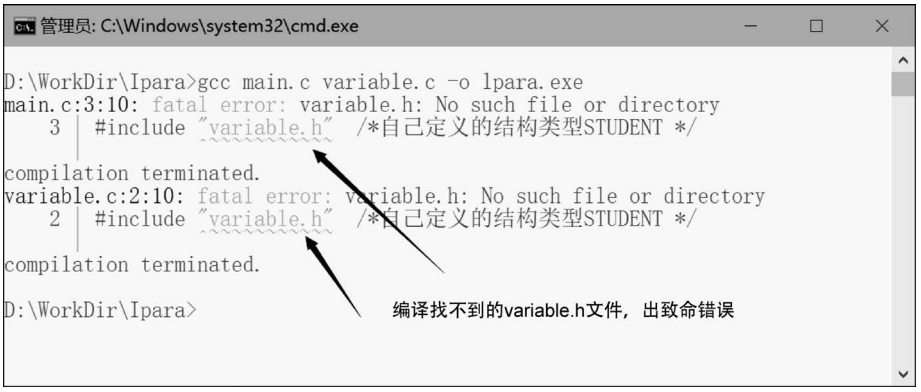

图 5.5　gcc 不使用参数-I 的使用效果

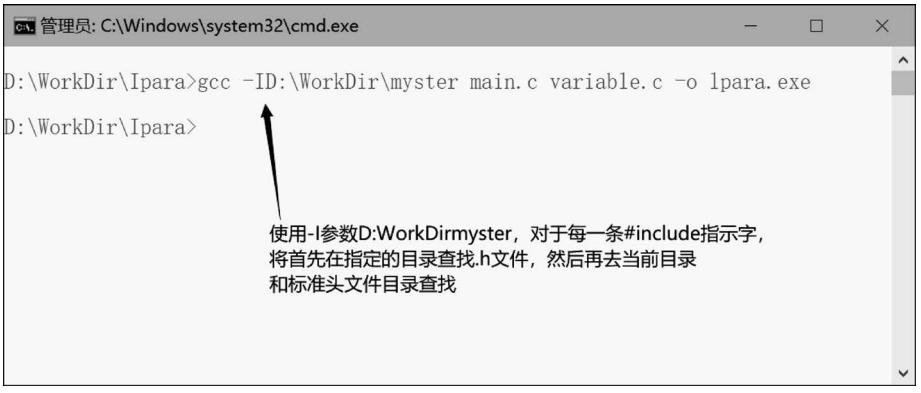

图 5.6　gcc 使用参数-I 找到.h 文件顺利通过编译生成可执行文件

5.8　#include_next 指示字

#include_next 指示字只用于某些特殊情况。它用在头文件内部来包含其他头文件，会令新头文件的查找由找到当前头文件的目录之后的目录开始。

例如，如果头文件的正常查找会依次查找目录 dir1、dir2、dir3、dir4 和 dir5，而当前头文件位于目录 dir3，则#include_next 指示字会要求在目录 dir4、dir5 中查找新的头文件。

该指示字可用于增加或修改系统头文件的定义，而不必修改文件本身。例如，系统头文件/usr/include/stdio.h 包含宏定义 getc，它会从输入流中读出单个字符。要改变这个宏定义，让它总是返回同一个字符，但保留头文件的其他内容，可以创建自己的 stdio.h 头文件，包含下面内容。

```
#include_next "stdio.h"
#undef getc
```

```
#define getc(fp) ((int)'G')
```

使用该头文件会包含系统的 stdio.h，以及自己重新定义的宏 getc(fp)。

5.9 #if、#elif、#else 和 #endif 指示字组

#if、#elif、#else 和 #endif 是一组互相配合的指示字，使用这一组指示字可以让预处理程序对 C 语言源代码文件进行条件预处理。根据 #if 和 #elif 指示字里面的逻辑表达式的值进行选择预处理不同的源代码块。

