第6章 编译预处理

华中科技大学计算机学院
甘早斌

主要内容

- 6.1文件包含#include
- 6.2 宏定义#define
- 6.3 取消宏定义#undef
- 6.4 条件编译
- 6.5 assert断言和静态断言**
- 6.6 _func_预定义标识符**
- 6.7 _Pragma预处理操作符**

编译预处理

- C编译程序的预处理功能是C区别于其他高级程序设计语言的特征之一。C源程序中以#开头、以换行符结尾的行称为编译预处理指令。

```
#define   PI  3.14159
#include<stdio.h>
```

- 编译预处理指令不是C语言的语法成分；在对源程序进行编译之前先进行处理。
- 常用编译预处理包括：
  - (1)宏定义；
  - (2)文件嵌入；
  - (3)条件编译。

编译预处理

- 编译预处理：在对源程序进行编译之前所作的工作，它由预处理程序负责完成。编译时，系统将自动引用预处理程序对源程序中的预处理指令进行处理。
- 预处理指令：以“#”号开始的指令。

- 程序中合理地使用预处理功能，可以：
  - 使程序便于阅读、修改、移植和调试；
  - 有利于模块化程序设计、代码重用。

6.1 文件包含#include

- #include 就是一个预处理指令，它命令预处理器进行的预处理是：把特定的头文件包含到我们的源代码里。
- 用指定文件内容取代该预处理指令行，有2种形式：
  - (1) #include <文件名>
    - 在指定的标准目录下寻找被包含文件
  - (2) #include "文件名"
    - 首先在用户当前目录中寻找被包含文件
    - 若找不到，再在指定的标准目录下寻找
    - 标准目录是与系统相关的，可由用户在设置环境时设置
  - 一般来说，包含系统标准的头文件时用尖括号；而包含自己的头文件时用双引号

- 例如语句

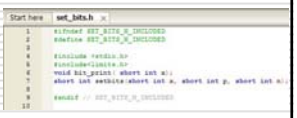
```
#include <stdio.h>
```

  - 其作用是将头文件 stdio.h 里的内容包含进我们的程序里；产生的结果是将 stdio.h 里的内容一字不漏地插入 #include <stdio.h> 出现的地方，并且删除 #include <stdio.h>；
  - 实质上是，用 stdio.h 里的内容替换#include <stdio.h>。
- 以上过程是在预处理阶段完成的。
- C 语言的这种包含文件的机制为多个程序**共享**相同信息提供了极大的方便。

- 头文件中包含着编译器进行编译时所需的信息。头文件中可能指明了函数名和函数调用方法，但是这些函数的实现代码并不在头文件中，而是在预先编译好了的库文件里。
- **链接器**负责在库文件中寻找我们的程序所需的代码，并且把那些代码和我们写的程序进行链接，从而将我的写的程序链接成可执行文件。
- 一句话，头文件用于指导编译器正确地将我们写的源程序编译成可执行文件。

例子

- 重写P53页2.13题
  - 将以二进制数形式显示一个整数的函数作为一个源文件bit_print.cpp;
  - 将一个整数x从第p位开始的向右n位(p从右至左编号为0-15)翻转，其余各位保持不变，完成此功能的函数作为一个源文件set_bits.cpp;
  - 主函数文件名为2.13.cpp;
  - 头文件名为set_bits.h;
  - 工程文件名为set_bits.dsp。



- 一个大的程序可以分为多个模块，由多个程序员分别编程
- 有些公用的符号常量定义、数据类型定义、函数说明或预处理指令等可单独组成一个文件
- 在其他文件的开头用文件包含指令包含该文件即可使用

∴ 在程序设计中，文件包含是很有用的，可避免在每个文件开头都去书写那些公用量，从而节省时间，减少出错

6.2 宏定义#define

- 用一个标识符来表示一个字符串，该标识符称为宏。
  - 宏名：被定义的标识符。
  - 宏代换（宏展开）：在编译预处理时，用字符串去取代程序中的宏名

预处理前

```
#define  M  (y*y+3*y )
int main(void)
{  int  s,y;
   printf("Input a number: ");
   scanf("%d",&y);
   s=3*M+4*M+y*M;
   printf("s=%d\n",s);
   return 0;
}
```

预处理后

```
int main(void)
{  int  s,y;
   printf("Input a number: ");
   scanf("%d",&y);
   s=3* (y*y+3*y ) +4* (y*y+3*y )
       +y* (y*y+3*y );
   printf("s=%d\n",s);
   return 0;
}
```


