语言变量声明和变量赋值

1）基本数据类型

在C语言中，仅有4种基本数据类型—整型、浮点型、指针和聚合类型（如数组和结构等），所有其他的类型都是从这4种基本类型的某种组合派生而来。

整型：整型家族包括字符、短整型、整型和长整型，它们都分为有符号（signed）和无符号（unsigned）两种版本。规定整数值相互之间大小的规则很简单：长整型至少应该和整型一样长，而整型至少应该和短整型一样长。

浮点类型：诸如3.14159和2.3×1023这样的数值无法按照整数存储。第一个数为非整数，而第二个数远远超出了计算机整数所表达范围，这样的数就可以用浮点数的形式存储。浮点数家族包括float、double和long double类型。通常，这些类型分别提供单精度、双精度以及在某种扩展精度的机器上提供扩展精度。ANSI标准仅仅规定long double至少和double一样长，而double至少和float一样长。标准同时规定了一个最小范围：所有浮点类型至少能够容纳从10-37到1037之间的任何值。

指针：变量的值存储于计算机内存中，每个变量都占据一个特定的位置。每个内存的位置都由地址唯一确定并应用，就像一条街上的房子由他们的门牌号码标识一样，指针知识地址的另一个名字。

2）变量声明形式

只知道基本的数据类型是远远不够的，你还应该知道怎样声明变量。变量的基本形式是：

说明符（一个或多个） 声明表达式列表

对于简单的类型，声明表达式列表就是被声明的标识符的基本类型。对于相对比较复杂的类型，声明表达式列表的每个条目实际上是一个表达式，显示被声明的名字的可能用途。

例如：int a, double b;

该语句就是一条声明语句，其中a,b就是变量名，该语句指明了变量a,b是int数据类型。所有变量在使用前都必须写在执行语句之前，也就是变量声明要与执行语句相分离，否则就是出现编译错误。

3）变量命名

C语言中任何的变量名都是合法的标示符。所谓标识符就是由字母、数字和下划线组成的但不以数字开头的一系列字符。虽然C语言对标示符的长度没有限制，但是根据ANSI标准，C编译器必须要识别前31个字符。C语言是对大小写敏感的，即C语言认为大写字母和小写字母的含义是不同的，因此a1和A1是不同的标识符。

到目前为止，没有一种命名规则可以让所有的程序员赞同，程序设计教科书一般都不指定命名规则。常用的命名规则有匈牙利命名法和驼峰命名法等，在实际操作中，我们会采取相对简单方便的命名规则，即“类型首字母”+“_”+“变量用途英文缩写”，其中英文缩写首字母为大写，例如 int i_Num，char c_Name[5]。

4）变量赋值方式

在一个变量声明中，你可以给一个标量变量指定一个初始值，方法是在变量名后面跟一个等号（赋值号），后面就是你想要给变量的值。例如：

```
int  i_Num=10;
char c_Name[]=”student”;
```

上述语句声明i_Num为一个整数变量，其初始值为10，声明c_Name为一个字符数组，其初始值为“student”。

在C语言中，全局变量和静态变量，如果没有赋初值，则默认初始值int，float，char分别为0,0.0，’\0’，除了全局变量和静态变量以外，其他变量如果没有赋初值，则默认初始值为内存中的垃圾内容，对于垃圾内容不能有任何假设。**注意：**定义指针后，默认初始值不是0，而是随机的一个值，故定义指针后，一定要初始化。

在实际操作中，变量的赋值都是以赋值语句的形式出现，赋值语句是由赋值表达式再加上分号构成的表达式语句。其一般形式为：

变量=表达式;

在赋值语句的使用中需要注意以下几点：

1. 由于在赋值符“=”右边的表达式也可以又是一个赋值表达式。

下述形式:

变量=（变量=表达式）;

该语句是成立的，从而形成了嵌套的情形。其展开后的一般形式为：

变量=变量=…=表达式;

例如：a=b=c=d=e=5;

按照赋值运算符的右结合性，因此实际上等效于：e=5;d=e;c=d;b=c;a=b;

2. 注意在变量声明中给变量赋初值和赋值语句的区别

给变量赋初值是变量说明的一部分，赋初值后的变量与其后的其他同类变量之间仍必须用逗号隔开，而赋值语句则必须用分号隔开。

例如：int a=5,b,c;

3. 在变量声明中，不允许连续给多个变量赋初值。

如下述说明就是错误的：

int a=b=c=5;

正确写法为：int a=5,b=5,c=5;

但是，赋值语句允许连续赋值。

4. 注意赋值表达式和赋值语句的区别。

赋值表达式是一种表达式，它可以出现在任何允许表达式出现的地方，而赋值语句则不能。下述语句是合法的：

If((x=y+7)>0) z=x;

语句功能为若表达式x=y+5大于0则z=x。

但是，下述语句是错误的：

If((x=y+7;)>0) z=x;

因为x=y+7；是语句，不能出现在表达式中。

实例：

```
#include “stdafx.h”
void main()
{
        int i_Tmp,i_Type=8;
        float f_Tmp;
        double d_Tmp;
```

```
        char c_Tmp;
        d_Tmp=d_Tmp=f_Tmp=12;
        f_Tmp=i_Type;
        i_Tmp=i_Type+3;
        printf("a=%d,b=%d,c=%.3f,d=%.6lf",i_Tmp,i_Type,f_Tmp,d_Tmp);
}
```

算术运算符及使用方式

C语言提供了最基本的算术运算符，如下表：

运算符	含义	举例	结果
+	加法运算符	a+b	a和b的和
-	减法运算符	a-b	a和b的差
*	乘法运算符	a*b	a和b的乘积
/	除法运算符	a/b	a除b的商
%	求余运算符	a%b	a除b的余数
++	自加运算符	a++,++a	a自加1
--	自减运算符	a--,--a	a自减1

1、+、-、*、/都适用于浮点类型和整数类型，当两个操作数都为整数时进行整数运算，其余情况则进行double型运算；当/除法运算符的两个操作数为整数时，结果为整数，舍去小数部分，例如5/3的结果为1；%求余运算符只接受两个整型操作数的运算，结果为余数

2、++、--：作用是使变量自加1或自减1，例如i++、++i，都是使i的值加1，但其执行的步骤是不同的。例如：

```
int i=3，j;
j=i++; // i的值为4 ，j的值为3
int i=3，j;
j=++i; //i的值为4，j的值为4
```

可见当变量在左侧时，先进行赋值运算再进行自加1操作，当变量在右侧时，先进行自加1操作再进行赋值运算。

3、在赋值运算符之前加上算术运算符既构成复合运算符，例如：a+=b，等价于a=a+b。-=、*=、/=也是如此。

位运算符及使用方式（<<、>>、~、|、&、^）

位运算符是用来对二进制位进行操作，如下表：

运算符	含义
<<	左移
>>	右移
~	取反
|	按位或
&	按为与
^	按为异或

<<、>>：移位运算符，例如左移运算符：

int i=3；i=i<<4;

