

局部变量和全局变量

每一个变量都有一个作用域问题，即它们在什么范围内有效。

局部变量

定义变量可能有3种情况:

- (1) 在函数的开头定义;
- (2) 在函数内的复合语句内定义;
- (3) 在函数的外部定义。

在一个函数内部定义的变量只在本函数范围内有效，也就是说只有在本函数内才能引用它们，在此函数以外是不能使用这些变量的。在复合语句内定义的变量只在本复合语句范围内有效，只有在本复合语句内才能引用它们。在该复合语句以外是不能使用这些变量的，以上这些称为“**局部变量**”。

局部变量

```
float f1(int a)
```

```
//定义函数f1
```

```
{int b,c;
```

```
//在函数f1中定义b,c
```

```
    :
```

```
} a, b, c有效
```

```
}
```

```
char f2(int x,int y)
```

```
//定义函数f2
```

```
{int i,j;
```

```
    :
```

```
} x, y, i, j有效
```

```
}
```

```
int main()
```

```
//主函数
```

```
{int m,n;
```

```
    :
```

```
} m, n有效
```

```
return 0;
```

```
}
```

(1) 主函数中定义的变量也只在主函数中有效。主函数也不能使用其他函数中定义的变量。

(2) 不同函数中可以使用同名的变量，它们代表不同的对象，互不干扰。

(3) 形式参数也是局部变量。只在定义它的函数中有效。其他函数中不能直接引用形参。

(4) 在一个函数内部，可以在复合语句中定义变量，这些变量只在本复合语句中有效，这种复合语句也称为“分程序”或“程序块”。

```
int main ()
```

```
{    int a,b;
```

```
    :
```

```
    {    int c;
```

```
        c=a+b;
```

```
        :
```

```
    }
```

```
    :
```

```
}
```

```
} c在此复合语句内有效 a,b在此范围内有效
```

全局变量

程序的编译单位是源程序文件,一个源文件可以包含一个或若干个函数。在函数内定义的变量是局部变量,而在函数之外定义的变量称为**外部变量**,外部变量是**全局变量**(也称全程变量)。全局变量可以为本文件中其他函数所共用。它的有效范围为从定义变量的位置开始到本源文件结束。

注意

- 在函数内定义的变量是局部变量，在函数外定义的变量是全局变量。

全局变量

```
int p=1,q=5;           //定义外部变量
float f1(int a)         //定义函数f1
{
    int b,c;           //定义局部变量
    :
}
char c1,c2;            //定义外部变量
char f2 (int x, int y) //定义函数f2
{
    int i,j;
    :
}
int main()             //主函数
{
    int m,n;
    :
    return 0;
}
```

全局变量p,q的作用范围

全局变量c1,c2的作用范围

设置全局变量的作用是增加了函数间数据联系的渠道。由于同一文件中的所有函数都能引用全局变量的值，因此如果在一个函数中改变了全局变量的值，就能影响到其他函数中全局变量的值。相当于各个函数间有直接的传递通道。由于函数的调用只能带回一个函数返回值，因此有时可以利用全局变量来增加函数间的联系渠道，通过函数调用能得到一个以上的值。

*为了便于区别全局变量和局部变量，在C程序设计人员中有一个习惯（但非规定），将全局变量名的第1个字母用大写表示。

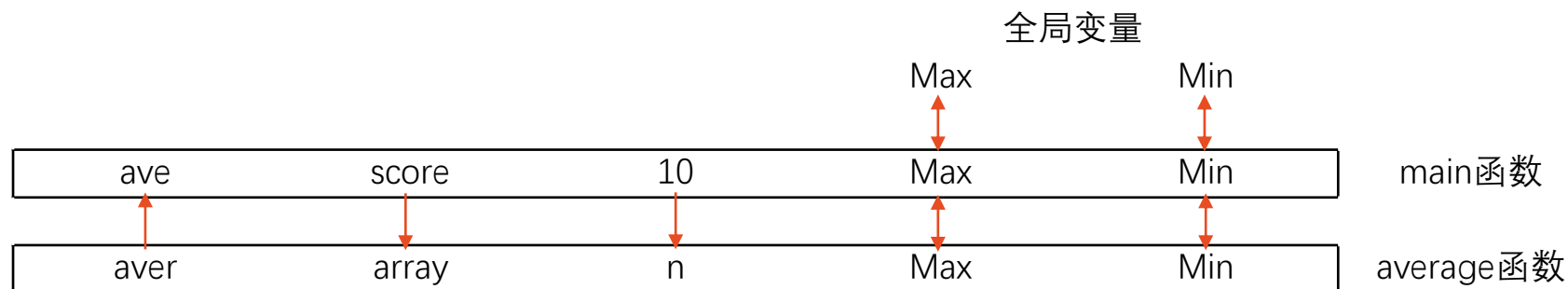
全局变量

【例7.14】有一个一维数组，内放10个学生成绩，写一个函数，当主函数调用此函数后，能求出平均分、最高分和最低分。

```
C:\WINDOWS\system32\cmd.exe
Please enter 10 scores:89 95 87.5 100 67.5 97 59 84 73 90
max=100.00
min= 59.00
average= 84.20
请按任意键继续. . .
```

