5.8 存储类别和作用域

在C程序中，程序对象都有自己的属性。

例如，变量有变量的名字、类型和值等；

函数有返回值类型、函数形参个数、各形参的类型等。

程序中的每一个标识符还有其它属性，包括存储类别、生存期、作用域等。

●变量的作用域

在讨论函数的参数传递时曾提到，形参变量只有在被调用时才分配内存单元，在调用结束时，即刻释放所分配的内存单元。这一点说明形参变量只有在函数内才有效。这种变量有效性的范围称作变量的作用域。不仅对于形参变量，C语言中所有的变量都有它的作用域。变量说明的方式不同，其作用域也不相同。变量只能在它的作用域内使用，即变量在它的作用域外不能被引用。在C语言中，按变量的作用域范围可分为两种，即局部变量（Local Variable）和全局变量（Global Variable）。

♦局部变量

在一个函数内部定义的变量被称作局部变量，也称为内部变量。局部变量有以下特点：

- ☑ 不同函数中可以使用相同名字的变量，它们代表不同的对象，互不干扰。
- ☑ 函数的形参也是局部变量，只在所在函数中有效。
- ☑ 在函数内部复合语句中定义的变量，只在该复合语句中有效，出了复合语句就无效。

对于局部变量，最重要的是要了解：

它们仅存在于被定义的当前执行代码块中，即局部变量在进入模块时生成，在退出模块时消亡。

例如，对于如下的3个函数main、f1和f2，其变量的作用域如下：

```
void main( )
{
   int m, n, s, t;
   …
}
flaot f1(float x, int y, char z)
{
   int m, k;
   …
}
int f2(int x1, char chx)
{
   char ch_0, ch_1;
   …
}
```

m、n、s、t只在main函数内有效

m、k只在f1函数内有效

ch_0, ch_1只在f2函数内有效

关于3个函数的说明：

（1）主函数中定义的变量也只能在主函数中使用，不能在其它函数中使用。因为主函数也是一个函数，它与其它函数是平行关系。这一点应予以注意。

（2）形参变量也是局部变量，作用范围在定义它的函数内，所以在定义形参时不能和函数体内的变量重名。

（3）允许在不同的函数中使用相同的变量名，它们代表不同的对象，分配不同的单元，互不干扰，也不会发生混淆。如在前例中，函数main和f1都有局部变量m，是完全允许的，他们占用不同的内存单元。

（4）在函数内部的复合语句中也可定义变量，这些局部变量的作用域只在复合语句范围内，离开复合语句则被释放。在复合语句中根据需要定义变量，可以提高内存的利用率。例如，对于如下的函数fun，其变量的作用域如下：

```
int fun(int x1)
{
   int a,b;
   …
  {
   int k;
   k=(a+b)*x1;
    …
  }
  …
}
```

变量k的作用范围

变量a,b的作用范围

♦全局变量

与局部变量不同，全局变量贯穿整个程序，并且可被任何一个模块使用。全局变量有以下特点：

- ☑ 全局变量定义在函数之外。
- ☑ 全局变量的有效范围是从定义的位置开始到整个程序的结束。
- ☑ 局部变量可以和全局变量同名，若同名，则在局部变量的作用域内，全局变量被"屏蔽"（即不起作用）。为了程序的可读性，建议读者尽量避免全局变量和局部变量的同名。
- ☑ 全局变量的作用降低了函数的通用性，这是因为采用全局变量的作用是增加了函数间数据联系的渠道，使得函数依赖这些变量，因而使得函数的独立性降低。从模块化程序设计的观点来看是不利的，模块化程序设计要求模块的"内聚性"强，与其它模块的"耦合性"弱。因此，尽量不要使用全局变量，除非不得已。（建议尽量少用或不用。）

全局变量引用示例:

```
int p=1,q=5;  /*p,q是全局变量*/
int f1(int k)
{
      int b ,c,p;//全局变量的p被"屏蔽"
       . . .
}
int  c1,c2; /*c1,c2是全局变量*/
int f 2(int x,int y)
{int i,j;
 ...}
主要部份 ()
{int i,j;
 ...
 }
```

c1,c2 有效

p,q 有效

7

●变量的存储类别

♦变量的存储方式:

从变量值存在的时间（或称变量的生存周期）角度，变量又可分为静态存储方式和动态存储方式。在计算机内存中，可供用户使用的存储空间分为3 个部分：程序区、静态存储区和动态存储区，如图 所示。

程序区
静态存储区
动态存储区

其中，程序代码存放在程序区，一般由操作系统控制；程序中运行的中间结果和最终结果数据存放在静态存储区或动态存储区。

静态存储方式:是指在程序运行期间分配固定的存储空间的方式。全局变量、静态变量和字符串常量的存储就采用这种方式。

8

在程序开始时给全局变量分配存储单元，程序执行完毕才释放。也就是说，在程序执行过程中，全局变量占据固定的存储单元，而不是动态地进行分配和释放。对于静态变量，不管是全局变量还是局部变量，在整个程序执行期间都不释放所占的存储空间。