3的二进制位为00000011，左移4位的结果为00110000，其操作中高位舍弃、低位补0，既i=48，等同于i乘以2的4次方。

右移运算符则有所不同，操作中是低位舍弃，高位则有两种补位方式。一种为逻辑移位，高位补0；另一种为算术移位，当符号位为1时高位全部补1，当符号位为0时则高位全部补0。具体使用哪种移位方式则取决于当前的编译环境。

~：取反运算符，为单目运算符，其操作是对操作数的二进制位按位求反，既1变0,0变1。例如i=5，二进制位为00000101，取反的结果为11111010。

在计算机系统中，数值一律用补码来表示和存储，其中最高位为符号位，用0表示正，1表示负。补码的规定如下：

1. 对正数来说，最高位为0，其余各位代表数值本身，例如14的补码为00001110；对负数而言，则将该数绝对值的补码按位取反，再加1，得该数的补码，既-i=~i+1。例如-14的补码为14的二进制00001110取反加1得11110010。

2. |、&、^：均为双目运算符，对操作数的二进位进行运行，且操作数以补码的方式出现。

3. |按位或，两个对应的二进位至少有一个为1则为1，否则为0；&按位与，两个对应的二进位都为1则为1，否则为0；^按位异或，两个对应的二进位不同则为1，否则为0。例如：

a=5;　　(00000101)

b=14;　(00001110)

a|b=15;(00001111)

a&b=4;　(00000100)

a^b=11; (00001011)

关系运算符及使用方式（>、>=、<、<=、==、！=）

运算符	含义
>	大于
>=	大于或等于
<	小于
<=	小于或等于
==	等于
！=	不等于

关系运算符用于比较两个数值之间的关系，例如：a>3为关系表达式，大于号为关系运算符，当表达式成立时，“a>3”的值为“真”，当“a>3”不成立时，“a>3”的值为“假”。

其中应当注意的是关系表达式的返回值为整型值，而不是布尔型。表达式为真时返回值为1，表达式为假时返回值为0。

逻辑运算符及使用方式（&&、||、！）

运算符	含义	举例	结果
&&	逻辑与	a&&b	a,b都为真则结果为真，否则为假
||	逻辑或	a||b	a,b至少有一个为真则结果为真，否则为假
!	逻辑非	!a	当a为真则结果为假，当a为假则结果为真

其中应当注意逻辑或，例如a||b，当a为真时，C语言中直接跳过对b的判断，其返回值为“真”。

当一个表达式包括几种运算符时，则以运算符的优先级对表达式进行运算，表达式的优先级如下：

优先级	运算符类型	说明
1	初等运算符	()、[]、->、.
2	单目运算符	！、~、++、--、*(指针运算符）、&(取地址运算符)
3	算术运算符	先乘除后加减
4	关系运算符	>、>=、<、<=、==、！=
5	逻辑运算符	&&、||
6	条件运算符	三目运算符，例如？：
7	赋值运算符	=
8	逗号运算符	,

指针的概念与使用

指针的定义

指针就是变量的地址，是一个常量。定义指针的目的就是为了通过指针访问内存单元。在C语言中，允许用一个变量来存放指针，这种变量称为指针变量。

指针变量定义的一般形式为：

*存储类型 数据类型 * 指针变量名*

指针变量运算符

1、取地址运算符：&

该运算符表示的是对&后面的变量进行取地址运算。

例：int a;

则 &a表示取变量a的地址，该表达式的值为变量a的首地址。

2、指针运算符：*

该运算符也称为“取内容运算符”，后面接一个指针变量。表示的是访问该指针变量所指向的变量，即访问指针所指向的存储空间中的数据。

例：int a=7;

int *p;

p=&a;

则 *p 表示指针变量 p 指向变量 a，即 *p 就是 a，所以 *p=7。

一个指针变量 p 在程序中通常有如下表示形式：

p： 指针变量，它的内容是地址量；

*p：指针所指向的变量，是指针所指向的内存空间中的数据；

&p：指针变量所占存储空间的地址；

【例1】分析程序的运行结果

源程序如下：

```
#include<stdio.h>
void Locate()
{
     int i_a;
       int *pst_a;
       printf("\n请输入i_a的值:");
       scanf("%d",&i_a);
       pst_a =&i_a;
       printf("i_a的值为：%d\n",i_a);
       printf("pst_a的值为：%x\n", pst_a);
       printf("&i_a的值为：%x\n",&i_a);
```

```
        printf("*pst_a的值为：%d\n",* pst_a);
        printf("&pst_a的值为：%x\n",& pst_a);
        printf("\n");
}
```

运行结果：

```
        请输入i_a的值:3
        i_a的值为：3
        pst_a的值为：12fe8c
        &i_a的值为：12fe8c
        *pst_a的值为：3
        &pst_a的值为：12fe80
```

以上实例中，12fe8c是pst_a的值，也就是i_a的地址；12fe80是pst_a的地址；两者有区别，不能混为一谈。

地址与指针的概念

指针可以有效地表示复杂的数据结构；动态分配内存；方便的使用字符串；有效而方便地使用数组；能直接处理内存地址。

如果在程序中定义了一个变量，在编译时就给这个变量分配内存单元。系统根据程序中定义的变量的类型，分配一定长度的空间。例如，一般微机使用的C系统为整形变量分配两个字节，为实型变量分配4个字节。内存区的每一个字节有一个编号，这就是“地址”，它相当于旅馆中的房间号。在地址所标志的内存单元中存放数据，这相当于旅馆中各个房间中居住旅客一样。

在程序中一般通过变量名对内存单元进行存取操作，这称作“直接访问”，还可以采用 另一种“间接访问”方式，将变量的地址存放在另一个变量中。所谓“指向”就是通过地址来体现的，由于通过地址能找到所需的变量单元，我们可以说，地址“指向”该变量单元，因此在C语言中，将地址形象化的称为“指针”。意思是通过它能找到以它为地址的内存单元。一个变量的地址成为该变量的“指针”。如果有一个变量专门用来存放另一个变量的地址，则称它为“指针变量”。

变量的指针和指向变量的指针变量

变量的指针就是变量的地址。存放变量地址的变量是指针变量，用来指向另一个变量。为了表示指针变量和它指向的变量之间的关系，用“*”符号表示“指向”。

定义指针变量的一般形式为：

*基类型　　*指针变量名;*

数组与指针

一个变量有地址，一个数组包含若干元素，每个数组元素都在内存中占有存储单元，它们都有相应的地址。指针变量也可以指向数组元素。

空间操作函数malloc、free

C语言提供了两个函数，malloc与free，分别用于执行动态内存分配与释放。这些函数维护一个可用内存池。当一个程序另外需要一些内存时，它就调用malloc函数从内存池中提取一块合适的内存，并返回一个指向这块内存的指针。这块内存此时并没有以任何方式进行初始化，使用时需手动初始化。

这两个函数的原型如下所示，它们都在头文件stdio.h中声明。

void *mallloc（size_t size）;