<pre>#include <stdio.h> float Max=0,Min=0; //定义全局变量Max,Min int main() { float average(float array[],int n); float ave,score[10]; int i; printf("Please enter 10 scores:"); for(i=0;i<10;i++) scanf("%f",&score[i]); ave=average(score,10); printf("max=%6.2f\nmin=%6.2f\naverage=%6.2f\n",Max,Min,ave); return 0; }</pre>	<pre>float average(float array[],int n) //定义函数，有一形参是数组 { int i; float aver,sum=array[0]; Max=Min=array[0]; for(i=1;i<n;i++) { if(array[i]>Max) Max=array[i]; else if(array[i]<Min) Min=array[i]; sum=sum+array[i]; } aver=sum/n; return(aver); }</pre>
---	---

变量的关系：



全局变量

但是，建议不在必要时不要使用全局变量，原因如下：

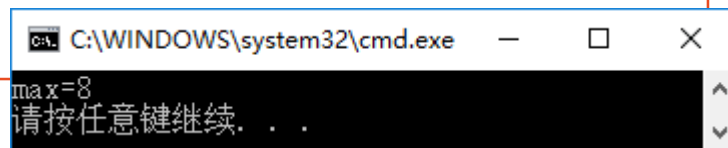
- ① 全局变量在程序的全部执行过程中都占用存储单元，而不是仅在需要时才开辟单元。
- ② 它使函数的通用性降低了，因为如果在函数中引用了全局变量，那么执行情况会受到有关的外部变量的影响，如果将一个函数移到另一个文件中，还要考虑把有关的外部变量及其值一起移过去。但是若该外部变量与其他文件的变量同名时，就会出现冲突。这就降低了程序的可靠性和通用性。在程序设计中，在划分模块时要求模块的“内聚性”强、与其他模块的“耦合性”弱。即模块的功能要单一（不要把许多互不相干的功能放到一个模块中），与其他模块的相互影响要尽量少，而用全局变量是不符合这个原则的。一般要求把C程序中的函数做成一个相对的封闭体，除了可以通过“实参—形参”的渠道与外界发生联系外，没有其他渠道。这样的程序移植性好，可读性强。
- ③ 使用全局变量过多，会降低程序的清晰性，人们往往难以清楚地判断出每个瞬时各个外部变量的值。由于在各个函数执行时都可能改变外部变量的值，程序容易出错。因此，要限制使用全局变量。

全局变量

【例7.15】若外部变量与局部变量同名，分析结果。

```
#include <stdio.h>
int a=3,b=5;           //a,b是全局变量
int main()
{
    int max(int a,int b); //函数声明。a,b是形参
    int a=8;             //a是局部变量
    printf("max=%d\n",max(a,b));
    return 0;
}

int max(int a,int b)    //a,b是函数形参
{
    int c;
    c=a>b?a:b;          //把a和b中的大者存放在c中
    return(c);
}
```



C:\WINDOWS\system32\cmd.exe

```
max=8
请按任意键继续. . .
```



程序第2行定义了全局变量a和b，并对其初始化。

第3行是main函数，在main函数中(第6行)定义了一个局部变量a。局部变量a的作用范围为第6~8行。在此范围内全局变量a被局部变量a屏蔽，相当于全局变量a在此范围内不存在(即它不起作用)，而全局变量b在此范围内有效。因此第6行中max(a,b)的实参a应是局部变量a，所以max(a,b)相当于max(8,5)。它的值为8。

第10行起定义max函数，形参a和b是局部变量。全局变量a和b在max函数范围内不起作用，所以函数max中的a和b不是全局变量a和b，而是形参a和b，它们的值是由实参传给形参的，即8和5。

变量的存储方式和生存期



动态存储方式与静态存储方式

从变量值存在的时间（即生存期）来观察，有的变量在程序运行的整个过程都是存在的，而有的变量则是在调用其所在的函数时才临时分配存储单元，而在函数调用结束后该存储单元就马上释放了，变量不存在了。

也就是说，变量的存储有两种不同的方式：**静态存储方式**和**动态存储方式**。

静态存储方式是指在程序运行期间由系统分配固定的存储空间的方式。

动态存储方式则是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。

动态存储方式与静态存储方式

数据分别存放在静态存储区和动态存储区中。全局变量全部存放在静态存储区中，在程序开始执行时给全局变量分配存储区，程序执行完毕就释放。在程序执行过程中它们占据固定的存储单元，而不是动态地进行分配和释放。