6.2.1 无参数宏定义

- 无参数的宏名后不带参数，其定义的一般形式为：
  - #define 标识符 字符串
- 标识符为所定义的宏名，字符串可以是常数、表达式、格式串等任意字符，预处理程序只是简单地以该字符串取代宏名，对它不做任何检查。
- 如果有错误，只能在编译宏展开后的源程序时发现。如果字符串太长需要分成若干行，则只需在尚待延续的行后加上一个续行符（即反斜线\）即可。

- 例：

```
#define  PI   3.14159
printf("PI=%d",PI);  ➔ printf("PI=%d",3.14159);
```

- 注意：字符串常数中出现的与宏名相同的字符串不在替换之列。例：

```
#define  YES  1
printf(" YES");  输出 YES，而不是1
```

6.2.2 带参数的宏定义

#define 标识符(标识符，标识符，…，标识符) 字符串

宏名　　形式参数

宏调用：给出实参

宏展开：（1）用字符串替换宏

（2）用实参去替换形参

- 例 定义计算x^2的宏

  #define square (x) ((x)*(x))
  - 宏调用：square (a+1)
  - 宏展开：((a+1) * (a+1))
  - 宏调用：square (square (a))
  - 宏展开：((((a)*(a))) * (((a)*(a))))

- 为什么要这么多的括号?
  - 考虑：#define SQ(x) x*x
  - 宏调用：SQ(a+b)
  - 宏展开：a+b*a+b /* 与(a+b)*(a+b)不同 */
  - 再考虑：#define SQ(x) (x)*(x)
  - 宏调用：27/SQ(3)
  - 宏展开：27/(3)*(3) /* 值27, 与 27/((3)*(3)) 不同 */
  - 定义带参数的宏时，为了保证计算次序的正确性，表达式中的每个参数用括号括起来，整个表达式也用括号括起来。

- 注意：宏名和与左括号之间不能有空格

  #define SQ (x) ((x)*(x))
  - 被认为是无参宏定义。宏名代表字符串" (x) ((x)*(x))"
  - 宏调用：SQ(3)
  - 宏展开：(x) ((x)*(x)) (3) /*显然错误的*/
- 在现代编程技术中，认为宏本质是不安全的。因为预处理器执行宏时，不做任何检查，不会发现错误。例如：

  #define SQ(x) ((x)*(x))
  - 宏调用：SQ(++a)
  - 宏展开：((++a)*(++a)) /*a加2次，与原意不符。而如是函数调用SQ(++a),将不会有问题,将实参表达式++a求值后传递给函数*/

- 带参的宏虽被认为不安全，但还是很有价值
  - 宏节省了函数调用的开销，程序运行速度更快，形式参数不分配内存单元，不必作类型说明。但是，宏展开后使源程序增长。
  - 宏比较适合于经常使用的简短表达式，以及小的可重复的代码段；当任务比较复杂，需要多行代码才能实现时，或者要求程序越小越好时，就应该使用函数。

6.2.3 空宏参数**

- C99允许宏调用中任意或所有参数为空。例如：

  #define ADD(x,y) (x+y) /* 定义两数相加的宏 */

  x = ADD(2,3);

  y = ADD(,3); /* 第1个参数为空*/
  - 预处理后变为：

    x = (2+3); /* x的值为5 */

    y = (+3); /* y的值为3 */

6.2.4 可变参数宏定义**

- C99增加了可变参数宏（variadic macros），允许像下面这样定义可变参数宏：

  #define debug(format, ...) printf (format, __VA_ARGS__)

- debug是一个可变参数宏，format是宏的一个参数，省略号代表一个能够改变的参数表，在每次被调用时，“…”被表示成零个或多个参数。
- 内建的预处理器标识符__VA_ARGS__ 用来把“…”部分传递给宏。当宏的调用展开时，实参就取代__VA_ARGS__。例如，宏调用

  debug("x= %d\n，y=%d\n",10,20); /* 输出 x=10，y=20 */

- 会被展开成：

  printf("x= %d\n，y=%d\n",10,20);