Void *free（viod* pointer）;

malloc的参数就是需要分配的内存字节数。如果内存池中的可用内存可以满足这个需要，malloc就返回一个指向被分配的内存块起始位置的指针，如果系统无法向malloc提供更多的内存，malloc就会返回一个NULL指针。

free的参数必须要么是NULL，要么是一个先前从malloc或其他空间申请函数返回的值。

malloc的一般用法

基类型=（基类型）*Malloc（数量*sizeof（基类型））;*

动态数组

动态数组是指在声明时没有确定数组大小的数组，即忽略方括号中的下标；当使用时可用malloc语句重新指出数组的大小。使用动态数组的优点是可以根据用户需要，有效利用存储空间。动态数组的内存空间是从堆上分配的。是通过执行代码而为其分配空间。当程序执行到这些语句时，才为其分配空间。程序员自己释放内存。

遵循原则：

申请的时候从外层往里层，逐层申请；

释放的时候从里层往外层，逐层释放。

【例2】：一维数组的动态开辟与释放:

```
int*i_Array;          //指向一维数组的指针
int i_Rows;           //数组个数
i_Array=(int*)malloc(i_Rows*sizeeof(int));
for (i_Count=0;i_Count<50;i_Count++)
{
    *(i_pTmpArray+i_Count)=i_Count;
}
free(i_Array);        //数组的释放
```

【例3】：二维数组的动态开辟：

```
int **i_Arry;            //指向二维数组的指针
int i_Rows;             //二维数组行
int i_Colums;           //二维数组列
printf("请输入行数与列数\n");
scanf("%d%d",&i_Rows,&i_Colums);
i_Arry=(int**)malloc(i_Rows*sizeof(int*));
for (int i=0;i<i_Rows;i++)
{
    i_Arry[i]=(int *)malloc(i_Colums*sizeof(int));

}

for (int i=0;i<i_Rows;i++)
{
    for (int j=0;j<i_Colums;j++)
    {
        i_Arry[i][j]=i+j;
    }
}

/***************二维数组的释放******************/
for (int i=0;i<i_Rows;i++)
{
   free(i_Arry[i]);
}
free(i_Arry);
```

数组

数组是构造类型，是一组具有相同类型数据的有序集合。每个数据成为数组的元素，用一个统一的数组名和下标来唯一地确定数组中的元素。

一维数组的声明方式为：

<类型标识符><数组名>[常量表达式]

类型标识符是任一种基本数据类型或构造数据类型；数组名由用户自定义，表示存储空间的地址；常量表达式表示数组元素的个数，也是数组的长度。

例：int a[6];表示一个整型、数组名为a、长度为6的一维数组。

（2）一维数组的引用形式：

下标法：*数组名[下标]*

例如：a[i];或p[i]；a为数组名，p为指向数组的指针变量。

注：C语言中不能依次引用整个数组，只能逐个引用数组中的各个元素。下标就是被访问的数组元素在所定义的数组中的相对位置。下标为0表示的是数组元素在数组的第一个位置上，下标等于1表示的是数组元素在数组的第二个位置上，依次类推。例如：

```
int a[10];
a[0]=100;//正确
a[10]=100;//不正确，下标越界
```

例如：

下标法：

```
int main()
{
   int a[10];
   for(int i=0;i<10;i++)
       a[i]=2*i;
   for(int i=0;i<10;i++)
   printf(“%d\t”,a[i]);
return 0;
}
```

指针法：*（a+i）或 *(p+i)；a是数组名，p为指向数组的指针变量。

例如：

指针法：

```
int main()
{
    int *p=a;
int i;
     for(i=0;i<10;i++)
        a[i]=2*i;
     for(i=0;i<10;i++)
     printf(“%d\t”,*（p+i））;
return 0;
```

}

二维数组的声明方式为：

<类型标识符><数组名>[常量表达式1][常量表达式2]：

二维数组与一维数组的区别在于多出[常量表达式2]。[常量表达式1] 是第一维，常称为行； [常量表达式2]是第二维，也就是列。

例：int a[3][5];表示一个3行5列的二维数组；数组元素的个数为：3*5=15个。

二维数组的引用形式：

下标法：

数组名[下标][下标]

注：二维数组在引用时和一维数组一样，只能逐个引用数组中的各个元素。例如：

sz_A[5][6]

下标可以是整数表达式，如sz_A[8-5][2*3-1]。不要写成sz_A[2,3]、sz_A[8-5,2*3-1]形式。

注意：严格区分定义数组时用的sz_A[5][6]和引用元素时用的sz_A[5][6]的区别。前者sz_A[5][6]用来定义数组的维数，后者sz_A[5][6]的5和6是下标，代表的是数组中的某一个元素。

【例2】分析程序的运行结果

源程序如下：

```
#include<stdio.h>
void main()
{
        int sz_Array[6];//一维数组
        int sz_DlArray[3][5];//二维数组
        int i_a;
int i_dla;
        int i_dlb;
//一维数组
        for (i_a=0;i_a<6;i_a++)
        {
sz_Array[i_a]=i_a*2+2;
        }
        printf("\n输出一维数组元素为：\n");
```

```
        for(i_a=0;i_a<6;i_a++)
        {
printf("%d\t",sz_Array[i_a]);
        }
//二维数组
        for(i_dla=0;i_dla<3;i_dla++)
        {
for(i_dlb=0;i_dlb<5;i_dlb++)
{
sz_DlArray[i_dla][i_dlb]=i_dla+i_dlb;}
            }
 printf("\n输出二维数组元素为：\n");
  for(i_dla=0;i_dla<3;i_dla++)
        {
for(i_dlb;i_dlb<5;i_dlb++)
{
               printf("%d  ",sz_DlArray[i_dla][i_dlb]);
            }
        printf("\n");
        }
}
```

运行结果为：

输出一维数组元素为：

2　4　6　8　10　12

输出二维数组元素为：

0　1　2　3　4

1　2　3　4　5

2　3　4　5　6

3　4　5　6　7

4　5　6　7　8

指针法：

可以通过行指针来引用二维数组元素。

定义行指针变量：int (*p)[3],指针p是指向一个由3个元素所组成的整型数组指针。

例如：

```
void main()
{
int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}}
    int (*p)[4]=a;
        for(int i=0;i<3;i++)
{
```

```
            for(int j=0;i<4;j++)
        printf(“%d\t”,p[i][j]);
    printf(“\n”);
  }
  }
```

参考书目：

1. C++语言程序设计教程与实验(第二版) 温秀梅 丁学钧 李建华主编
2. C/C++程序设计教程 张世民主编

字符数组

字符数组的定义与赋值

字符数组是一串字符的集合，其数组元素为字符型。

字符数组的赋值形式：

char 数组名[常量表达式]=“字符串”；

或 char 数组名[常量表达式]={“字符串”};

例：char sz_A[5]={‘s’, ‘t’, ‘u’, ‘d’, ‘y’, };