在动态存储区中存放以下数据:

- ① 函数形式参数。在调用函数时给形参分配存储空间。
- ② 函数中定义的没有用关键字static声明的变量，即自动变量。
- ③ 函数调用时的现场保护和返回地址等。

对以上这些数据，在函数调用开始时分配动态存储空间，函数结束时释放这些空间。在程序执行过程中，这种分配和释放是动态的，如果在一个程序中两次调用同一函数，而在此函数中定义了局部变量，在两次调用时分配给这些局部变量的存储空间的地址可能是不相同的。

如果一个程序中包含若干个函数，每个函数中的局部变量的生存期并不等于整个程序的执行周期，它只是程序执行周期的一部分。在程序执行过程中，先后调用各个函数，此时会动态地分配和释放存储空间。





存储类别

在C语言中，每一个变量和函数都有两个属性：**数据类型**和**数据的存储类别**。

存储类别指的是数据在内存中存储的方式(如静态存储和动态存储)。

在定义和声明变量和函数时，一般应同时指定其数据类型和存储类别，也可以采用默认方式指定（即如果用户不指定，系统会隐含地指定为某一种存储类别）。

C的存储类别包括4种：**自动的（auto）**、**静态的（static）**、**寄存器的（register）**、**外部的（extern）**。根据变量的存储类别，可以知道变量的作用域和生存期。



局部变量的存储类别

自动变量(auto变量)

函数中的局部变量，如果不专门声明为static（静态）存储类别，都是动态地分配存储空间的，数据存储在动态存储区中。函数中的形参和在函数中定义的局部变量（包括在复合语句中定义的局部变量），都属于此类。在调用该函数时，系统会给这些变量分配存储空间，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**。自动变量用关键字auto作存储类别的声明。

```
int f(int a)           //定义f函数, a为形参
{
    auto int b,c=3;    //定义b,c为自动变量
    :
}
```

实际上，关键字auto可以省略，**不写auto则隐含指定为“自动存储类别”**，它属于动态存储方式。

程序中大多数变量属于自动变量。 `auto int b,c=3; //等价于int b,c=3;`

有时希望函数中的局部变量的值在函数调用结束后不消失而继续保留原值，即其占用的存储单元不释放，在下一次再调用该函数时，该变量已有值（就是上一次函数调用结束时的值）。这时就应该指定该局部变量为“**静态局部变量**”，用关键字**static**进行声明。

局部变量的存储类别

静态局部变量(static局部变量)

【例7.16】考察静态局部变量的值。

```
#include <stdio.h>
int main()
{
    int f(int);           //函数声明
    int a=2,i;             //自动局部变量
    for(i=0;i<3;i++)
        printf("%d\n",f(a)); //输出f(a)的值
    return 0;
}

int f(int a)
{
    auto int b=0;          //自动局部变量
    static int c=3;        //静态局部变量
    b=b+1;
    c=c+1;
    return(a+b+c);
}
```



静态变量与自动变量的值的比较分析

第几次调用	调用时初值		调用结束时的值		
	b	c	b	c	a+b+c
第1次	0	3	1	4	7
第2次	0	4	1	5	8
第3次	0	5	1	6	9

```
C:\WINDOWS\system32\cmd.exe
7
8
9
请按任意键继续. . .
```

局部变量的存储类别

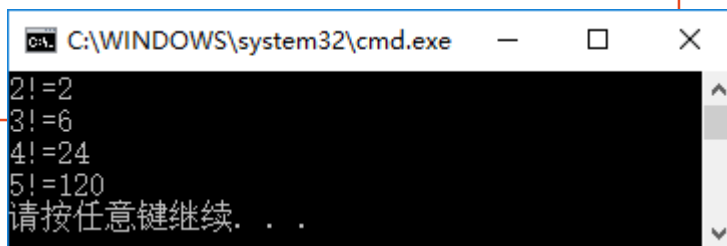
静态局部变量(static局部变量)

- (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（即动态局部变量）属于动态存储类别，分配在动态存储区空间而不在静态存储区空间，函数调用结束后即释放。
- (2) 对静态局部变量是在编译时赋初值的，即只赋初值一次，在程序运行时它已有初值。以后每次调用函数时不再重新赋初值而只是保留上次函数调用结束时的值。而对自动变量赋初值，不是在编译时进行的，而是在函数调用时进行的，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'\0'（对字符变量）。而对自动变量来说，它的值是一个不确定的值。这是由于每次函数调用结束后存储单元已释放，下次调用时又重新另分配存储单元，而所分配的单元中的内容是不可知的。
- (4) 虽然静态局部变量在函数调用结束后仍然存在，但其他函数是不能引用它的。因为它是局部变量，只能被本函数引用，而不能被其他函数引用。