动态存储方式:是指在程序运行期间根据需要进行动态的分配存储空间的方式。形参、自动变量（auto）、函数调用时的现场保护和返回地址等的存储就采用这种方式。由于是动态分配存储单元，且使用完立即释放，所以，如果在一个程序中两次调用同一个函数，分配给函数中局部变量等数据的存储空间地址可能是不相同的。在C 语言中，每一个变量和函数具有两个属性：数据类型和数据的存储类别。在前面的章节中，定义变量时，形式上只是声明变量的数据类型。作为定义变量的完全形式，还应该包括存储类别。

9

变量定义的一般形式为：

[存储类型] 数据类型 变量标识符表；

- ☑ 变量的数据类型规定了变量的存储空间大小和取值范围；（已介绍）
- ☑ 变量的存储类型规定了变量的生存期和作用域。变量的存储类型有4个，分别是自动型、寄存器型、外部型和静态型，其说明符分别是auto、register、extern和static。
  - auto 和register 说明的变量属于动态存储方式
  - static和extern 说明的变量属于静态存储方式。

例如： auto int a,b,c;

表示定义了3 个自动存储类别的整型变量a、b、c，它们属于动态存储方式。

10

♦自动变量

C 语言规定，函数中的局部变量如不专门声明为static存储类别，默认的都是自动类别，这些变量存放在动态存储区中。这种存储类型是C程序中使用最广泛的类型之一，前面各章节中所定义的局部变量都属于这种存储类别。在调用该函数时系统会给它们动态分配存储空间，在函数调用结束时就自动释放这些存储空间。因此，这类局部变量称为自动变量，属于动态存储方式。

自动变量用关键字auto 作存储类别的声明。例如：

```
void main( )
{
   auto int x=1,y=2;/*定义自动类别的整型变量x 和y*/
   . . .
}
```

实际上，关键字auto可以省略，auto不写则隐含为"自动存储类别"。

如，以下两个定义形式完全等价： auto int x=1,y=2; ⟺ int x=1,y=2;

11

例:auto型变量作用域示例。

```
#include <stdio.h
void main( )
{
   [auto] int x=1,y=2;
   {
      [auto] int  x = 10;
      x=x+10;
      printf("x=%d ",x);
      printf("y=%d ",y);
   }
      x=x+9;
      printf("x=%d ",x);
}
```

复合语句中变量x的生存期和作用域

主函数中变量x，y的生存期和作用域

外层模块变量x的作用域被内层模块同名变量的作用域屏蔽

程序运行结果为：x=20，y=2，x=10

12

♦寄存器变量

一般情况下，变量的值是放在内存中的，当程序用到哪一个变量的值时，由控制器发出指令将内存中该变量的值送到运算器中。经过运算后，如需存放，再从运算器将数据送到内存存放。但有些变量使用频繁，为了减少存取所花的时间，提高执行效率，C 语言允许将局部变量的值放在运算器中的寄存器中，需要时，直接从寄存器中取出参加运算，提高执行效率，这种变量叫"寄存器变量"。C 程序函数中的局部变量可以声明为寄存器存储类别，由关键字"register"说明。这类局部变量称为寄存器型变量，属于动态存储方式。例如：用关键字register说明：

```
register int i, j;
```

表示定义了寄存器型的整型变量i 和j。//但不常用

说明：

（1）只有局部自动变量和形参可以声明为寄存器变量，其它变量则不能。

（2）CPU 中的寄存器数目是有限的，不能定义任意多的寄存器型变量。

一般只有那些使用频率非常高的变量才需声明为register 变量。

为提高处理速度,将变量的值不存入内存，而只需保存在CPU内的寄存器中。由于CPU内的寄存器数量是有限的，不可能为某个变量长期占用。因此，一些操作系统对寄存器的使用做了数量的限制。或多或少，或根本不提供，用自动变量来替代。这种变量称为"寄存器变量"，

例如下面的程序段：

```
int  i, sum=0;
for(i=1;i<=1000000;i++)
        sum=sum+i;
```

可改成：

```
register i, sum=0;
for(i=1;i<=1000000;i++)
        sum=sum+i;
```

说明：当今的优化编译系统能够识别使用频繁的变量，从而自动地将这些变量放在寄存器中，而不需要程序设计者指定。因此在实际上用register声明变量是不必要的。读者对它有一定了解即可。

结构化程序设计的原则之一是代码和数据的分离。C语言是通过局部变量和函数的使用来实现这一分离的。下面用两种方法编制计算两个整数乘积的简单函数mul ( )。

通用的

```
mul (int x , int y )
{
   return ( x * y ) ;
}
```

专用的

```
int x,y;
mul ( x , y )
{
    return ( x * y ) ;
}
```

两个函数都是返回变量x和y的积，通用的或称为参数化版本可用于任意两整数之积，而专用的版本仅能计算全局变量x和y的乘积。

♦静态变量

函数中的局部变量也可以声明为静态存储类别，由关键字"static"说明。这类局部变量称为静态变量，属于静态存储方式。

有时，人们希望一个局部变量在函数结束后系统分配给它的存储空间不被释放，且存放在其中的值仍然保留，这样，在下一次调用此函数时，就可以直接利用已有值。静态变量可以满足这一要求。静态变量分静态局部变量和静态全局变量。