定义数组sz_A,包含5个元素，其在内存中的存放情况为：

sz_A[0]	sz_A[1]	sz_A[2]	sz_A[3]	sz_A[4]
s	t	u	d	y

则各元素赋值如下：

sz_A[0]=‘s’； sz_A[1]=‘t’； sz_A[2]=‘u’； sz_A[3]=‘d’；sz_A[4] =‘y’

如果花括号中的字符个数大于数组长度，编译系统就会报错，如果花括号中的字符个数小于数组长度，其余元素则由系统自动定义为空字符，即‘\0’。

‘\0’作为字符串的结束标志，因此在定义数组长度时，应在字符串原有的长度上加1，为字符串结束标志预留空间。

例：char sz_A[6]={‘s’, ‘t’, ‘u’, ‘d’, ‘y’, };

定义数组sz_A,包含6个元素，其在内存中的存放情况为：

sz_A[0]	sz_A[1]	sz_A[2]	sz_A[3]	sz_A[4]	sz_A[5]
s	t	u	d	y	\0

则各元素赋值如下：

sz_A[0]=‘s’； sz_A[1]=‘t’； sz_A[2]=‘u’

sz_A[3]=‘d’； sz_A[4]=‘y’； sz_A[5]=‘\0’

【例3】分析程序的运行结果

源程序如下：

```
#include<stdio.h>
void main()
{
    char sz_A[10]="work hard";
    int i;
    for(i=0;i<9;i++)
    printf("%c",sz_A[i]);
}
```

运行结果：

```
work hard
```

以上实例中，逐个显示字符数组的各个元素，但需注意的是在定义字符数组的下标时，至少比后面的字符串长度大1。其中，字符串长度应包括其中空格的长度。

字符串操作函数：

1. 字符串复制函数strcpy()

格式：strcpy（字符数组1，字符数组2）

功能：是将字符数组2中字符串复制到字符数组1中去。

注：字符数组1的长度必须大于字符数组2，从而能够容纳复制的字符数组2的字符串；字符数组1必须写成数组名形式，字符数组2既可以是字符数组名，也可以是字符串；字符数组之间不能相互赋值。

例如：
```
char sz_str1[10],sz_str2[6]="work hard";
Strcpy(sz_str1,sz_str2);
printf("%s\n",sz_str1);
```

运行结果：

```
work hard
```

2. 字符串连接函数strcat()

strcat(字符数组1，字符数组2)

功能：将字符数组1和字符数组2中的字符串连接起来,字符数组2中的字符串2接到字符数组1中的字符串后面。

注：字符数组1的长度必须足够大，能够同时容纳字符数组1中的字符串和字符数组2中的字符串。

字符数组名2中的字符串连接到字符数组1的字符串时，删除字符数组1中的字符串后面的标志‘\0’,只在新串的最后保留“\0”。

例如：
```
char sz_str1[10]="work",sz_str2[6]="hard";
strcat(sz_str1,sz_str2);
printf("%s\n",sz_str1);
```
运行结果：
```
workhard
```
3. 字符串比较函数strcmp()

strcmp（字符数组1，字符数组2）

功能：比较字符数组1和字符数组2中字符串，通过函数返回值得出比较结果。

若字符数组1中的字符串<若字符数组1中的字符串,函数返回值<0；

若字符数组1中的字符串>若字符数组1中的字符串,函数返回值>0；

若字符数组1中的字符串=若字符数组1中的字符串,函数返回值=0；

注：比较规则：比较过程中，按照从左到右的顺序，逐个比较字符的ASCII码值，直到遇到不相同的字符或“\0”，即结束比较。

例如：
```
char sz_str1[10]="work",sz_str2[10]="hard";
 if (strcmp(sz_str1,sz_str2)>0)
{
   printf("大于\n");
}
 if (strcmp(sz_str1,sz_str2)<0)
{
   printf("小于\n");
}
 if (strcmp(sz_str1,sz_str2)==0)
   printf("相等\n");
```
运行结果：
```
大于
```

4. sprintf()

sprintf（s,”%s%d%c”,”text”,1,’char’）;

将输出结果写入数组S中；其函数返回值为字符串长度，相当于strlen；计算长度时不计算“\0”,而sizeof计算时是加上“\0”的。

例如：

```
char sz_str;
sprintf(sz_str,"%s%d%c","work",6,"hard");
i_Tmp=sprintf(sz_str,"%s%d%c","work",6,"hard");
printf("sprintf结果输出：%s\n",sz_str);
printf("sprintf函数返回值输出：%d\n",i_Tmp);
```

5. sscanf()

sscanf（s,”%d%f%s”,&a,&b,&c）;

从一个字符串中读进与指定格式相同的数据；其返回值为读入有效数据的个数；从数组S中，以固定格式向a,b,c输入，sscanf不识别空格。

```
char sz_str[11]="work6hard";
char sz_str1[11];
    sscanf(sz_str,"%4s",sz_str1);
    i_Tmp=sscanf(sz_str,"%4s",sz_str1);
    printf("输出结果:%s\n",sz_str1);
    printf("sscanf函数返回值：%d\n",i_Tmp);
  运行结果：
  输出结果：work
  sscanf函数返回值：1
```

全局变量与局部变量的定义和区别

变量的作用域指变量所起的作用范围，变量的定义位置决定了变量的作用域，它可分为全局变量和局部变量。

全局变量

全局变量是指在函数外部定义的变量，它不属于哪一个函数，它属于一个源程序文件，它的作用范围从定义变量定义的位置开始到本源文件的结束。在函数中使用全局变量，一般应做全局变量声明。只有在函数内部经过声明的全局变量才能使用。但在一个函数之前定义全局变量，在该函数内使用可不加以声明。

全局变量的作用是增加函数间数据联系的渠道。同一个文件中的所有函数都能引用全局变量的值，当一个函数改变了全局变量的值，就会影响到其它函数，有利于函数之间信息的传递。

局部变量

在函数中或者复合语句中定义的变量称为局部变量，它的作用范围只限于该函数或者该复合语句中，在其它位置无效。

如果全局变量和局部变量重名，在定义局部变量中的子程序中局部变量起作用，全局变量失效。

程序在编译过程中，系统并不会为局部变量分配存储单元，而在程序的运行过程中，当局部变量所在函数被调用时，系统才会为变量分配临时内存，函数调用结束后，释放空间。

全局变量和局部变量的区别

1 全局变量的有效范围从定义该变量的位置开始到本源文件的结束，局部变量只在定义该变量的函数中有效，在函数外部无效。

2局部变量是程序运行到该函数时给变量分配内存空间，函数运行结束后释放空间，全局变量在程序运行时先分配内存空间，直到本源文件执行完以后释放空间。

程序实例

```
#include "stdafx.h"
int x=6,y=8;
void plus()
{
        printf("x=%d\t,y=%d\n",x,y);
        x++;
        y++;
}
void main()
{
        void plus();
        int x=2,y;
        x=2,y=3;
        printf("x=%d\t,y=%d\n",x,y);
        plus();
        printf("x=%d\t,y=%d\n",x,y);
        plus();
```