局部变量的存储类别

【例7.17】输出1到5的阶乘值。

```
#include <stdio.h>
int main()
{
    int fac(int n);
    int i;
    for(i=1;i<=5;i++) //先后5次调用fac函数
        printf("%d!=%d\n",i,fac(i)); //每次计算并输出i!的值
    return 0;
}
int fac(int n)
{
    static int f=1; //f保留了上次调用结束时的值
    f=f*n; //在上次的f值的基础上再乘以n
    return(f); //返回值f是n!的值
}
```



```
C:\WINDOWS\system32\cmd.exe
2!=2
3!=6
4!=24
5!=120
请按任意键继续. . .
```



(1) 每次调用fac(i)，输出一个i!，同时保留这个i!的值以便下次再乘(i+1)。

(2) 如果函数中的变量只被引用而不改变值，则定义为静态局部变量(同时初始化)比较方便，以免每次调用时重新赋值。

注意

- 用静态存储要多占内存（长期占用不释放，而不能像动态存储那样一个存储单元可以先后为多个变量使用，节约内存），而且降低了程序的可读性，当调用次数多时往往弄不清静态局部变量的当前值是什么。因此，若非必要，不要多用静态局部变量。

局部变量的存储类别

寄存器变量(register变量)

一般情况下，变量（包括静态存储方式和动态存储方式）的值是存放在内存中的。当程序中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。经过运算器进行运算，如果需要存数，再从运算器将数据送到内存存放。

如果有一些变量使用频繁（例如，在一个函数中执行10 000次循环，每次循环中都要引用某局部变量），则为存取变量的值要花费不少时间。为提高执行效率，允许将局部变量的值放在CPU中的寄存器中，需要用时直接从寄存器取出参加运算，不必再到内存中去存取。由于对寄存器的存取速度远高于对内存的存取速度，因此这样做可以提高执行效率。这种变量叫做寄存器变量，用关键字register作声明。如

```
register int f;    //定义f为寄存器变量
```

由于现在的计算机的速度愈来愈快，性能愈来愈高，优化的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，现在实际上用register声明变量的必要性不大。

注意

- 3种局部变量的存储位置是不同的: 自动变量存储在动态存储区；静态局部变量存储在静态存储区；寄存器存储在CPU中的寄存器中。



全局变量的存储类别

全局变量都是存放在静态存储区中的。因此它们的生存期是固定的，存在于程序的整个运行过程。

一般来说，外部变量是在函数的外部定义的全局变量，它的作用域是从变量的定义处开始，到本程序文件的末尾。在此作用域内，全局变量可以为程序中各个函数所引用。但有时程序设计人员希望能扩展外部变量的作用域。



全局变量的存储类别

在一个文件内扩展外部变量的作用域

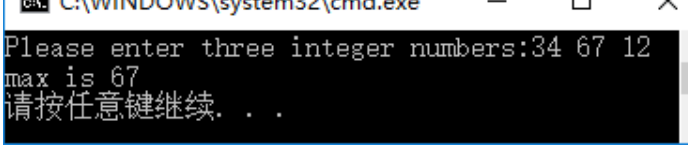
如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件结束。在定义点之前的函数不能引用该外部变量。如果由于某种考虑，在定义点之前的函数需要引用该外部变量，则应该在引用之前用关键字**extern**对该变量作“**外部变量声明**”，表示把该外部变量的作用域扩展到此位置。有了此声明，就可以从“声明”处起，合法地使用该外部变量。

【例7.18】调用函数，求3个整数中的大者。

注意

- 提倡将外部变量的定义放在引用它的所有函数之前，这样可以避免在函数中多加一个extern声明。
- 用extern声明外部变量时，类型名可以写也可以省写。例如，“extern int A,B,C;”也可以写成“extern A,B,C;”。因为它不是定义变量，可以不指定类型，只须写出外部变量名即可。

```
#include <stdio.h>
int main()
{
    int max();
    extern int A,B,C;    //把外部变量A,B,C的作用域扩展到从此处开始
    printf("Please enter three integer numbers:");
    scanf("%d %d %d",&A,&B,&C);    //输入3个整数给A,B,C
    printf("max is %d\n",max());
    return 0;
}
int A,B,C;                //定义外部变量A,B,C
int max()
{
    int m;
    m=A>B?A:B;            //把A和B中的大者放在m中
    if(C>m) m=C;          //将A,B,C三者中的大者放在m中
    return(m);            //返回m的值
}
```