- 例6.2 用C99的可变参数宏，打印调试信息
  - 在编写代码的过程中，为了调试程序，经常会输出一些调试信息到屏幕上，随着项目的调试，输出的信息可能会越来越多，信息的输出一般要调用printf等函数。但是，当调试完后，又需要手工将这些地方删除或者注释掉。这样做工作量比较大，很麻烦。
  - 如何方便地处理这些用于输出调试信息的语句？用C99的可变参数宏，可方便地输出调试信息。
  - 源程序\ex6_2.c
  - 调试阶段定义DEBUG宏，在需要输出调试信息的地方用宏msg，调试成功后，软件正式发行时，只需将第2行的#define指令删除或注释掉即可，非常方便。

6.2.5 通用类型宏**

- 通用类型宏或者泛型宏（type-generic macros）是一种编译期技术，它允许开发人员根据宏的某个参数的类型来确定生成的内容。
- 早在C99时就有了通用类型宏的概念，只不过当时没有对它进行标准化。C99中引用了头文件<tgmath.h>，给开发人员提供大量初等数学函数接口。
- 在C99中，程序员可以用不同的类型来调用 sin函数，比如：sin(1)，实际上调用sin(1.0)；sin(1.0F)，实际上调用sinf(1.0F)；sin(1.0L)，实际上调用sinl(1.0L)。实际上，sin这个接口是一个宏，它会根据传来的实际参数类型展开成特定的函数。 sin就是一个通用类型宏。

- 关键字_Generic
  - C11中引入了新关键字_Generic来实现通用类型宏，它根据第一个参数的类型和后面的类型-表达式关联来实现编译期的替换。利用_Generic，C99的sin可定义如下：

    #define sin(x) _Generic ((x), long double: sinl, double: sinf, defalt: sin) (x)
  - _Generic对第一个参数进行类型判断，然后根据从第二参数开始的类型-表达式关联表来进行编译期替换。如果x为long double类型，那么_Generic((x), …)的结果为sinl，如果x为double类型，那么_Generic((x),…)的结果为sinf，否则结果为sin。可见，根据x的类型，宏sin(x)转换为sinl(x), sinf(x)或sin(x)。

- 例6.3 用_Generic，编写求和的通用类型宏sum

```
1.  int sumi(int *arr, int cnt)          /* 整数求和 */
2.  { int sum = 0;
3.    int i;
4.    for(i = 0; i < cnt; ++i)  sum += arr[i];
5.    return sum;
6.  }
7.  double sumf(double *arr, int cnt) /* 浮点数求和 */
8.  { double sum = 0.0;
9.    int i;
10.   for(i = 0; i < cnt; ++i)  sum += arr[i]
11.   return sum;
12. }
13. /* 通用类型宏sum，它会根据传递的实际类型来决定最终调用的函数 */
14. #define sum(_arr, _cnt) _Generic(_arr[0],int: sumi, defualt: sumf)(_arr, _cnt)
```


6.3 取消宏定义#undef

- 宏名作用域从其定义开始直到该宏定义指令所在文件结束。如要终止宏名作用域,可使用#undef指令,其形式为：

  #undef 标识符
  - 标识符是由#define指令定义过的宏名，它使得前面的宏定义被取消。例如：

    #undef PI
  - 前面PI被定义 #define PI 3.1415926，那么#undef指令之后PI失去定义，或直到PI被再次定义为止。
  - 如果前面没有#define PI 3.1415926，则#undef 指令不起作用。

- 何时使用#undef指令?
  - 防止宏名的冲突

```
#include "everything.h"
#undef  SIZE   /*everything.h中定义了SIZE，就取消它；
                否则该指令不起作用*/
#define  SIZE  100
```

  - 保证调用的是一个实际函数而不是宏

```
#undef  getchar
int  getchar(void) {…}
```


6.4 条件编译*

- 6.4.1 #if、#ifdef和 #ifndef指令
- 6.4.2 defined运算符
- 6.4.3 条件编译的应用

6.4.1 #if、#ifdef和 #ifndef指令

- 预处理程序提供了条件编译指令，用于在预处理中进行条件控制，根据所求条件的值有选择地包含不同的程序部分，因而产生不同的目标代码。
- 这对于程序的移植和调试是很有用的。对源程序的各部分有选择地进行编译称为条件编译。
- 条件编译有三种形式，如表6.1所示(P131)，每种形式的控制流与if语句的控制流类似。
- “程序段”中可以包含#include和#define预处理行，常量表达式必须是整型的并且不能含有sizeof与强制类型转换运算符或枚举常量。