}

参考书目：

1.《C 语言程序设计》作者 张书云 姜淑菊 朱雷 P89-P91

文件操作以处理

FILE类型

文件：存储在外部介质上数据的集合。声明FILE结构体类型的信息须声明#inclu<stdio.h>议案不能定义指向FILE类型变量的 指针变量FILE *fp；

fopen函数

调用方式：fopen（文件名，使用文件方式），文件打开成功则返回一个FILE类型指针，否则返回NULL。

表 01文件打开方式

文件使用方式	含义
“r”（只读）	为输入打开一个文本文件
“w” （只写）	为输出打开一个文本文件
“a” （追加）	向文本文件尾部增加数据
“rb” （只读）	为输入打开一个二进制文件
“wb” （只写）	为输入打开一个二进制文件
“ab” （追加）	向二进制文件尾增加数据
“r+” （读写）	为读/写打开一个文本文件
“w+” （读写）	为读/写建立一个新的文本文件
“a+” （读写）	为读/写打开一个文本文件
“rb+” （读写）	为读/写打开一个二进制文件
“wb+” （读写）	为读/写建立一个新的二进制文件
“ab+” （读写）	为读/写打开一个二进制文件

```
fopen("a1","r");
```

FILE *fp；fp=fopen("a1","r");

将fopen函数的返回值赋给指针变量fp。

常用下面方法打开一个文件：

```
if((fp=fopen("file1","r"))==NULL)
{
printf("cannot open this file\n");
exit(0);
}
```

fclose函数（文件的关闭）

关闭　fclose(文件指针)；当顺利关闭文件，则返回0值，否则返回EOF(-1).

fprintf函数与fscanf函数

fprintf函数、fscanf函数与printf函数、scanf函数相仿都是格式化读写函数。只有一点不同:fprintf和fscanf函数的读写对象不是终端而是磁盘文件。它们的一般调用方式为：

fprintf(文件指针，格式化字符串，输出列表)；

fscanf(文件指针，格式化字符串，输入列表)；

例如：

fprintf(fp,”%d,%6.2f”,I,t);

它的作用是将变量i与t的值按%d与%6.2f的格式输出到fp指向的文件上。

用fprintf函数和fscanf函数对磁盘文件读写，使用方便，容易理解，但由于在输入时要将ASCII码转换为二进制形式，在输出时又要讲二进制形式转换成字符，花费时间较多。因此，在内存与磁盘频繁交换数据的情况下，最好不要使用fprintf和fscanf函数。

fread与fwrite函数

ANSI C标准提出设置两个函数（fread与fwrite），用来读写一个数据块，它们的一般调用形式为

fread（buffer，size，count，fp）；

fwrite（buffer，size，count，fp）；

其中：

buffer：是一个指针。对于fread来说，它是读入数据的存放地址。对fwrite来说，是要输出的数据的地址。

Size：要读写的字节数。

Count：要进行读写多少个size字节的数据项。

fp：文件型指针。

如果fread与fwrite调用成功，则函数返回值为count的值，即输入输出数据项的完整个数。

常用fread与fwrite函数进行文件的读写操作

例:

冒泡排序与折半排序

冒泡排序

冒泡排序（Bubble Sort），是一种计算机科学领域的较简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。

冒泡排序算法的运作如下：（从后往前）

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

实例：

```
#include<stdio.h>
#defineSIZE8
Void Bubble_Sort(int a[],int n);


Void Bubble_Sort(int a[],int n) //n为数组a的元素个数
{
    int i,j,temp;
    for(j=0;j<n-1;j++)
        for(i=0;i<n-1-j;i++)
        {
            if(a[i]>a[i+1])//数组元素大小按升序排列
            {
                temp=a[i];
                a[i]=a[i+1];
                a[i+1]=temp;
            }
        }
}
int main()
```

```
{
    int i_Num[SIZE]={95,45,15,78,84,51,24,12};
    int i;
    Bubble_Sort(i_Num,SIZE);
    for(i=0;i<SIZE;i++)
    {
        printf("%d", i_Num [i]);
    }
    printf("\n");
}
```

折半排序

折半排序实质上是不断地对有序数据集进行对半分割，并且获取每个分区的中间元素，用中间元素与待排序元素进行比较，最终确定数据集顺序。

折半排序算法思想：折半排序的重点是查找待排序元素A[i]在有序数据集中的位置。在处理A[i]时，A[0]……A[i-1]已经按关键码值排好序。所谓折半比较，就是在插入A[i]时，取A[i-1/2]的关键码值与A[i]的关键码值进行比较，如果A[i]的关键码值小于A[i-1/2]的关键码值，则说明A[i]只能插入A[0]到A[i-1/2]之间，故可以在A[0]到A[i-1/2-1]之间继续使用折半比较；否则只能插入A[i-1/2]到A[i-1]之间，故可以在A[i-1/2+1]到A[i-1]之间继续使用折半比较。如此重复，直到最后能够确定插入的位置为止。

算法的基本过程：

1. 计算 0 ~ i-1 的中间点，用 i 索引处的元素与中间值进行比较，如果 i 索引处的元素大，说明要插入的这个元素应该在中间值和刚加入i索引之间，反之，就是在刚开始的位置 到中间值的位置，这样很简单的完成了折半；
2. 在相应的半个范围里面找插入的位置时，不断的用（1）步骤缩小范围，不停的折半，范围依次缩小为 1/2 1/4 1/8 .....快速的确定出第 i 个元素要插在什么地方；
3. 确定位置之后，将整个序列后移，并将元素插入到相应位置；
4. 对形成的新的有序序列再次进行插入，重复步骤a），b），c），直到所有数据完成排序。

数组实例：

```
void BisSort(int p_Num[],int i_Len)
{
	int i_Left,i_Right,i_Middle;
	int i_Loop,i_Subloop;
	int i_Buffer;//临时变量
	if (p_Num[0]>p_Num[1])
	{
		i_Buffer=p_Num[0];
		p_Num[0]=p_Num[1];
		p_Num[1]=i_Buffer;
	}
	for (i_Loop=2;i_Loop<i_Len;i_Loop++)
	{
		i_Left=0;
		i_Right=i_Loop-1;
		i_Buffer=p_Num[i_Loop];
		if (i_Buffer<=p_Num[0]||i_Buffer>=p_Num[i_Right])
		{
			if (i_Buffer<=p_Num[0])
				i_Middle=0;
			else
				i_Middle=i_Right+1;
		}
		else
			while(i_Left<=i_Right)
			{
				i_Middle=(i_Left+i_Right)/2;//折半中数
				if(i_Buffer>=p_Num[i_Middle])
					i_Left=i_Middle+1;
				else
					i_Right=i_Middle-1;
			}                                                    //确定要插入
的位置
		for (i_Subloop=i_Loop;i_Subloop>i_Middle;i_Subloop--)//生成新的有序数组
		{
			p_Num[i_Subloop]=p_Num[i_Subloop-1];
		}
		p_Num[i_Middle]=i_Buffer;
	}
```

```
}
```