(1) 静态局部变量

静态局部变量是定义在函数体的复合语句中，用关键字"static"进行标识的变量。

静态局部变量定义的一般形式为：

```
static  数据类型  变量名表;
```

生存期：从变量定义开始，直到程序运行结束。

作用域：在所定义的函数或复合语句中有效。

因此静态局部变量是一种具有全局寿命、局部可见的变量。

例：局部静态变量

```
int p(int x)
{  auto int y=1;
   static int z=2;
   y++;
   z++;
   return x+y+z;
}

void main()
{ printf("% d",p(3)) ;
  printf("% d",p(3)) ;
  printf("% d",p(3)) ;
}
```

	x	y	z	p(3)
p	3	2	2（3）	8
p	3	2	3（4）	9
p	3	2	4（5）	10

定义的静态存储变量无论是做全局变量或是局部变量，其定义和初始化在程序编译时进行。

作为局部变量，调用函数结束时，静态存储变量不消失并且保留原值。

例:利用静态局部变量求 1+2+…+10,程序如下：

```
#include <stdio.h>
int fun(int n)
{
        static  int s=0;
        s=s+n;
        return s;
}
void main( )
{
        int n,p;
        for(n=1;n<=10;n++)
           p=fun(n);
        printf("%d\n",p);
}
```

(2) 静态全局变量

定义在所有函数（包括主函数）之外，用关键字"static"标识的变量，称为静态全局变量。

生存期：静态全局变量和外部变量都具有全局寿命，即在整个程序运行期间都存在。

作用域：静态全局变量只能在所定义的文件中使用，具有局部可见性。这与外部变量不同。

注意：自动变量没有赋初值时，其值是一个随机值。对于静态变量或外部变量没有赋初值时，数值型变量的值系统默认为0。

例如：　　static int x;

局部静态变量的初始值为 0 (数值)，或 "\0" （字符串）

静态局部变量自动变量的区别：

（1）静态局部变量属于静态存储类别，在静态存储区内分配存储单元。在程序整个运行期间都不释放。而自动变量属于动态存储类别，函数调用结束后即释放。

（2）静态局部变量在编译时赋初值，即只赋初值一次；而对自动变量赋初值是在函数调用时进行，每调用一次函数重新给一次初值，相当于执行一次赋值语句。

（3）如果在定义局部变量时不赋初值的话，则对静态局部变量来说，编译时自动赋初值0（对数值型变量）或空字符（对字符变量）。而对自动变量来说，如果不赋初值则它的值是一个不确定的值。

（4）静态局部变量的生存周期虽然为整个源程序，但其作用域仍与自动变量相同，即只能在定义变量的函数内部使用。离开函数后，尽管该变量还继续在内存中存在，但并不能使用。如前例中的静态变量s，其作用域为函数fun 内部。

●外部变量

外部变量是指在函数外部定义的变量。外部变量是一个全局变量，其作用范围是从定义开始到本源文件末尾结束。C 语言中外部变量固定分配在静态存储区保存。为了扩大外部变量的作用范围，可以用关键字"extern"来声明外部变量。外部变量的作用域不仅可以是一个文件范围，而且可以包含多个文件范围。

1．在一个源文件内声明外部变量

如果一个全局变量不在文件开头处定义，则其作用范围是从定义开始到本源文件末尾结束。如果在该变量的定义处前需要引用该变量，则应将此变量声明为 extern 类别的变量，表示此变量是一个外部变量。

```
/*01*/ #include <stdio.h>
/*02*/ void main( )
/*03*/ {int max(int, int);
/*04*/ extern a; /*声明外部变量a*/
/*05*/ int x=7, y=8, result;
/*06*/ result=(x+y)*a+max(x,y)*a;
/*07*/ printf("result=%d\n", result);
/*08*/ }

/*1*   int a=10; /*定义外部变量a*/
/*02*/ int max(int x, int y)
/*03*/ {
/*04*/      return(x>y?x:y);
/*05*/ }
```


例　外部变量引用示例

```
#include <stdio.h>
int x=10; /*外部变量x的定义*/
void f1( )
{   x++;
    printf("x=%d ",x) ;
    printf("y=%f\n",y);
}
float y=2.0;/*外部变量y的定义*/
void main( )
{  int x=1;
   x++;
   f1( );printf("x=%d,y=%f\n",x,y) ;
}
extern float y; /*外部变量声明，扩展了外部变量y的作用域*/
```