- 例 利用R计算圆或正方形的面积

预处理前

```
1.  #define  R
2.  int  main(void)
3.  {   float  r，s;
4.      printf (“input a number: ”);
5.      scanf(“%f”，&r);
6.      #ifdef  R
7.          s=3.14159*r*r;
8.          printf(“%f\n”，s);
9.      #else
10.         s=r*r;
11.         printf(“%f\n”，s);
12.     #endif
13.     return 0;
14. }
```

预处理后

```
1.  int  main(void)
2.  {
3.      float  c, r, s;
4.      printf (“input a number: ”);
5.      scanf(“%f”，&r);
6.      s=3.14159*r*r;
7.      printf(“%f\n”,s);
8.      return 0;
9.  }
```

生成的目标程序较短

6.4.2 defined运算符

- defined是预处理运算符，其形式为:

  defined（标识符）或 defined 标识符
- 它用来判断标识符是否被#define定义了，如被定义，值为1，否则为0
- defined运算符，可以将第2种和第3种形式的条件编译指令改用第1种形式的条件编译指令。
- 例如，例6.4中的#ifdef可为

```
#if  defined(R)
```


- 用该运算符可以写比较复杂的条件编译指令。#ifdef只能判断一个宏，如果条件比较复杂实现起来会比较烦锁，而用#if defined( )就比较方便。
- 有两个宏MACRO_1和MACRO_2，只有两个宏都定义过才会编译程序段A，可通过如下方式实现:

```
#if defined(MACRO_1) && defined(MACRO_2)
    程序段A
#endif
```

6.4.3 条件编译的应用

- (1) 采用条件编译，避免多次包含同一个头文件。例6.5
  - 为了避免一个头文件被多次包含，可在头文件的最前面两行和最后一行加上预编译指令，让头文件在被多个源文件引用时不会多次编译。

```
#ifndef _NAME_H
#define _NAME_H          /* 定义头文件的标识符 */
……                                  /* 头文件的内容 */
#endif
```

  - 其中，NAME是头文件的名字。比如头文件为myFile.h，则其标识符可为_MYFILE_H。
  - 在创建一个头文件时，用#define指令为它定义一个唯一的标识符。通过#ifndef指令检查这个标识符是否已被定义，如果已被定义，则说明该头文件已经被包含了，就不要再次包含该头文件，#ifndef就帮助编译器跳过直到#endif的所有文本；反之，则定义这个标识符，以避免以后再次包含该头文件。

- 例6.6 条件编译允许有选择地编译程序的某些部分，可以将程序的特殊性能纳入不同版本。
  - 例如对于不同语言版本中的某个应用程序，需要改变货币的显示，可以使用以下条件编译，使用预定义常数 ACTIVE_COUNTRY 的值来决定货币符号。

```
#define  US                    0
#define  ENGLAND               1
#define  FRANCE                2
#define  ACTIVE_COUNTRY        US
#if  ACTIVE_COUNTRY = = US
   char currency[ ]= "dollar";    /* 美元 */
#elif ACTIVE_COUNTRY= =ENGLAND
   char currency[ ]= "pound";     /* 英镑 */
#else
   char currency[ ]= "franc";     /* 法郎 */
#endif
```

  - #elif指令的意义与else if 相同，它形成一个if-else-if阶梯状语句，可进行多种编译选择。每个 #elif 后跟一个常量表达式。如果表达式为非0，则编译其后的程序段，不再对其他#elif表达式进行测试。否则，顺序测试下一个条件。

- (2) 调试程序时临时忽一些代码
  - 在程序开发过程中，程序员经常需要临时忽略或封闭一些代码，从而防止编译器编译这些代码。要做到这一点，可以把代码放在注释中。
  - 如果代码中也含有注释，这个方法就会导致语法错误。使用条件编译能解决这个问题：
    - #if 0
    - 不编译的代码
    - #endif
  - 要让编译器编译这段代码，把原来的0改为1就可以了。

- (3) 调试程序时跟踪程序的执行
  - 源程序调试中，常常要跟踪程序的执行情况，可在程序中加一些输出信息的语句，通过这些输出信息来跟踪判断程序是否有错误。
  - 在调试结束后，需要把调试时新增的输出语句删除掉。然而手工删除既不方便，也易出错。使用条件编译方便得多，把这些新增的调试语句放在条件编译指令之间，在调试时编译这些语句。如：

```
#ifdef  DEBUG
   printf("Variable x=%d\n",x);
#endif
```