链表实例：

```
struct _Road* RoadOrder(struct _Road *pst_Road )
{
	struct _Road *pst_Road_Head;
	pst_Road_Head=pst_Road;
	struct _Road *p_Low,*p_Hight,*p_Mid,*p_Current;
	p_Low=pst_Road;
	p_Hight=pst_Road->pst_Next;
	int i_Low,i_Hight,i_Mid;//中数
	int i_Len=RoadListLength(pst_Road);
	if (pst_Road_Head->i_GeoLength>p_Hight->i_GeoLength)
	{
		pst_Road_Head->pst_Next=p_Hight->pst_Next;
		p_Hight->pst_Next=pst_Road_Head;
		pst_Road_Head=p_Hight;
	}
	for (int i=3;i<=i_Len;i++)
	{
		p_Low=p_Mid=p_Hight=pst_Road_Head;
		i_Low=1,i_Hight=i-1;
		for(int j=1;j<i-1;j++)
		{
			p_Hight=p_Hight->pst_Next;
		}
		p_Current=p_Hight->pst_Next;
		//如果小于头指针数据
		if (p_Current->i_GeoLength<=p_Low->i_GeoLength)
		{
			p_Hight->pst_Next=p_Current->pst_Next;
			p_Current->pst_Next=pst_Road_Head;
			pst_Road_Head=p_Current;
		}
		else if (p_Current->i_GeoLength<p_Hight->i_GeoLength)
		{
			while(i_Low<=i_Hight)
			{
				i_Mid=(i_Low+i_Hight)/2;
				for(int j=i_Low;j<i_Mid;j++)
```

```
                        {
                                p_Mid=p_Mid->pst_Next;
                        }

                        if (p_Current->i_GeoLength>p_Mid->i_GeoLength)
                        {
                                i_Low=i_Mid+1;
                        }
                        else
                        {
                                i_Hight=i_Mid-1;
                        }
                }//确定插入位置

                //插入数据
                for (i=1;i<i_Mid-1;i++)
                {
                        p_Low=p_Low->pst_Next;
                }
                p_Hight->pst_Next=p_Current->pst_Next;
                p_Current->pst_Next=p_Mid;
                p_Low->pst_Next=p_Current;
        }
    }
    return  pst_Road_Head;
}
```

参考文献：Kyle Loudon《算法精解》260-270

单链表的使用

单链表的基础知识

线性表（Chain）的连接存储结构成为单链表。为了能够正确的表示链表中元素之间的逻辑关系，每一个存储单元在存储数据元素的同时，还必须存储其后继元素所在的地址信息，这个地址信息成为指针。这两个部分组成了数据元素的存储映像，成为结点（ChainNode）。结构如下：

图1

其中，data为数据域，用来存放数据元素；next为指针域，用来存放该结点的后继结点的地址。如下图所示，为单链表的存储示意图。从图中可以看出，除了最开始的结点，其他每个结点的存储地址都存放在其前驱结点的next域中。

图：链表有一个“头指针”变量，图中以head表示，它存放一个地址。该地址指向一个元素。可以看出，head指向第一个元素，第一个元素又指向第二个元素・・・直到指向最后一个元素，该元素不再指向其他元素，它称为“表尾”，它的地址部分存放一个“NULL”（表示“空地址”），链表到此结束。

图2

单链表的实现

单链表上进行定位，链表结点的插入、删除操作算法。

a. 定位

在链表中查找出第i个结点，若存在则返回该结点的地址，否则就返回表头结点的地址。

操作步骤：定义整型变量i_Temp和指针变量p，i_Temp初值为0，p指向表的头结点；判断i_Temp<i的时候，令p指向县一个结点，同时i_Temp加1；反复循环这个过程。

b. 插入

在第i个结点后面插入一个数值为x的新结点，用i=0表示将新结点插在表头结点之后，成为线性表的第一个结点。

操作步骤：首先，判断参数i是否正确，若i<0或者i的长度超过链表长度，则说明i不正确，无法进行插入操作；否则，先进行定位，确定第i个结点的地址p，在p结点的后面插入一个新结点：

为新节点分配存储单元

将x存入新结点的数值字段中

将p结点指针字段的值存入新结点的指针字段中

将新结点地址存入p结点的指针字段中

令线性表的长度值加1.

c. 删除

删除线性表中的第i个结点

操作步骤：首先，判断参数i是否正确，若i<0或者i的长度超过链表长度，则说明i不正确，无法进行删除操作；否则，调用定位算法，令p指向第i个结点的前面一个结点，删除p节点的下一个结点：

将p结点指针字段的值存入变量q中（其中，q指向待删除的结点）

将q结点指针字段的值存入p结点的指针字段中

释放q结点所占的存储单元

令线性表长度减1.