全局变量的存储类别

将外部变量的作用域扩展到其他文件

如果一个程序包含两个文件，在两个文件中都要用到同一个外部变量Num，不能分别在两个文件中各自定义一个外部变量Num，否则在进行程序的连接时会出现“重复定义”的错误。正确的做法是：在任一个文件中定义外部变量Num，而在另一文件中用extern对Num作“外部变量声明”，即“extern Num;”。在编译和连接时，系统会由此知道Num有“外部链接”，可以从别处找到已定义的外部变量Num，并将在另一文件中定义的外部变量Num的作用域扩展到本文件，在本文件中可以合法地引用外部变量Num。

【例7.19】给定b的值，输入a和m，求a*b和a^m的值。

file1.c

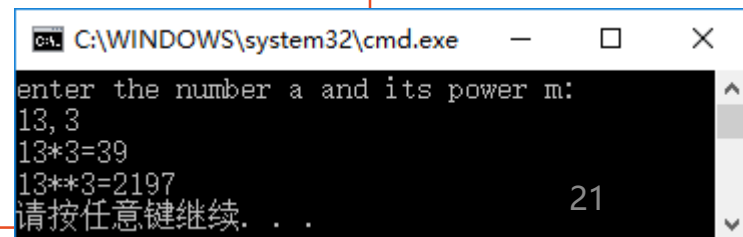
```
#include <stdio.h>
int A; //定义外部变量
int main()
{   int power(int); //函数声明
    int b=3,c,d,m;
    printf("enter the number a and its power m:\n");
    scanf("%d,%d",&A,&m);
    c=A*b;
    printf("%d*%d=%d\n",A,b,c);
    d=power(m);
    printf("%d**%d=%d\n",A,m,d);
    return 0;
}
```

注意

- 用这种方法扩展全局变量的作用域应十分慎重，因为在执行一个文件中的操作时，可能会改变该全局变量的值，会影响到另一文件中全局变量的值，从而影响该文件中函数的执行结果。

file2.c

```
extern A;
//把file1中定义的外部变量的作用域扩展到本文件
int power(int n)
{   int i,y=1;
    for(i=1;i<=n;i++)
        y*=A;
    return(y);
}
```



```
C:\WINDOWS\system32\cmd.exe
enter the number a and its power m:
13, 3
13*3=39
13**3=2197
请按任意键继续. . .
```

全局变量的存储类别

将外部变量的作用域扩展到其他文件

extern既可以用来扩展外部变量在本文件中的作用域，又可以使外部变量的作用域从一个文件扩展到程序中的其他文件，系统在编译过程中遇到extern时，

- 先在本文件中找外部变量的定义，如果找到，就本文件中扩展作用域；
- 如果找不到，就在连接时从其他文件中找外部变量的定义。如果从其他文件中找到了，就将作用域扩展到本文件；
- 如果再找不到，就按出错处理。

全局变量的存储类别

将外部变量的作用域限制在本文件中

有时在程序设计中希望某些外部变量只限于被本文件引用，而不能被其他文件引用。这时可以在定义外部变量时加一个static声明。

```
file1.c
static int A;
int main()
{
    :
}
```

```
file2.c
extern A;
void fun(int n)
{
    :
    A=A*n;    //出错
    :
}
```



这种加上static声明、只能用于本文件的外部变量称为**静态外部变量**。在程序设计中，常由若干人分别完成各个模块，各人可以独立地在其设计的文件中使用相同的外部变量名而互不相干。只须在每个文件中定义外部变量时加上static即可。这就为程序的模块化、通用性提供方便。如果已确认其他文件不需要引用本文件的外部变量，就可以对本文件中的外部变量都加上static，成为静态外部变量，以免被其他文件误用。至于在各文件中在函数内定义的局部变量，本来就不能被函数外引用，更不能被其他文件引用，因此是安全的。

全局变量的存储类别

将外部变量的作用域限制在本文件中

不要误认为对外部变量加static声明后才采取静态存储方式（存放在静态存储区中），而不加static的是采取动态存储（存放在动态存储区）。

声明局部变量的存储类型和声明全局变量的存储类型的含义是不同的。

对于局部变量来说，声明存储类型的作用是指定变量存储的区域(静态存储区或动态存储区)以及由此产生的生存期的问题，而对于全局变量来说，由于都是在编译时分配内存的，都存放在静态存储区，声明存储类型的作用是变量作用域的扩展问题。

全局变量的存储类别

将外部变量的作用域限制在本文件中

用**static**声明一个变量的作用是:

- (1) 对局部变量用static声明, 把它分配在静态存储区, 该变量在整个程序执行期间不释放, 其所分配的空间始终存在。
- (2) 对全局变量用static声明, 则该变量的作用域只限于本文件模块(即被声明的文件中)。