1. #if

#if 指示字后面加空格，然后跟一个表达式，预处理程序会对这个表达式进行运算，计算出表达式的值，并根据这个结果值做出相应的处理。如果结果值为非 0，就认为条件为真，会编译后面的代码块。如果结果值为 0，就认为条件为假，不会编译后面的代码块。例如，下面的字符串只有在宏 NEEDCOMP 未被定义为 0 的情况下才会声明。

```
#if NEEDCOMP
        char *conststr = "宏 NEEDCOMP 是非 0 值才会有这个字符串";
#endif
```

#if 指示字后面的表达式里可以使用有值的宏，它可以是算术表达式，也可以是关系表达式或者逻辑表达式。因此可以运用 C 语言里的相关运算符组合成复杂的表达式。

下面是应用于表达式和 #if 指示字的特征和规则。

(1) 表达式可以使用括号，用括号指出表达式计算的顺序。

(2) 表达式可包含整数常数，如果宏被定义为有值，也可以包含宏名。

(3) 表达式不可以使用 C 语言源代码中定义的变量，因为预处理时这些变量都不存在，编译程序还没有给任何这种变量初始化。

(4) 表达式可以使用的算术运算符是加(+)、减(-)、乘(*)、除(/)运算符，这些算术运算符和 C 语言中对应的整数算术运算符用法一致。所有算术运算符都可操作预处理程序(系统平台)支持的最大长度二进制位数的整数，通常为 64 位整数。

(5) 表达式可以使用的位运算符是左移(<<)和右移(>>)位运算符，这两个位运算符和 C 语言中对应的整数位运算符用法一致。

(6) 表达式可以使用的关系运算符是等于(==)、不等于(!=)、大于(>)、小于(<)、大于或等于(>=)、小于或等于(<=)，它们和 C 语言中的相应运算符一样。

(7) 表达式可以使用的逻辑运算符是逻辑或(||)、逻辑与(&&)、逻辑非(!)，它们和 C 语言中的相应运算符一样。

(8) 为 #if 指示字新增的预处理程序运算符是 defined 运算符。它可以用来判定宏操作数是否被定义了。例如，已经定义了宏 DESKCOLOR，那么下面的表达式就为真。

```
#if defined(DESKCOLOR)
```

逻辑非运算符(!)通常和 defined 运算符一起使用来测试宏是否被定义，如下所示。

```
#if  !defined(DESKCOLOR)
```

(9) 如果表达式中有用户标识符未被定义为宏，那么表达式的值总是会等于 0。

-Wundef 选项可用于在这种环境下产生警告信息。

(10) 具有参数的宏可以定义为只计算出结果 0。-Wundef 选项可用于在这种情况下产生警告。

2. #elif

#elif 指示字可用于提供一个可选的表达式。格式是 #elif 指示字后面加空格，然后跟着一个表达式，预处理程序会对这个表达式进行运算，计算出表达式的值，并根据这个结果值做出相应的处理。#if 指示字与 #elif 指示字组成的一组条件判断，按照先后出现的次序逐个判断，遇到任何一个条件为非 0 值时就不再进行其他条件判断，只处理此非 0 值后面的代码块。#elif 指示字不能出现在 #else 指示字与 #endif 指示字之间。

如果 #elif 中的表达式结果值为非 0，就认为条件为真，如果有 #else 指示字，那么 #elif 指示字与 #else 指示字之间的代码块会被继续预处理和编译；如果没有 #else 指示字，那么 #elif 指示字与 #endif 指示字之间的代码块会被继续预处理和编译。

如果 #elif 中的表达式结果值为 0，就认为条件为假，如果有 #else 指示字，那么 #elif 指示字与 #else 指示字之间的代码块不会被继续预处理和编译；如果没有 #else 指示字，那么 #elif 指示字与 #endif 指示字之间的代码块不会被继续预处理和编译，示例如下所示。

```
#if DESKCOLOR <= 512              //表达式结果值为非 0,到 #else 的代码块都会预处理、编译
#define COLOR 311
#elif DESKCOLOR > 1256            //#if 表达式结果值为 0,#elif 表达式结果值为非 0 下面代码
                                  //块都会预处理、编译
#define COLOR1 211
#elif DESKCOLOR == 1000           //#if 表达式结果值为 0,#elif DESKCOLOR > 1256 结果值也为 0
//#elif DESKCOLOR= 1000           //结果值为非 0,下面代码块都会预处理、编译
#define COLOR2 111
#else                             //#if 和 #elif 表达式结果值为 0,到 #endif 的代码块都会预处
                                  //理、编译.可以没有 else
#define COLOR 88
#endif //有 #if 就必须有 #endif 结尾
```