编译结果：
Error: 'y': undeclared identifier

运行结果：
x=11, y=2.000000
x=2, y=2.000000

2．在多个源文件的程序中声明外部变量

一个C 程序可以由多个源程序文件组成，如果在一个文件file1.c 中想引用另一个文件file2.c 中的全局变量a，则需要在file1.c 中使用关键字"extern"来声明为外部变量。注意，此时在文件file1.c 中不能再定义外部变量a，否则编译系统会提示"变量重复定义"。例如：

第一个文件的内容如下：

```
/*file1.cpp*/
#include <stdio.h>
int a=2;
static int b=3;
void func( )
{
    a++; b++;
    printf("a=%d,b=%d\n",a,b);
}
```

第二个文件的内容如下：

```
/*file2.cpp*/
#include <stdio.h>
extern int a;
int b;
void main( )
{   void func(void);
    func( );
    printf("a=%d, b=%d\n",a,b);
}
```


将两文件置于同一工程中。

程序运行结果如下：

a=3,b=4

a=3,b=0

从上述运行结果可以看出，文件file1.cpp中的外部变量a可以在文件file2.cpp中使用，只需在file2.cpp中加上声明语句：extern int a;即可.

但file1.cpp中的静态全局变量b不能在file2.cpp中使用。若读者将file2.cpp中变量定义语句：int b;改写成变量声明语句：

```
extern int b;
```

想一想，会出现什么错误？为什么？

表5.1不同变量存储类型的区别

变量类型	存储类型	可见性	存在性	未赋初值
局部变量	自动变量	定义的范围内	离开定义范围，值消失	值不确定
	寄存器变量	定义的范围内	离开定义范围，值消失	值不确定
	静态变量	定义的范围内	离开定义范围，值仍保留	值为0
	形参	定义的函数内	离开定义函数，值消失	值不确定
全局变量	静态变量	本文件内	程序运行期间，值有效	值为0
	外部变量	本文件或其他文件	程序运行期间，值有效	值为0

25

目前我们在一般的编程实践中，绝大部分涉及到的都是"自动变量"(auto)，偶尔会涉及到"静态局部变量"(static)或"外部变量"(extern)。而"自动变量"(auto)在书写时可以缺省.

要重点理解"自动变量"的作用域与生存期。

26

局部变量和全局变量的小结

- 函数相关性
  - 局部变量只在某个函数内部有效，因此功能明确，不容易混淆。
  - 全局变量解决了函数只能有一个返回值的问题，但可能导致程序可读性差、函数独立性降低。如在编制大型程序时，变量值有可能在程序其它地点偶然改变。
- 内存占用
  - 局部变量在需要时才开辟存储单元。
  - 全局变量在程序的整个执行过程中都占用存储单元。

27

【例5.20】说明全局变量、自动局部变量和静态局部变量的作用域的程序实例。

```
#include <stdio.h>
void u(void), v(void),w(void);//函数原型说明
int a = 2;
int main()/*一个说明变量作用域的小程序*/
{   int a = 4;
    printf("主函数中，外层局部变量a的值是%d\n ", a); //4
   { int a = 8; printf("主函数中，内层局部变量a的值是%d\n", a);}//8
    printf("程序从主函数的内层程序块退出\n ");
    printf("主函数中，外层局部变量a的值是%d\n ", a);
    u();   //函数u()拥有自动局部变量a
    v();   //函数v()拥有静态局部变量a
    w();   //函数w()没有局部变量，使用全局变量a
    u();   //函数u()对自动局部变量a重新初始化
    v();   //静态局部变量a保持前一次调用后的值，函数v()对它继续修改
    w();   //函数w()没有局部变量，继续使用全局变量a
    printf("主函数中的局部变量a是%d\n", a);//4
    return 0;
}
```

28

```
void u(void)
{
    int a = 20; /*a是函数u()的自动局部变量，每次调用都被建立和
                  初始化*/
    printf("进入函数u()时，函数u()的局部变量a是%d\n", a);//20
    a += 5;
    printf("离开函数u()时，函数u()的局部变量a是%d\n", a);//25
}
void v(void)
{  static int a = 30; /*a是函数v()的静态局部变量，首次调用时初
                        始化*/
    printf("进入函数v()时，函数v()的静态局部变量a是%d\n", a);//30
    a += 2;
    printf("离开函数v()时，函数v()的静态局部变量a是%d\n", a);//32
}
void w(void)
{
    printf("进入函数w()时，全局变量a是%d\n", a);//2
    a += 3;
    printf("离开函数w()时，全局变量a是%d\n", a);//5
}
```

29

【程序说明】

全局变量a在定义时被初始化为2。以后在程序块中如果用标识符a定义新的变量，都会将全局变量a隐藏起来。主函数定义局部变量a，并初始化为4，然后输出它的值。在主函数的程序块中又用a定义新的局部变量，并把它初始化为8，该值的输出说明外层的局部变量a和全局变量a都被隐藏起来了。在程序退出这内层程序块时，值为8的内层局部变量a已被自动撤销。然后再输出a是外层的局部变量。