  - 在调试程序时，在前面加#define DEBUG，就编译printf语句，输出供判断参考的x值。完成调试后，从程序中去掉#define指令，编译就忽略为调试而插入的printf语句，相当于它被“自动”删除了。

6.5 assert断言和静态断言**

- 使用断言可以创建更稳定、品质更好且不易于出错的代码。断言用于在代码中捕捉一些假设，当假设不成立时中断当前操作，可以将断言看作是异常处理的一种高级形式。
- assert断言是动态断言，只能在程序运行出现错误时做出判断。C11增加了静态断言，它可以在编译时就对程序的错误做出判断。

6.5.1 assert断言

- 在头文件assert.h中，用来测试表达式的值是否符合要求，其形式如下：

```
assert(condition)
```

- 如果condition值非0，程序继续执行下一个语句。如果condition值0，就输出错误信息，并通过调用实用库中的函数abort终止程序的执行。

- 用assert宏判断数据是否合法

assert (x<=10);

- 如果x大于10，就会输出如下包含行号和文件名的错误信息并中断执行：

Assertion failed:x<= 0,file test.c,line 12

- 对于大多数编译器来说，在头文件assert.h的assert宏定义中，如果定义了符号常量NDEBUG，其后的assert将被忽略。因此，如果不再需要assert，那么可把代码行

#define NDEBUG

- 插入到程序中，而无需手工删除assert。

6.5.2 静态断言

- assert宏只能在程序运行出现错误时进行退出操作并产生调试信息，而静态断言（Static assertions）可用于在编译时进行检查，不会产生任何运行时的额外开销（包括时间和空间）。
- 在C11标准中，从语言层面加入了对静态断言的支持，引入了新的关键字_Static_assert来表示静态断言，断言失败会产生有意义的且充分的诊断信息。

- 关键字_Static_assert
  - _Static_assert (constant-expression, string-literal);
  - 其中，第一个参数constant-expression必须是一个编译时可知的整型常量表达式，如果用一个变量作为第一个参数会遇到编译错误；第二个参数string-literal是在断言失败时输出的提示信息（即字符串）。
  - 当constant-expression的布尔值为true时，该静态断言声明不会产生任何影响；否则，编译器将给出错误诊断信息string-literal。例如：

_Static_assert(sizeof(int) == 8, "A 64-bit machine needed!");

  - 在32位机上编译这条语句时，就会输出如下诊断信息：

static assertion failed: "A 64-bit machine needed!"

  - 在头文件assert.h中，定义static_assert宏为关键字_Static_assert的同义词。

6.6 _func_预定义标识符**

- C99标准中引入了预定义标识符（predefined identifier）的概念，并定义了一个预定义标识符_func_。它的性质和关键字相似，尽管它本身并不是关键字。_func_定义为字符数组，用于指出_func_所在的函数名，类似于字符串赋值。例如：

```
1. #include <stdio.h>
2. void myfunc(void)
3. {
4.   printf("%s\n",__func_);   /* ... */
5. }
```

- 每次调用函数myfunc时，都输出：myfunc。

6.7 _Pragma预处理操作符**

- C99标准中增加了_Pragma操作符，以便更灵活地利用pragma工具，其使用形式如下：

_ Pragma(字符串常量)

- 它把字符串常量的内容（把字符串常量内部的\"替换为"，把\\替换为\之后）看成是#pragma指令中所出现的预处理器标记。例如：

_Pragma("OPTIMIZE OFF") /* 注意：后面是没有分号的 */

  - 与

#pragma OPTIMIZE OFF /* 指示编译器停止优化代码 */

  - 是一样的。
  - 与#pragma相比，_Pragma的优势在于：_Pragma可以用于宏定义。

本章小结

- 本章集中讨论了与预处理程序有关的问题。编译前的预处理功能是C语言区别于其他高级语言的一个重要特征之一，所有的预处理指令都是以“#”开头。
- 最常用的预处理功能有三种：#include、#define和条件编译。#include把指定的文件包含到程序中。#define用来定义宏，有带参数和不带参数的宏，使用宏可以减少程序的执行时间。条件编译使程序员能够控制预处理指令的执行和程序代码的编译。
- 头文件assert.h中的宏assert用来测试表达式的值，有助于表达式的值满足要求，保证程序的正确性。使用assert是一种良好的编程方法。

Assignments:

- 必做题：
  - 6.1，6.2，6.3，6.4，6.5，6.6
- 我建议：
  - 后面习题，每题都做，并且搞懂！