3. #else

#else 指示字可用来提供用于继续预处理和编译的可选代码。#if 指示字与 #elif 指示字后面的表达式结果值只要有一个为非 0，那么 #else 指示字与 #endif 指示字之间的代码块就不会被继续预处理和编译；#if 指示字与 #elif 指示字后面的表达式结果值全部为 0，那么 #else 指示字与 #endif 指示字之间的代码块就会被继续预处理和编译，示例如下所示。

```
#if DESKCOLOR <= 512
#define   COLOR   311             //DESKCOLOR <= 512 这一条被预处理
#else
#define   COLOR   88              //DESKCOLOR > 512 这一条被预处理
#endif
```

4. #endif

#endif 指示字是 #if 指示字的配对结尾指示字。每个 #if 指示字可以没有 #else 指示字，但是必须要有 #endif 指示字进行配对。如果 #if 指示字没有 #endif 指示字配对，则预处理程序会给出警告信息。

#if…#endif 指示字是可以嵌套使用的，例如下面的嵌套结构就是 stdio.h 系统头文

件中的代码。

```
#ifndef NULL
#ifdef __cplusplus
#ifndef _WIN64
#define NULL 0
#else
#define NULL 0LL
#endif /* W64 */              //这个对应的是#ifndef _WIN64
#else
#define NULL ((void *)0)
#endif                        //这个对应的是#ifdef __cplusplus
#endif                        //这个对应的是#ifndef NULL
```

5.10 #ifdef、#ifndef、#else 和 #endif 指示字组

#ifdef、#ifndef、#else 和 #endif 是一组互相配合的指示字，使用这一组指示字可以根据 #ifdef 指示字或者 #ifndef 指示字后面的宏是否定义，判断是否让预处理程序对 C 语言源代码文件进行条件预处理。根据 #ifdef 指示字或者 #ifndef 指示字后面的宏是否定义选择预处理、编译不同的源代码块。

1. #ifdef

跟在 #ifdef 指示字后的代码块只有在指定宏被定义的情况下才会被继续预处理、编译。#ifndef 指示字与 #ifdef 指示字正好相反；跟在 #ifndef 指示字后的代码块只有在指定宏未被定义的情况下才会被继续预处理、编译。

如果 #ifdef 指示字后面的宏定义了，就认为条件为真，如果有 #else 指示字，那么 #ifdef 指示字与 #else 指示字之间的代码块会被继续预处理和编译，#else 指示字与 #endif 指示字之间的代码块不会被继续预处理和编译；如果没有 #else 指示字，那么 #ifdef 指示字与 #endif 指示字之间的代码块会被继续预处理和编译。

如果 #ifdef 指示字后面的宏没有定义，就认为条件为假，如果有 #else 指示字，那么 #ifdef 指示字与 #else 指示字之间的代码块不会被继续预处理和编译，#else 指示字与 #endif 指示字之间的代码块会被继续预处理和编译；如果没有 #else 指示字，那么 #ifdef 指示字与 #endif 指示字之间的代码块不会被继续预处理和编译。

#ifdef 指示字应该与另一个指示字 #endif 配对终止。如果不配对，预处理程序会发出警告信息。例如，下面的 floatarray 数组只有在定义了宏 WIN32 || WIN64 || WINNT 的情况下（Windows 系统下）才会被声明。

```
#ifdef WIN32 || WIN64 || WINNT
    float floatarray[MINTARRAY];     //宏 WIN32 || WIN64 || WINNT 定义了
                                     //(即在 Windows 系统下),才会继续预处理、编译
#else
    int intarray[100];               //非 Windows 系统下,这一条被预处理、编译
#endif                               /* (非 Windows 系统下), */
```