30

程序定义了三个没有形参没有返回值的函数。
函数u()定义了自动局部变量a，a的初值为20。函数u()输出a的值，然后将a的值增加5后再输出，结束函数。每次调用函数u()都要将自动局部变量a重新建立和初始化为20，函数输出a后，a增加5，再输出增加5后的a，函数返回时将a撤销。

函数v()定义了静态局部变量a，a的初值为30。函数v()输出a的值，然后将a的值增加2后再输出，函数返回后，v()的a保留着值32。下一次再调用函数v()，静态局部变量a就拥有值32，函数输出32后，a又增加2，再输出增加2后的a，函数返回后，a保留着值34。

函数w()没有定义局部变量，使用的是全局变量a。函数w()输出全局变量a，将a增加3，然后再输出a。最后，主函数输出局部变量a。输出的值说明没有别的函数对这个局部变量a有修改。

31

运行上述程序，输出结果为：
```
    主函数中，外层局部变量a的值是4
    主函数中，内层局部变量a的值是8
    程序从主函数的内层程序块退出
    主函数中，外层局部变量a的值是4
    进入函数u()时，函数u()的局部变量a是20
    离开函数u()时，函数u()的局部变量a是25
    进入函数v()时，函数v()的静态局部变量a是30
    离开函数v()时，函数v()的静态局部变量a是32
    进入函数w()时，全局变量a是2
    离开函数w()时，全局变量a是5
    进入函数u()时，函数u()的局部变量a是20
    离开函数u()时，函数u()的局部变量a是25
    进入函数v()时，函数v()的静态局部变量a是32
    离开函数v()时，函数v()的静态局部变量a是34
    进入函数w()时，全局变量a是5
    离开函数w()时，全局变量a是8
    主函数中的局部变量a是4
```
32

●内部函数与外部函数(简介)

C 程序中的函数可能分布在多个文件中。可根据其使用范围将这些函数分为两种：内部函数和外部函数。内部函数只能被函数所在文件中的函数调用。外部函数既可以被同一个文件中的函数调用，也可以被其它文件中的函数所调用。

♦内部函数

内部函数，又称静态函数。如果在一个源文件中定义的函数，只能被本文件中的函数调用，而不能被同一程序其它文件中的函数调用，这种函数称为内部函数。定义一个内部函数，只需在函数类型前再加一个"static"关键字即可，如下所示：

```
static  函数类型  函数名(形参表列)
{     说明部分；执行部分；}
```

说明：

（1）此处"static"的含义不是指存储方式，而是指对函数的作用域仅局限于本文件。内部函数定义时，关键字"static"一定不能省略。

（2）使用内部函数的好处是：不同的人编写不同的函数时，不用担心自己定义的函数，是否会与其它文件中的函数同名，因为同名也没有关系。

33

♦外部函数

在定义函数时，如果没有加关键字"static"，或冠以关键字"extern"，表示此函数是外部函数。定义形式如下：

```
[extern] 函数类型 函数名(形参表列)
{     说明部分；执行部分；}
```

一般情况下，没有特别说明，定义的函数时，均为外部函数。我们前面所编的程序实际上都是外部函数。

例：内部函数和外部函数的使用。

```
/*源程序名：ch1.cpp*/
/*01*/ int a=10; /*定义外部变量a*/
/*02*/ extern int max(int x, int y) /*定义外部函数max，
         extern 可以省略*/
/*03*/ {
/*04*/      return(x>y?x:y);
/*05*/ }
```

34

```
/*源程序名：ch_2.cpp*/
/*01*/ #include <stdio.h>
/*02*/ extern a; /*声明外部变量a*/
/*03*/ extern int max(int x, int y); /*声明外部函数max*/
/*04*/ static int min(int x, int y) /*定义内部函数min，static 不能省略*/
/*05*/ {
/*06*/       return(x<y?x:y);
/*07*/ }
/*08*/ void main( )
/*09*/ {
/*10*/       int x=7, y=8, result1, result2;
/*11*/       result1=(x+y)*a+max(x,y)*a;
/*12*/       a-=3;
/*13*/       result2=min(a, max(x,y));
/*14*/       printf("result1=%d, result2=%d \n", result1, result2);
/*15*/ }
```

程序的运行结果为：

```
result1=230, result2=7
```

说明：源程序ch_1.cpp 中的函数不能访问源程序ch_2.cpp 中的static 函数。

35

5.9 预处理命令简介

◆"编译预处理"是C语言编译系统的一个组成部分。编译预处理是在编译前由编译系统中的预处理程序对源程序的预处理命令进行加工。

■源程序中的预处理命令均以"#"开头，结束不加分号，以区别源程序中的语句，它们可以写在程序中的任何位置，作用域是自出现点到源程序的末尾。

◆预处理命令包括执行宏定义(宏替换)、包含文件和条件编译。宏定义的目的是允许程序员以指定标识符代替一个较复杂的字符串。本节将介绍如何应用预处理程序的开发过程，并提高程序的可读性。