注意

- 用auto, register和static声明变量时, 是在定义变量的基础上加上这些关键字, 而不能单独使用。

```
int a;    //先定义整型变量a
static a; //企图再将变量a声明为静态变量
```



重新定义

»» 存储类别小结

对一个数据的定义，需要指定两种属性：**数据类型**和**存储类别**，分别使用两个关键字。

```
static int a;      //静态局部整型变量或静态外部整型变量  
auto char c;       //自动变量，在函数内定义  
register int d;     //寄存器变量，在函数内定义
```

此外，可以用extern声明已定义的外部变量。

```
extern b;          //将已定义的外部变量b的作用域扩展至此
```

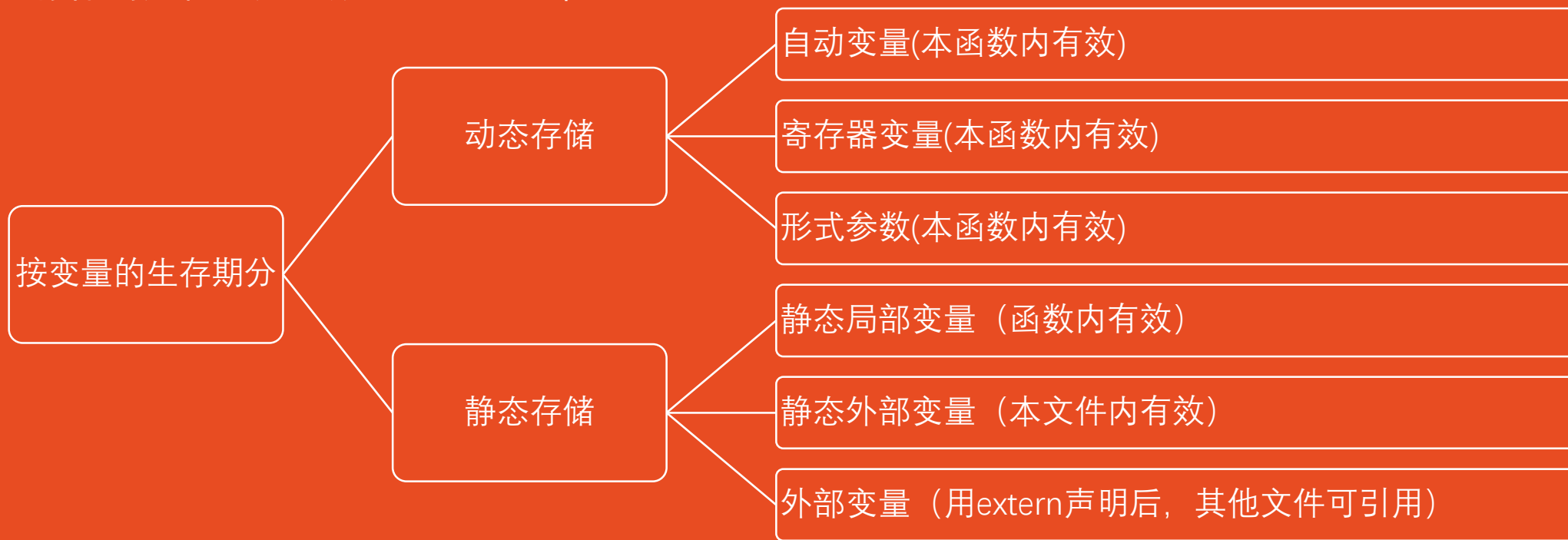
» 存储类别小结

(1) 从作用域角度分，有局部变量和全局变量。它们采用的存储类别如下：



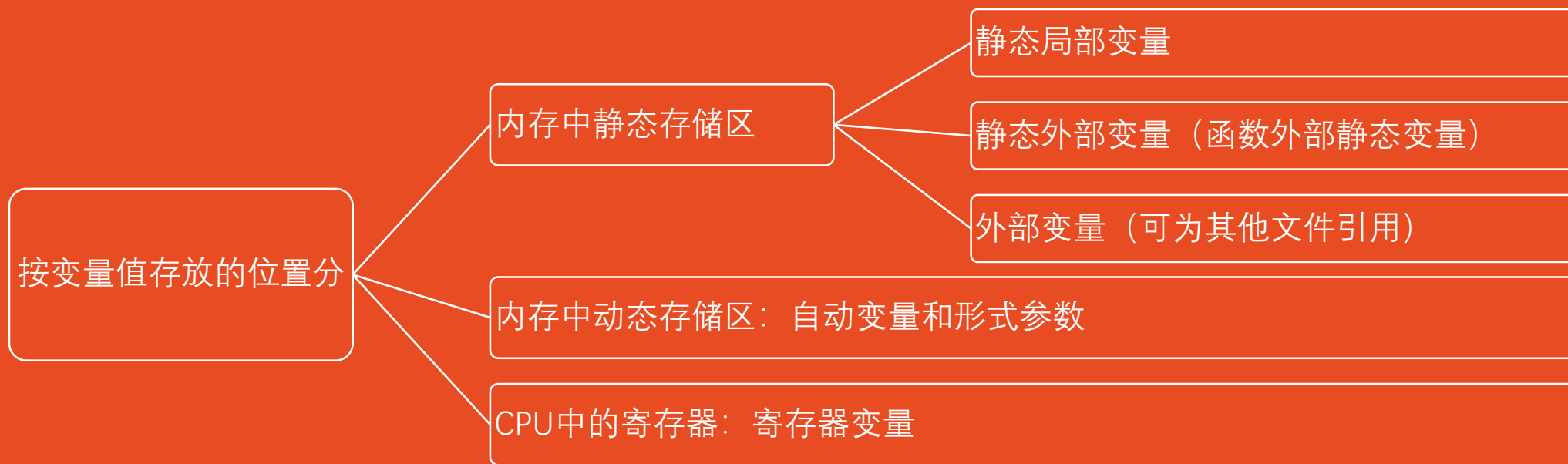
»» 存储类别小结

(2)从变量存在的时间(生存期)来区分,有动态存储和静态存储两种类型。静态存储是程序整个运行时间都存在,而动态存储则是在调用函数时临时分配单元。



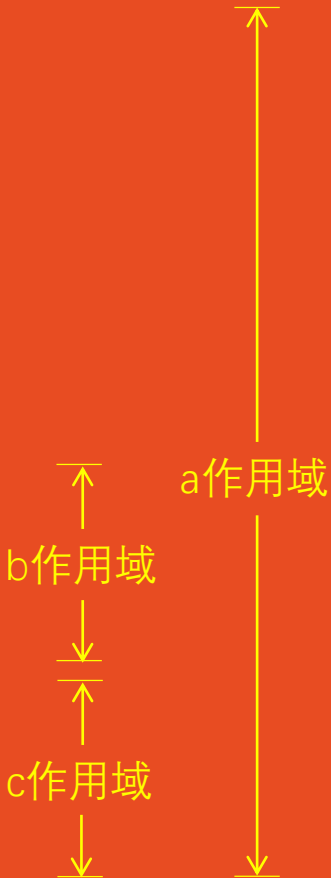