注意：注释、续行等 C 语言规则在预处理指示字行同样适用。

2. #ifndef

#ifndef 指示字是 #ifdef 指示字的反义词，它只在宏未被定义的条件下才编译这些条

件代码。#ifndef 指示字是#ifdef 指示字的另一种替代，两者二选一与后面的#else 指示字和#endif 指示字配套。

如果#ifndef 指示字后面的宏没有定义，就认为条件为真，如果有#else 指示字，那么 ifndef 指示字与#else 指示字之间的代码块会被继续预处理和编译，#else 指示字与#endif 指示字之间的代码块不会被继续预处理和编译；如果没有#else 指示字，那么#ifndef 指示字与#endif 指示字之间的代码块会被继续预处理和编译。#ifndef macro 与#if! defined(macro)等价。

如果#ifndef 指示字后面的宏定义了，就认为条件为假，如果有#else 指示字，那么#ifndef 指示字与#else 指示字之间的代码块不会被继续预处理和编译，#else 指示字与#endif 指示字之间的代码块会被继续预处理和编译。如果没有#else 指示字，那么#ifdef 指示字与#endif 指示字之间的代码块不会被继续预处理和编译。

#ifndef 指示字应该与另一个指示字#endif 配对终止。如果不配对，预处理程序会发出警告信息。例如，如果定义了 DESKCOLOR，变量 xarray 的类型为字符型，否则会为整型。

```
#ifndef DESKCOLOR
    int xarray;                    //宏 DESKCOLOR 没有定义,这一条被预处理、编译
#else
    char xarray;                   //宏 DESKCOLOR 定义了,这一条被预处理、编译
#endif/ * DESKCOLOR * /
```

3. #else

#else 指示字可用来提供用于继续预处理和编译的可选代码。#ifdef 指示字或#ifndef 指示字宏测试条件成立，那么#else 指示字与#endif 指示字之间的代码块就不会被继续预处理和编译。#ifdef 指示字或#ifndef 指示字宏测试条件不成立，那么#else 指示字与#endif 指示字之间的代码块就会被继续预处理和编译，示例如下所示。

```
#ifdef DESKCOLOR
#define COLOR 311                  //DESKCOLOR 定义了,这一条被预处理
#else
#define COLOR 88                   //DESKCOLOR 没有定义,这一条被预处理
#endif
```

4. #endif

#endif 指示字是#ifdef 指示字或#ifndef 指示字的配对结尾指示字。每个#ifdef 指示字或#ifndef 指示字可以没有#else 指示字，但是必须要有#endif 指示字。如果#if 指示字没有#endif 指示字配对，预处理程序会给出警告信息。

#ifdef、#ifndef…#endif 指示字是可以嵌套使用的，例如下面的嵌套结构就是 stdio.h 系统头文件中的代码。

```
#ifndef _STDIO_DEFINED
#ifdef _WIN64
  _CRTIMP FILE * __cdecl __iob_func(void);
#define _iob __iob_func()
#else
#ifdef _MSVCRT_
extern FILE _iob[];                          / *  A pointer to an array of FILE * /
```

```
#define __iob_func()(_iob)
#else
extern FILE (* __MINGW_IMP_SYMBOL(_iob))[];      /* A pointer to an array of FILE */
#define __iob_func()(* __MINGW_IMP_SYMBOL(_iob))
#define _iob __iob_func()
#endif                                           //这个对应的是#ifdef _MSVCRT_
#endif                                           //这个对应的是#ifdef _WIN64
#endif                                           //这个对应的是#ifndef _STDIO_DEFINED
```

5.11 #line 指示字

调试器运行时需要将文件名和行号与数据项和可执行代码关联起来，这样跟踪原始文件名字和行号。但是预处理程序会组合(例如#include)一些文件，因此需要预处理程序将这类定位信息插入编译程序的输出结果。编译程序在编译插入目标代码中的表时，会使用这些数字。