例子：要求建立一个有3个学生的数据的单向动态链表，同时可以进行插入，删除。

代码如下：

```
/***********定义一个用于单链表的结构体***************/
struct student
{
        long l_Num;
        float f_Score;
        struct student *next;
};
/*****************单链表创建************************/
struct student *creat(void)  //创建链表，此函数带回一个指向链表头的指针
{
        struct student *head;
        struct student *stu1,*stu2;
    i_Count=0;
        stu1=stu2=(struct student *)malloc(LEN);//开辟一个新单元
        scanf("%ld,%f",&stu1->l_Num,&stu1->f_Score);
        head=NULL;
        while(stu1->l_Num!=0)
        {
                i_Count=i_Count+1;
                if(i_Count==1)head=stu1;
                else stu2->next=stu1;
```

```
                stu2=stu1;
                stu1=(struct student *)malloc(LEN);
                scanf("%d,%f",&stu1->l_Num,&stu1->f_Score);
        }
        stu2->next=NULL;
        return(head);
};
/*****************输出单表链函数**********************/
void print(struct student *head)
{
        struct student *stu;
        printf("\nNow,These %d records are :\n",i_Count);
        stu=head;
        if(head!=NULL)
        {
                do
                {
                        printf("%ld%5.1f\n",stu->l_Num,stu->f_Score);
                        stu=stu->next;
                } while (stu!=NULL);
        }
}
/*****************清空单表链函数**********************/
void Erase(struct student *head)
{
        struct student *stu1,*stu2;
        stu1=head;
        while(stu1)
        {
                stu2=stu1->next;
                delete stu1;
                stu1=stu2;
        }
}
/*****************删除单链表节点函数**********************/
struct student *Del(struct student *head,long Del_Num) //删除节点
{
        struct student *stu1,*stu2;
        if (head==NULL)
        {
```

```
        printf("\nlist null!\n");/*goto end;*/
    }
    stu1=head;
    while (Del_Num!=stu1->l_Num&&stu1->next!=NULL)//p1指向的不是所要找的结点，并且后面还有结点
    {
        stu2=stu1;
        stu1=stu1->next;
    };
    if (Del_Num==stu1->l_Num)//找到了
    {
        if (stu1==head)// p1指向的首结点，把第二个结点地址赋值给head
        {
            head=stu1->next;
        }
        else
            stu2->next=stu1->next;
        printf("delete:%ld\n",Del_Num);
        i_Count=i_Count-1;
    }
    else
        printf("%ld not been found! \n",Del_Num);//招不到该结点
    return(head);
}

/*****************插入单链表节点函数(按照学号顺序排序输入的，存在很大的局限性！！！）**********************/
struct student *Insert(struct student *head,struct student *stud)  //插入操作          （书上插入的代码有问题！！）
{
    struct student *stu0,*stu1,*stu2;
    stu1=head;
    stu0=stud; //p0指向要插入的结点
    if (head==NULL)
    {
        head=stu0;stu0->next=NULL;
    }//如果原来链表是空的
    else
    {
```

```
                while ((stu0->l_Num>stu1->l_Num)&&(stu1->next!=NULL))
                {
                        stu2=stu1;  //p2指向刚才p1指向的结点
                        stu1=stu1->next; //p1向后移动一个
                }
                if (stu0->l_Num<=stu1->l_Num)
                {
                        if (head==stu1) //插到第一个结点之前
                        {
                                head=stu0;
                        }
                        else stu2->next=stu0; //插到p2指向的结点之后
                }
                else
                {
                        stu1->next=stu0;
                        stu0->next=NULL;
                }
        }
        i_Count=i_Count+1;
        return(head);
}
/*****************单链表技术***********************/
void Case_List()
{
        struct student *head,stu;
        long Del_Num;
        int i_Choice;
        char c_temp;
        printf("%s","第十三层--链表技术：\n");
        printf("请创建单链表！\n");
        printf("请您输入信息，输入0表示结束！\n");
        head=creat();
        print(head);
        fflush(stdin);
        printf("输入成功！\n");
        printf("1、删除结点 2、插入节点 3、清空链表 。 请输入您的选择：\n");
        c_temp=getchar();
        i_Choice=atoi(&c_temp);
```

```
        while (i_Choice!=4)
        {
                switch(i_Choice)
                {
                        case 1:
                                fflush(stdin);
                                printf("请输入您要删除的学号：");
                                scanf("%ld",&Del_Num);
                                head=Del(head,Del_Num);
                                print(head);
                                fflush(stdin);
                                break;
                        case 2:
                                fflush(stdin);
                                printf("请输入您要插入的学生信息：");
                                scanf("%ld,%f",&stu.l_Num,&stu.f_Score);
                                head=Insert(head,&stu);
                                print(head);
                                fflush(stdin);
                                break;
                        case 3:
                                fflush(stdin);
                                Erase(head);
                                printf("链表清空成功！");
                                break;
                        default:
                                fflush(stdin);
                                printf("意外的输入！！");
                }
                c_temp=getchar();
                i_Choice=atoi(&c_temp);
                fflush(stdin);
        }
}
```

双向链表

在双链表中，每一个结点都包含两个指针字段，分别用于存放前驱结点地址和后继结点地址。

与单链表相比，双向链表具有更加灵活的优点，从表中的任何一个结点出发，既可以顺着后继指针链往后查找，又可以顺着前驱指针链往前查找。

在双向链表上进行插入和删除操作室，需要同时修改两个方向上的指针。假设采用动态存储的结构，prior表示前驱指针，next表示后驱指针。

插入

在结点p的后面插入一个新的结点s，需要修改4个指针：

```
s->prior-p;
s->next=p->next;
p->next->prior=s;
p->next=s;
```

注：这里要注意指针的修改顺序。在修改第二、第三步的指针时，要用到p->next来找到p的后继结点，所以第四步指针的修改要在第二、第三步的指针修改完成后才能进行。

删除

设指针p指向待删除结点，删除操作可通过以下语句完成：

```
(p->prior)->next=p->next;
(p->next)->prior=p->prior;
```

注：这两个语句顺序是可以颠倒的。虽然在执行上述语句后结点p的两个指针域扔指向前驱结点和后继结点，但是在双链表中已经找不到结点p，而且，执行完删除操作以后，还要将结点p所占的存储空间释放。

struct,union和预处理命令的概念及使用

struct概念及使用

数据经常以成组的形式存在。例如学校必须知道每个学生的基本情况，包括姓名，学号，出生日期等信息。如果这些值能够存储在一起，访问时会简单一些。但是，如果这些值得类型不同，他们就无法存储在同一个数组中。在C中，就会使用结构把不同类型的值存储在一起。

结构是一种聚合数据类型，也是一些值得集合，这些值称为结构的成员，但一个结构的各个成员可能具有不同的类型。

结构定义：

```
struct TypeName
{
        int a;
        char b;
        float c;
        …
};
```

关键字struct引出了结构的定义。标识符TypeName是结构“标记”，命名了结构的定义，与关键字struct一起使用可声明结构类型的变量，在定义结构体花括号内部声明的变量是结构的成员。同一结构体的成员不能使用相同的名字，但是不同结构可包含同名的成员而不会发生冲突。此外，每个结构定义都要以分号结束。常见的程序设计错误都是忘记结构定义的分号。

编译器不为上述结构定义保留任何内存空间，而是建立了用于声明变量的一种新的数据的类型。声明结构变量与声明其他类型变量类似。

声明语句：

```
struct TypeName  str_Stu,*p_Stu;
```

把str_Stu声明为struct TypeName类型的变量，p_Stu是指向struct TypeName的指针。

此外，也可以用下述方法声明给定结构类型的变量：在结构定义的花括号之后用逗号隔开变量名列表，再在最后加上结束结构定义的分号。例如：

```
struct TypeName
{
        int a;
        char b;
        float c;
        …
} str_Stu,*p_Stu;
```

如果结构定义中没有包含结构标记名，那么该结构类型只能在定义结构时声明，而不能单独声明。我们在进行代码设计时，最好在建立结构类型时提供结构标记名，以后可以方便地用结构标记名声明该结构类型的新的变量。

当然我们也可以采用typedef来方便的标记结构类型，例如：

```
Typedef struct TypeName
{
        int a;
        char b;
        float c;
        …
} TypeName;
```

我们在声明该结构体变量时，就可以直接写为：

```
TypeName  str_Stu,*p_Stu;
```

结构的初始化

和数组一样，结构也可以用初始化值列表初始化，即在声明结构变量时，在变量名后用等号连接在花括号中的初始化值列表来初始化该结构变量，初始化值用逗号分开。例如：

声明语句：

```
struct TypeName  str_Stu={10,  ‘a’,1.2};
```

建立了类型为struct TypeName的变量str_Stu，并将成员a,b,c分别初始化为10,a,1.2。如果初始化的个数少于结构中成员变量的数目，剩余的成员被自动初始化为0。

union概念及使用

和结构一样，联合也是一种聚合数据类型，但是其成员共享了同一个存储空间。程序中的变量无非是两种情况：某些变量是相关的，某些是不相关的。联合是用来使相关变量共享存储空间而不是把空间浪费给不使用的变量。联合的成员可以使任何数据类型。用来存储联合的字节至少能够足以存储最大成员。多数情况下，联合包含了两种或多种数据类型。同一时候只能引用一个成员。