36

C提供的预处理功能主要有以下三种：

1）宏定义
2）文件包含
3）条件编译

37

1. 宏定义 C语言的宏定义分为两种：

不带形式参数的宏定义(简单宏定义)
与带参数的定义。

宏定义的作用：用标识符来代表一串字符。
宏定义后，该程序中宏名就代表了该字符串。

一旦对字符串命名，就可在源程序中使用宏定义标识符，系统编译之前会自动将标识符替换成字符串。

38

♦不带形式参数的宏定义

不带形式参数的宏定义的一般形式为：

#define 宏名 宏体

其中，"#"表示这是一条预处理命令，define为宏定义命令；

宏名：

为一个合法的标识符，通常使用大写英文字母和有直观意义的标识符命名，以区别于源程序中的其它标识符；

宏体：

可以是常数、表达式或语句，甚至可以是多条语句。三者之间要用一个或多个空格分隔。

例如：

```
#define PI 3.1415926
#define MSG printf("Hello,");printf("world!\n");
```

在预处理时，系统在该宏定义以后出现的每一个宏名都用宏体来代替，这个过程叫宏替换。

39

如有：

```
#define TRUE   1
#define FALSE  0
```

则在它们被定义的源程序文件中，后面程序正文中凡出现TRUE的地方将用1替代之；出现 FALSE的地方用0替代之。以下是经常使用的不带参数宏定义的例子：

```
#define PI   3.1415926
#define NL   printf("\n")
#define EOF  (-1)
```

40

```
#include <stdio.h>
#define PRICE  30
void main()
{
        int num,total;
        num=10;
        total=num*PRICE;
        printf("total=%d\n",total);
}
```

定义PRICE代表常量30,此后凡在本文件中出现的PRICE都代表30，可以和常量一样进行运算，程序运行结果为:

total=300

通常，不带形式参数的宏定义多用于定义常量。 程序中统一用宏名表示常量值，便于程序前后统一，不易出错，便于修改，提高程序的可读性和可移植性。

41

附例：宏体可以是多条语句

```
#include <stdio.h>
#define MSG   printf("Hello,");printf("world!\n");
void main( )
{
      MSG
}
```

程序运行结果：

Hello, world!

42

使用符号常量的好处和特点：

好处：

(1)含义清楚

如上面的程序中，看程序时从PRICE就可知道它代表价格。因此定义符号常量名时应考虑"见名知意"。在检查程序时搞不清各个常数究竟代表什么。应尽量使用"见名知意"的变量名和符号常量。

(2)在需要改变一个常量时能做到"一改全改"

例如在程序中多处用到某物品的价格，如果价格用常数表示，则在价格调整时，就需要在程序中作多处修改，若用符号常量PRICE代表价格，只需改动一处即可。如：

```
#define  PRICE 35
```

在程序中所有以PRICE代表的价格就会一律自动改为35。

43

特点：

（1）C语言中，用宏名替换一个字符串是简单的转换过程，不作语法检查。若将宏体的字符串中符号写错了，宏展开时照样代入，只有在编译宏展开后的源程序时才会提示语法错误。例如：

```
#define  E  2.71828
```

若把字母8写成B，即：`#define  E  2.71B2B`

预处理时照样替换，而不管其含义是否正确，一直到对宏展开的结果进行编译时，才会产生错误提示。

（2）宏定义命令行放在源程序的函数外时，宏名的作用域从宏定义命令行开始到本源文件结束。

（3）宏名的作用域可以使用#undef命令终止，形式如下：

```
#undef  标识符
```

（4）在宏定义中，允许在宏体字符串中使用已定义过的宏名，这个过程称为嵌套宏定义。

44

【例5.20】用如下公式计算圆周率π的近似值，直到最后一项的绝对值小于ε。π≈ 4 - 4/3 + 4/5 - 4/7 + …

```
#include <stdio.h>
#include <math.h>
#define Epson 1e-4  /* 定义宏名Epson */
void main()
{ int sign = 1;   /* 符号位，奇次为正，偶次为负 */
  long i, m = 1;  /* m 为分母，第一项分母为1 */
  double pi = 4, t = 4; /* t 为某一项值 */
  for(i = 1; fabs(t) > Epson; i++)
  { sign = -sign;/* 改变符号 */
    m += 2;       /* 分母 +2 */
    t=sign*4.0 / m;/* 计算某一项的值 */
    pi += t;
  }
  printf("pi = %f\n", pi);
 }
```

采用宏定义后，如果要修改Epson值，只需在宏定义处修改Epson值，而不需在出现的地方等多处修改，减少了代码修改量，降低了程序出错的可能性，提高了程序的健壮性。

45

♦带形式参数宏定义

带形式参数的宏定义进一步扩充了不带形式参数宏定义的能力，在字符序列替换同时还能进行参数替换。

带形式参数宏定义的一般形式为：

```
#define  标识符(形式参数表)  字符序列
```

形式参数表中的形式参数之间用逗号分隔，字符序列中应包含形式参数表中的形式参数。宏名与后续圆括号之间不能留空格。(为什么？)