通常，预处理程序通过计算来确定行号，但也有可能用其他一些处理来去掉这些行号。例如，实现 SQL 语句的通常方法就是将它们写成宏，然后用特殊的处理器将这些宏扩展成具体的 SQL 函数调用。这些扩展可在很多代码行中运行，这样计算行号就很困难。预处理程序会通过在输出中插入#line 指示字进行更正，这样预处理程序就会跟踪原始源代码的行号。

下面是可用于#line 指示字的特征和规则。

(1) 为#line 指示字指定一个数字，会令预处理程序将当前行号替换为指定行号。从插入#line 指示字的行开始，以后每处理一行源代码该行号就增加 1。例如，下面的指示字设置当前行号为 137。

```
#line 137
```

(2) 为#line 指示字指定行号和文件名，会令预处理程序改变行号以及当前文件的名字。如果源代码文件#include 了很多其他文件进来，在源代码正式开始处用#line 指示字指定行号和文件名将是非常正确的决定。例如，下面的指示字会设置当前位置为文件 myheader.h 的第一行。

```
#line 1 "myheader.h"
```

(3) #line 指示字修改预定义宏__LINE__和__FILE__的内容。参看附录 C 了解这两个参数的意义。

(4) #line 指示字对由#include 指示字查找到的文件名或目录没有影响。

5.12 #pragma 指示字和_Pragma 运算符

#pragma 指示字提供一种标准方法用来让编译程序执行某些特殊操作。根据标准，编译程序可以执行#pragma 指示字希望的操作。通常#pragma 指示字书写规则如下。

```
#pragma 参数表
```

其中，参数表的各参数之间用空格隔开，常用的几种形式如下所示。

1. #pragma once

#progma once 指示字用于 C 语言头文件编译次数控制，只要在头文件的最开始加入这条#progma once 指示字就能够保证头文件被编译一次。#progma once 是和具体的编译程序相关，但现在基本上已经每个编译程序都支持这个定义了。gcc 编译程序支持#progma once 指示字和参数。

#progma once 指示字除了可以做到头文件只被编译一次之外，还可以用#if、#ifndef 指示字配合#define 指示字做到头文件只被编译一次，而且这种方法通用性更好，后面的指示字使用技巧中会有介绍。

2. #pragma message

#progma message 指示字用于在编译时输出信息，并不做其他控制动作。示例代码段如下。

```
#pragma message("this is message 马上警告")
```

3. #pragma pack(n)

#pragma pack(n)指示字用于指定编译程序内存对齐方式或取消编译程序当前内存字节对齐方式。其中 pack(n)中的 n 为 1、2、4、8、16 等数字。

```
#pragma pack()                    //取消当前对齐方式
#pragma pack(1)                   //按照 1 字节对齐方式
#pragma pack(2)                   //按照 2 字节对齐方式
#pragma pack(4)                   //按照 4 字节对齐方式
```

因为对齐方式不一样，会在计算复杂的构造类型变量占用内存大小字节数上出现一些不一致的情况，通常#pragma pack(1)总会让 sizeof 的计算结果和理论上学习的类型大小相同。例如下面的示例程序中理论上 stu1 和 stu2 都应该是 8 字节，但运行结果就出现了偏差。

```
#include <stdio.h>
struct student1
{
    char cGender;                 //1 字节
    short int nAge;               //2 字节
    char cStatus;                 //1 字节
    int nSerialNO;                //4 字节
} stu1;                           //理论上是 8 字节
struct student2
{
    char cGender;                 //1 字节
    char cStatus;                 //1 字节
    short int nAge;               //2 字节
    int nSerialNO;                //4 字节
} stu2;                           //理论上也是 8 字节
int main()
{
    printf("stu1 大小：%d 字节;stu2 大小：%d 字节\n",sizeof(stu1),sizeof(stu2));
    return 0;
}
```