»» 存储类别小结

(3)从变量值存放的位置来区分,可分为:

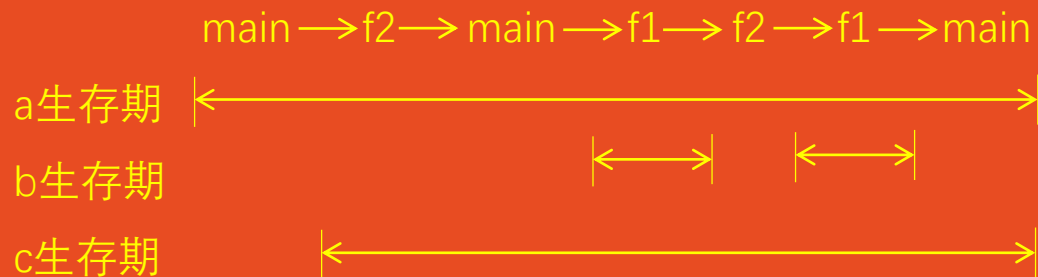


»» 存储类别小结

```
int a;  
int main()  
{  
    f2();  
    f1();  
}  
void f1()  
{  
    auto int b;  
    f2();  
}  
void f2()  
{  
    static int c;  
}
```



(4)关于作用域和生存期的概念。从前面叙述可以知道，对一个变量的属性可以从两个方面分析，一是变量的作用域，一是变量值存在时间的长短，即生存期。前者是从空间的角度，后者是从时间的角度。二者有联系但不是同一回事。



如果一个变量在某个文件或函数范围内是有效的，就称该范围为该变量的**作用域**，在此作用域内可以引用该变量，在专业书中称变量在此作用域内“**可见**”，这种性质称为变量的**可见性**。
如果一个变量值在某一时刻是存在的，则认为这一时刻属于该变量的**生存期**，或称该变量在此时刻“**存在**”。

»» 存储类别小结

各种类型变量的作用域和存在性的情况

变量存储类别	函数内		函数外	
	作用域	存在性	作用域	存在性
自动变量和寄存器变量	√	√	×	×
静态局部变量	√	√	×	√
静态外部变量	√	√	√（只限本文件）	√
外部变量	√	√	√	√