46

如有宏定义：

```
#define  MAX(A, B)  ((A) > (B) ? (A) : (B))
```

预处理时，系统在用宏体代替宏名的同时，实在参数会代替宏体中形式参数，同样宏替换仍只是一种简单的替换，不能进行计算或其它的功能。

当程序出现下列语句时：

```
y = MAX(p+q, u+v)
```

程序在预处理时，将被替换成如下语句：

```
y = ((p+q) > (u+v) ? (p+q) : (u+v));
```

又如：

```
#define M(a,b) (a)>(b)?(a)-(b): (a)+(b)
#define M(a,b) ((a)>(b)?(a)-(b): (a)+(b))
int i=20,j=7;  // int i=7,j=20;
printf("%d\n",M(i,j)*5);
printf("%d\n",5*M(i,j));
```

47

【例5.21】使用宏定义，计算半径为2、4、6、8、10时圆的面积和圆的周长。

```
#include <stdio.h>
#define  PI  3.141592653
#define  AREA(R)  PI * (R) * (R)
#define  PERI(R)  2 * PI * (R)
void main()
{ int r;
  for(r = 2; r <= 10; r += 2)
  { printf(" 半径 = %2d, 圆面积 = %f\t",
                r, AREA(r));
    printf("圆周长 = %f\n", PERI(r));
  }
}
```

48

对带形参的宏定义，请注意以下几点：

（1）在宏定义时，标识符与左圆括号之间不允许有空白符号，应紧接在一起，否则将空白字符之后的字符都作为字符序列的一部分，即变成了不带形参的宏定义了。

（2）在宏调用时，实参的个数必须与宏定义中的形参个数相同。

（3）在定义带参数的宏时，宏体中的所有形式参数和整个表达式最好都加圆括号，否则在宏替换后的表达式中，运算顺序可能会发生不希望的变化。

有如下宏定义

```
#define SQR(x) x * x
```

则宏调用p = SQR(y); 被展开为p = y * y;

但宏定义

```
q = SQR(u + v); 被展开为
q = u + v * u + v;
```

显然，这个展开结果不是编程者所希望的。

为能保持实参的独立性，应在宏定义中给形参加上括号"（）"。如，SQR宏定义改写为：

```
#define SQR(x) (x) * (x)
```

如要保证宏定义整体的完整性，还可以将宏定义中的序符序列加括号。如，SQR宏定义进一步改写为：

```
#define SQR(x) ((x) * (x))
```

对于最后的宏定义，含宏调用的表达式

```
r = 1.0/SQR(u+v)
```

也能得到正确结果：

```
r = 1.0/((u+v)*(u+v))
```


宏和函数的区别

- 函数调用在程序运行时实行，占用运行时间；
  宏展开是在编译阶段进行，只占用编译时间；
- 函数调用时，先求出实参的值，再传递给形参；而宏展开不进行实参运算，直接进行字符串替换；
- 宏定义与宏调用是为了减少书写量和提高运行速度；而函数定义、函数调用是为了实现模块程序设计，便于构造软件。函数调用要求实参与形参类型一致，而宏没有类型概念，只有字符序列的对应关系。
- 宏与函数可以互相替代：

```
#define MAX(x,y) (x)>(y)?(x):(y)
```

⇕

```
int max(int x,int y){return(x>y?x:y);}
```


●宏调用展开后的代码是嵌入源程序中的，且每调用一次，嵌入一次代码。因此，宏调用时总的程序代码是增加的；而函数调用是执行时转入对应的函数，执行后返回主调函数，无论调用多少次，函数体的代码都不会增加。所以函数也解决代码重用问题。

●除了将宏展开结果嵌入源程序外，宏调用不存在内存分配问题；而对函数可能需分配临时空间以存放函数调用之结果。

2. 文件包含

◆编译预处理的文件包含功能是一个源程序通过#include命令把另外一个源文件的全部内容嵌入到源程序中的#include命令行所在的位置。

◆在编译时并不是作为两个文件联接，而是作为一个源程序编译，得到一个目标文件。

图1是一个文件包含关系的示意图。文件file1.c中的包含命令#include "file2.c"将文件file2.c包含进文件file1.c。图1a和图1b是预处理前的情况，图1c是将文件file2.c包含进文件file1.c之后的file1.c结构示意图。

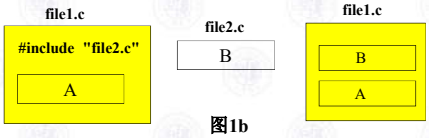

图1a　图1b　图1c

图1

●#include命令的两种形式

格式一：#include <文件名>

本命令的特点是在文件名的首尾用尖括号括起来。文件名可以带路径。在预处理时只检索C语言编译系统所确定的标准目录（即C系统安装后形成的include子目录，该子目录中有系统提供的头文件）中查找包含文件。

格式二：#include "文件名"

本命令格式的特点是在文件名的首尾用双引号括起来。其中，文件名指出的是待包含进来的文件，且可以带路径。首先对使用包含文件的源文件所在的目录进行检索，若没有找到指定的文件，再在标准目录中检索。因此格式二的查找功能包含了格式一的查找功能。另外，两种格式的include后可不带空格。