一般编译程序总是以 1、2、4……方式进行内存对齐，很多都是默认以 4 字节方式对齐，这样可以提高内存读写速度。gcc 编译程序也是默认以 4 字节方式对齐，这个内存对齐方

式与特定数据类型对齐方式不同。图 5.7 是运行结果。

图 5.7 默认内存对齐方式构造类型大小运行结果

可以看出理论上应该大小相同的构造类型变量，实际用 sizeof 运算符求出来的大小却不一样。在定义结构型变量前用 #pragma pack(1)让编译程序按照 1 字节内存对齐方式对齐，这样结构中的成员变量就按单字节顺序强制连续排列了，结构大小与理论值就一样了。但是这样却会增加程序运行时读取内存周期数，让运行效率降低一点。这属于软件优化的范畴。在上面的示例程序中定义结构类型之前，用 #pragma pack(1)指示字让编译程序按照 1 字节内存对齐方式对齐，运行结果如图 5.8 所示。

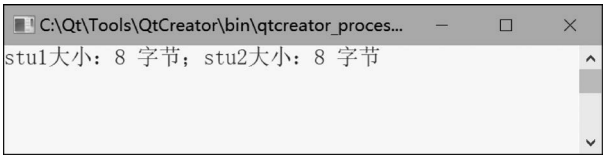

图 5.8 #pragma pack(1)内存对齐方式构造类型大小运行结果

这次求出来的内存大小就与理论计算大小一样了。

4. pragma GCC

所有 GCC 专用参数都包含两个参数：第一个为 GCC；第二个为指定的 pragma 的名字。

1) #pragma GCC dependency

#pragma GCC dependency 会测试当前文件的时间戳，对比其他文件的时间戳。如果其他文件更新一些，预处理程序就会发出警告消息。例如，下面的指示字会测试文件 myfun.obj 的时间戳，如果需要更新则会给出提示信息。

```
#pragma GCC dependency "myfun.obj"
```

如果 myfun.obj 比当前文件新，预处理程序就会产生如下消息。

```
warning: current file is older than "myfun.obj"
```

上述的 pragma 指示字中还可以加入其他文本提示信息，它会作为警告消息的一部分，示例如下所示。

```
#pragma GCC dependency "myfun.obj" Source myfun.c needs to be updated
```

它会创建下面的警告消息。

```
MulticastListener.c:26:warning: current file is older than "myfun.obj"
MulticastListener.c:26:warning:Source myfun.c needs to be updated
```

2) #pragma GCC poison

#pragma GCC poison 在每次使用指定标识符时预处理程序都会发出消息。因此可用它确保源代码中从未出现某函数被调用的情况。例如 char *gets(char *str)系统标准库函数在 C99 标准中被弃用，需要特别注意这个不安全的函数，那么下面的 pragma 指示字会让预处理程序在源代码中发现调用 gets()、getchar()函数时发出警告消息。

```
#pragma GCC poison   gets   getchar
    str = gets(str);
```

预处理程序会为该函数调用代码产生如下警告消息。

```
MulticastListener.c:38:9:attempt to use poisoned "gets"
```

3）#pragma GCC system_header

#pragma GCC system_header 指示字会让预处理程序从此行开始到文件尾的代码被看作系统头文件的一部分。编译系统头文件代码有一些不同，由于运行时库不能被改写，因此它们是严格的纯 C 语言标准格式。系统头文件编译会限制所有警告消息（除了 #warnings 指示字）。使用这种 #pragma GCC system_header 指示字方法可以让某些宏定义和扩展不会发出警告消息。

4）_Pragma

不同厂商的 C 语言预处理程序和编译程序之间通常总是会有些扩展功能上的差异，因此有些 #pragma 指示字不能作为扩展中的一部分包含进来，这就造成兼容性问题。为了解决这类 #pragma 指示字问题，C99 标准设计了预处理程序的 _Pragma 运算符用于生成内部的 #pragma 指示字。例如为了创建内部的 poison pragma，可以编写如下 C 语言代码。

```
_Pragma("GCC poison printf gets")
```