联合是用关键字union声明的，其格式与结构的声明是一样的。

声明语句：

```
union Num{
        int x;
        float y;
};
```

表示Num是具有成语int x和float y的union类型。联合通常定义在程序的main函数之前，因而能够被程序中所有函数用来声明变量。

和结构的声明一样，联合的声明仅仅是建立了一种新的数据类型，把联合和结构的声明放在所有函数之外不能建立全局变量。

注意：

在声明语句中，联合只能用与第一个成员具有相同类型的值初始化。例如，声明语句：

union Num value={10};

因为是用int类型的值初始化联合变量value，所以是正确的。下面声明是非法的：

union Num value={1.2}; //非法语句

预处理程序宏等的概念及使用

1. 宏的概念及使用

宏是预处理指令#define中定义的一种操纵。和符号常量一样，程序中的宏标识符也在编译之前被文本取代。可以定义带有或不带有参数的宏。预处理就像处理符号常量一样处理不带参数的宏。对带有参数的宏处理方式是：先用替换文本取代参数，然后再把宏展开，即用替换文本取代程序中的标识符和参数列表。例如，定义一个带有求圆面积参数的宏：

#define CIRCLE_AREA(x) (PI(x)*(x))*

不论文件中何时出现CIRCLE_AREA(x)，替换文本中的x都会用x的值取代，符号常量PI是定义的符号常量。例如，语句

Area= CIRCLE_AREA(4);

被展开为：

Area= (3.14(4)*(4))*

当然我们也可以把求圆面积写成一个函数，但是函数需要函数调用的开销，而是用宏则可以直接把代码插入到程序中，并且保持了程序的可读性，但是，宏定义也有一个缺点，就是函数会被调用多次。

但是如果这个宏参数是一个函数，那么就有可能被调用多次从而达到不一致的结果，甚至会发生更严重的错误。比如：

```
#define min(X,Y) ((X) > (Y) ? (Y) : (X))
//...
c = min(a,foo(b));
```

这时foo()函数就被调用了两次。为了解决这个潜在的问题，我们应当这样写min(X,Y)这个宏：

```
 #define min(X,Y)
({typeof (X) x_ = (X);
typeof (Y) y_ = (Y);
(x_ < y_) ? x_ : y_; })
```

({...})的作用是将内部的几条语句中最后一条的值返回，它也允许在内部声明变量（因为它通过大括号组成了一个局部Scope）。

2. 条件编译

条件编译能够让程序员控制预处理智力的执行程序和程序代码的编译，每一个条件预处理指令都计算一个整常数表达式的值。不能在预处理指令中计算强制类型转换表达式、sizeof表达式和枚举类型常量。

条件编译的结构与if选择结构非常相似。以以下预处理代码为例：

```
#if !defined(NULL)
        #defined NULL 0
#endif
```

这些预处理指令确定是否定义了NULL。如果定义了NULL，表达式defined（NULL）的计算结果为1，否则为0。如果计算结果为0，那么!defined(NULL)结果为1，从而定义NULL为0，否则就跳过#defined指令。每一个#if结构都是用#endif结束的。在使用时，可以把#if defined（）和#if ! defined（）缩写为#ifdef和#ifndef。可以用#elif(等价于 else if)和#else（等价于else）指令测试包含多个部分的条件预处理结构。

3. 特殊符号#、##

1. #的使用

在一个宏中的参数前面使用一个#,预处理器会把这个参数转换为一个字符数组

简化理解：#是“字符串化”的意思，出现在宏定义中的#是把跟在后面的参数转换成一个字符串。

#define ERROR_LOG(module)

fprintf(stderr,"error: "#module"\n")ERROR_LOG("add");

转换为：*fprintf(stderr,"error: "add"\n");*

ERROR_LOG(devied =0);

转换为：*fprintf(stderr,"error: devied=0\n");*

2. ##的使用

“##”是一种分隔连接方式，它的作用是先分隔，然后进行强制连接。在普通的宏定义中，预处理器一般把空格解释成分段标志，对于每一段和前面比较，相同的就被替换。但是这样做的结果是，被替换段之间存在一些空格。如果我们不希望出现这些空格，就可以通过添加一些##来替代空格。

1 #define TYPE1(type,name) type name_##type##_type

2 #define TYPE2(type,name) type name##_##type##_typeTYPE1(int, c);

转换为：*int name_int_type ;*

(因为##号将后面分为 name_ 、type 、 _type三组，替换后强制连接)

TYPE2(int, d);

转换为： *int d_int_type ;*

(因为##号将后面分为 name、_、type 、_type四组，替换后强制连接)

参考文献：

1. H.M.Deitel P.J.Deitel等《how to program Second Edition》415-420

书名 《C++程序设计》

作者 谭浩强

P64-P71

1. 函数的使用

在程序中，通过函数我们可以实现特定的功能，因此对函数的定义，函数的返回值以及函数参数的理解对我们编写程序十分重要。

3.1函数定义的一般形式：

返回值类型 函数名（参数…..）

{

函数体；

}

函数的返回值有void(无返回值),*p(指针),int(整形) ….

例：

```
void  Change (int a,int b)        函数返回值   函数名  （参数列表）
{
    if(a>=b)
    printf(“%d”,a);                函数体
 else
    Printf(“%d”,b);
}
```

3.2 函数参数的形式

函数参数传递形式有参数的值传递，参数的指针传递和参数的引用传递。

3.2.1 参数的值传递

参数的值传递只是将数据拷贝一份进入函数体，但是进入函数体的数据源并未发生变化。

例：通过值传递交换两个数的值

```
void change1(int a,int b)
{
        int i_changetmp;
        i_changetmp=a;
        a=b;
        b=i_changetmp;
        printf("a=%d\t",a);
        printf("b=%d\n",b);
}
void main()
{
        int i,j;
        printf("请输入待交换的数:\n");
        scanf("%d%d",&i,&j);
        change1(i,j);
        printf("i=%d\t",i);
        printf("j=%d\n",j);
}
```

通过结果我们可以看出，通过值传递并没有对两个数字进行交换，原因是在执行交换函数时，只是对待交换的数据进行了复制，并没有对数据本身进行操作，因此，无法交换两个数据的值。

3.2.2 函数参数的指针传递

函数参数的指针传递是直接对函数参数的地址进行操作，即对数据源的操作。

例：通过指针交换两个数据

```
void change1(int *a,int *b)
{
        int i_changetmp;
        i_changetmp=*a;
        *a=*b;
        *b=i_changetmp;
        printf("a=%d\t",*a);
        printf("b=%d\n",*b);
```

```
}
void main()
{
        int i,j;
        int *p_i,*p_j;
        printf("请输入待交换的数:\n ");
        scanf("%d%d",&i,&j);
        p_i=&i;
        p_j=&j;
        change1(p_i,p_j);
        printf("i=%d\t",i);
        printf("j=%d\n",j);
}
```