例：　设计一个求n!的函数，存放于文件exam.cpp中，然后设计主函数文件file.cpp，计算n!可以编写文件exam.cpp如下：

```
/*exam.cpp*/
long fact(int n)
{
   if(n<=1)return 1;
   return n*fact(n-1);
}
```

另一文件file.cpp的内容如下：

```
/*file.cpp*/
#include <stdio.h>
#include "exam.cpp"
void main( )
{  int n;
   printf("Input n\n");
   scanf("%d",&n);
   printf("%ld\n",fact(n));
}
```


【例5.23】计算x^y。

文件file1.c中的内容：

```
#include <stdio.h>
#include "d:\c\file2.c"  /* 指明要包含文件
                            file2.c的完整路径 */
int main()
{ int x, y;
  double power(int, int); /* 函数说明 */
  printf(" 请输入计算 x**y 的两个自然数：");
  scanf("%d%d", &x, &y);
  printf("%d**%d = %.0f\n",
              x, y, power(x, y));
  return 0;
}
```


D盘C目录下file2.c文件中的内容：

```
double power(int m, int n)
{ int i;
  double y = 1.0;
  for(i = 1; i <= n; i++)
      y *= m;
  return y;
}
```

#include命令的嵌套使用

当一个程序中使用#include命令嵌入一个指定的包含文件时，被嵌入的文件中还可以使用#include命令，又可以包含另外一个指定的包含文件。例：

f1.cpp文件：

```
#include "f2.cpp"
void fu1( )
{
 ...
}
```

f2.cpp文件：

```
void fu2( )
{
 ...
}
```

f12.cpp文件：

```
#include "f1.cpp
"
void main( )
{
 ...
}
```

61

3.条件编译

一般情况下，源程序的所有语句都会参加编译。但有时若希望只对其中的部分有选择地进行编译，该过程称为条件编译。使用条件编译功能，为程序的调试和移植提供了有力的机制，使程序可以适应不同系统和硬件设置的通用性和灵活性。

条件编译是在对源程序编译之前的处理中，根据给定的条件，决定只编译其中的某一部分源程序，而不编译另外一部分源程序。

62

条件编译命令主要有三种形式。

♦ #ifdef 标识符
  程序段1
#else
  程序段2
#endif

功能是：

如果标识符已经被#define命令定义过，则在程序编译时只对程序段1进行编译，否则只对程序段2进行编译。其中的程序段即可以是一条语句，也可以是一组语句。如果是一组语句，也不必象复合语句一样加上花括号。

63

注意：标识符在预处理#define命令中可以被定义为任何字符，甚至后面什么也不写，如：

```
#define OK 1
#define OK a
#define OK
```

在上述一般形式中，如果程序段2 为空，则可简写成如下一般形式(单分支)

```
#ifdef 标识符
  程序段
#endif
```

64

```
#include <stdio.h>
#define OK 1
#ifdef OK
    #define STRING "you have defined OK!"
    #define STRING1 "\nOK=1\n"
#else
    #define STRING "you have not defined OK!"
    #define STRING1 "\nOK未定义\n"
#endif
void main( )
{   printf(STRING);
    printf(STRING1);
}
```

运行结果

```
you have defined OK!
OK=1
```

65

♦ #ifndef 标识符
  程序段1
#else
  程序段2
#endif

格式二与格式一的不同之处是将"ifdef"改成"ifndef"。

其功能是：如果标识符没有被#define命令定义过，则在程序编译时只对程序段1进行编译，否则只对程序段2进行编译。这与格式一的功能恰好相反。但二者在用法上是相似的，不再赘述！

在上述形式中，当程序段2不出现时，也可简写成

```
#ifndef 标识符
  程序段
#endif
```

66

♦ #if 表达式
程序段1
#else
程序段2
#endif

功能是：如果常量表达式的值为真（即非0），则在程序编译时只对程序段1进行编译，否则只对程序段2进行编译。

当程序段2不出现时，也可简写为

#if 表达式
程序段
#endif


```
#include <stdio.h>
#define A 2
void main( )
{
  #if (A > 0 )
      printf("A>0\n");
  #else
      printf("A<0 or A=0\n");
  #endif
}
```

程序运行结果：

A＞0

与条件语句的区别：若用条件语句将会对整个源程序进行编译，造成目标程序长，运行时间长；而采用条件编译，可减少被编译的语句，从而减少了目标程序的长度和运行时间。

条件编译预处理命令也可呈嵌套结构。特别是为了便于描述 #else 后的程序段又是条件编译情况，引入预处理理命令符 #elif。它的意思就是else if。例如：

```
#if  常量表达式1
      语句1
#elif  常量表达式2
      语句2
     …
#elif  常量表达式n
      语句n
#else
      语句n+1
  #endif
```


4. #undef命令行

#undef 标识符

其中#undef为预编译符。

功能：使所指标识符变为无定义。