完整的示例程序如下所示。

```
#include <stdio.h>
_Pragma("GCC poison printf gets")
int main()
{
    char str[80];
    gets(str);
    printf("%s,Hello World!\n",str);
    return 0;
}
```

使用 gcc 编译时会给出错误信息提示，而且因为产生的是 error 类型信息，所以没有通过编译，也没有生成可执行程序 _Pragma.exe。编译信息如图 5.9 所示。

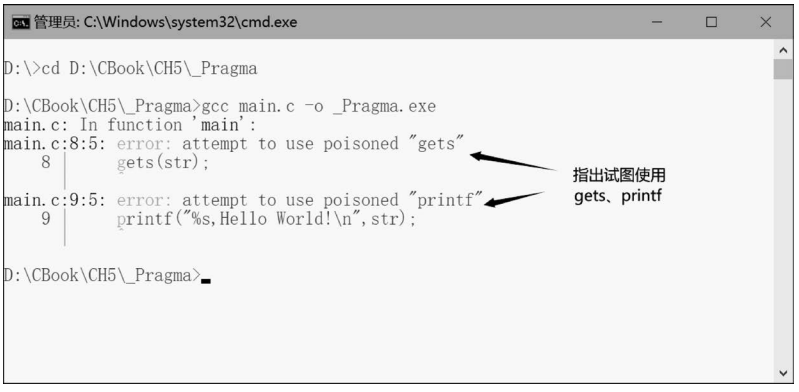

图 5.9　_Pragma 运算符效果

在使用_Pragma生成内部的#pragma指示字时，与C语言定义一样，可以用反斜线字符产生转义字符，可用这种方式插入引用的字符串来创建dependency pragma指示字。

```
_Pragma("GCC dependency \" myfun.obj \"")
```

5.13 ##连接指示字

##连接指示字用于宏内部将两个源代码权标连接成一个，可用来构造不会被解析器错误解释的名字。例如，下面示例程序中的两个宏会实现连接操作。

```
#include <stdio.h>
#define COLOR(a)  a## #CAR                    //参数连接 CAR
#define COMPOUND(a,b)  a## #b                 //两个参数拼接
#define REDCAR 100
#define BLUECAR 101
#define GRAYCAR 102
#define SILVERCAR 103
#define SMALLHOUSE 500
#define FARMHOUSE 501
#define RANCHHOUSE 502
#define TOWNHOUSE 503
int main()
{
    int nCarcolor,nHousetype;
    nCarcolor = COLOR(RED);
    printf("%d \n",nCarcolor);
    nCarcolor = COLOR(BLUE);
    printf("%d \n",nCarcolor);
    nCarcolor = COLOR(SILVER);
    printf("%d \n",nCarcolor);
    nHousetype = COMPOUND(RANCH,HOUSE);
    printf("%d \n",nHousetype);
    nHousetype = COMPOUND(TOWN,HOUSE);
    printf("%d \n",nHousetype);
    printf("Hello World!\n");
    return 0;
}
```

使用命令行cpp -E -dD main.c -o precompiler.txt预处理后，查看precompiler.txt文件，主函数main()内带参数的宏被预处理成拼接后的宏标识符并且被宏值替换。预处理后主函数main()内部分代码如下。

```
int main()
{
    int nCarcolor,nHousetype;
    nCarcolor = 100;
    printf("%d \n",nCarcolor);
    nCarcolor = 101;
    printf("%d \n",nCarcolor);
    nCarcolor = 103;
    printf("%d \n",nCarcolor);
    nHousetype = 502;
```

```
    printf("%d\n",nHousetype);
    nHousetype = 503;
    printf("%d\n",nHousetype);
    printf("Hello World!\n");
    return 0;
}
```

5.14 指示字使用技巧

本章已经介绍了很多C语言预处理程序的指示字,有些是通用的,有些则是GCC专有的指示字。虽然有示例程序但是多是讲这些指示字怎么用,下面就介绍一些实际编程中的指示字使用技巧。

5.14.1 头文件包含检测

C语言源代码很多都以#include开头,包含很多系统头文件,还有自己定义的头文件。由于头文件还可能会包含其他头文件,因此源代码程序很有可能多次包含同样的头文件。这会导致出错信息,因为一些已经定义了的宏可能因为头文件多次包含会被再次定义。为防止发生这样的情况,使用“5.12 #pragma指示字和_Pragma运算符”里的#progma once指示字就能够保证头文件只被编译一次。不过为了更通用,还可以使用如下方法编写头文件。

```
/* myheader.h */
#ifndef  MYHEADER_H
#define  MYHEADER_H
/* The body of the header file */
#endif /* MYHEADER_H */
```

在这个例子中,通过#ifndef指示字和#define宏定义来检测自身是否已经被包含。头文件第一行测试是否已经定义了宏MYHEADER_H,如果已经定义了,就会跳过整个头文件。如果MYHEADER_H未被定义,立即定义它并预处理头文件且编译头文件。

系统头文件都使用了这种技术。它们定义的宏名字都由下画线字符开始,以防止和用户定义的其他宏名字冲突。定义宏名字的规则是全部使用大写字母,包含文件的名字。例如stdio.h文件里定义的方式就是。

```
#ifndef _INC_STDIO                    //是不是想起了 include stdio
#define _INC_STDIO                    //定义_INC_STDIO
//头文件体
#endif                                //整个头文件结束
```

GCC预处理程序能识别这种结构,并保存头文件使用的过程。通过这种方式识别头文件名,并在已经包含该文件的情况下根本不再读文件,就能够优化头文件的处理过程。

5.14.2 使用预定义宏的定位信息

GCC预定义了大量的宏。这些预定义宏依赖于被编译的语言、指定的命令行选项、使用的平台、目标平台、使用的编译程序的版本以及设置的环境变量。可以使用命令行方式执行预处理程序,使用-dM选项来查看完整列表,命令如下。

```
cpp -E -dM myprog.c | sort | more
```

由该命令输出的包含#define指示字的宏列表是处理特定输入源文件和所有包含头文件之后在预处理程序中定义的宏。

预定义的宏可用来自动构造出错消息，它包含发生错误的位置的详细信息。附录C中列出了常用的预定义宏，预定义的宏__FILE__、__LINE__和__func__包含这些信息，但它们必须用于创建信息的那个时刻点上。由于在预处理程序处理源代码时，这些宏的值是在不断重新定义并变化着。因此，如果写一个函数包含这些预定义宏，就会在函数被调用时输出出错消息内容及其定位信息。

完美的解决方法就是定义一个包含它们的宏。当预处理程序扩展宏时，它们都处于正确位置并含有正确信息。下面是一个定义出错宏的例子，这个宏将定位信息和错误消息写到标准错误输出终端中。

```
#define msg(str) fprintf(stderr,"File: %s Line: %d Function: %s %s\n",\
__FILE__,__LINE__,__func__,str);
```

为从代码中的任意位置激活该宏，可以将描述错误的字符串作为宏参数使用宏，例如：

```
msg("There is an error here.");
```

这样做的另一个优势是处理错误条件的方法可以简单通过改变宏来实现。它可以转换为抛出异常或将错误信息记录到文件中。由本例产生的消息如下所示。

```
File:myfunc.c Line:822 Function: hashcompress There is an error here.
```

5.14.3 源代码安全去除与恢复

在调试代码的过程中，经常会尝试暂时去掉(或隐掉)一些代码看看程序运行会怎么样，并且会在以后需要的时候恢复这些代码。有人可能认为用/*…*/注释包围起来就可以了，但这会引起问题，因为C语言中的/*…*/注释不能嵌套，而在需要被隐掉的代码中可能含有大量注释。一种清晰而安全的方式就是通过使用预处理程序的#if指示字来省略代码，如下所示。

```
#if 0       //使用0值,直接让#if和#endif之间的代码块全部失效
/* 代码块全部失效 */
#endif
```

这不仅能够清晰地处理/*…*/注释，而且很明显就是要有意地去掉这部分代码。