» 存储类别小结

(5)static对局部变量和全局变量的作用不同。

对局部变量来说，它使变量由动态存储方式改变为静态存储方式。

而对全局变量来说，它使变量局部化(局部于本文件)，但仍为静态存储方式。

从作用域角度看，凡有static声明的，其作用域都是局限的，或者局限于本函数内(静态局部变量)，或者局限于本文件内(静态外部变量)。

关于变量的声明和定义

在声明部分出现的变量有两种情况：一种是需要建立存储空间的(如“int a;”), 另一种是不需要建立存储空间的(如“extern a;”)。前者称为**定义性声明**(defining declaration), 或简称**定义**(definition)；后者称为**引用性声明**(referencing declaration)。一般把**建立存储空间的声明称定义**, 而把**不需要建立存储空间的声明称为声明**。

```
int main()
{
    extern A;           //是声明，不是定义。声明将已定义的外部变量A的作用域扩展到此
    :
    return 0;
}
int A;                 //是定义，定义A为整型外部变量
```

外部变量定义和外部变量声明的含义是不同的。外部变量的定义只能有一次，它的位置在所有函数之外。在同一文件中，可以有多次对同一外部变量的声明，它的位置可以在函数之内（哪个函数要用就在哪个函数中声明），也可以在函数之外。系统根据外部变量的定义（而不是根据外部变量的声明）分配存储单元。对外部变量的初始化只能在“定义”时进行，而不能在“声明”中进行。所谓“声明”，其作用是声明该变量是一个已在其他地方已定义的外部变量，仅仅是为了扩展该变量的作用范围而作的“声明”。

注意

- 有一个简单的结论，在函数中出现的对变量的声明(除了用extern声明的以外)都是定义。在函数中对其他函数的声明不是函数的定义。



内部函数和外部函数

函数本质上是全局的，因为定义一个函数的目的就是要被另外的函数调用。如果不加声明的话，一个文件中的函数既可以被本文件中其他函数调用，也可以被其他文件中的函数调用。但是,也可以指定某些函数不能被其他文件调用。根据函数能否被其他源文件调用，将函数区分为**内部函数**和**外部函数**。



内部函数

static 类型名 函数名(形参表);

```
static int fun(int a,int b)  
//表示fun是一个内部函数，不能被其他文件调用
```

内部函数又称静态函数，因为它用**static**声明的。使用内部函数,可以使函数的作用域只局限于所在文件。这样，在不同的文件中即使有同名的内部函数，也互不干扰，不必担心所用函数是否会与其他文件模块中的函数同名。

通常把只能由本文件使用的函数和外部变量放在文件的开头，前面都冠以static使之局部化，其他文件不能引用。这就提高了程序的可靠性。

外部函数

extern 类型名 函数名(形参表);

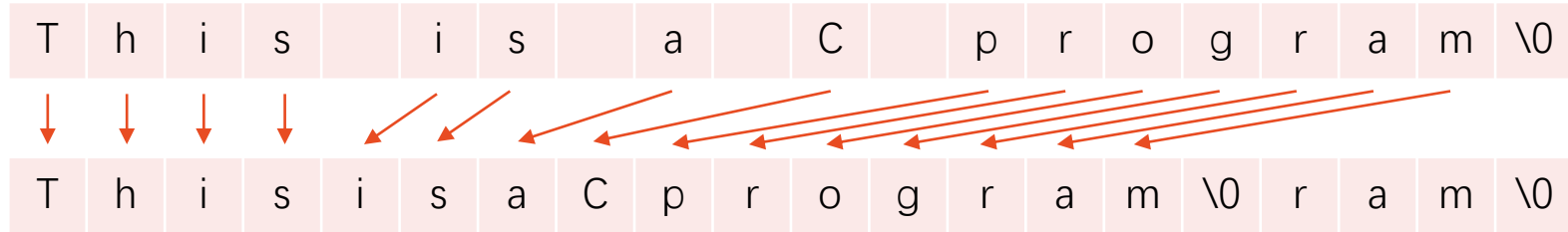
```
extern int fun(int a,int b)  
//表示fun可以被其他文件调用
```

在需要调用此函数的其他文件中，需要对此函数作声明(不要忘记，即使在本文件中调用一个函数，也要用函数原型进行声明)。在对此函数作声明时，要加关键字**extern**，表示该函数“是在其他文件中定义的外部函数”。

外部函数

【例7.20】有一个字符串,内有若干个字符,现输入一个字符,要求程序将字符串中该字符删去。
用外部函数实现。

解题思路: 设要删除空格



file1.c

```
#include <stdio.h>
int main()
{
    extern void enter_string(char str[]);           //对函数的声明
    extern void delete_string(char str[],char ch); //对函数的声明
    extern void print_string(char str[]);          //对函数的声明
    //以上3行声明了在本函数中将要调用的已在其他文件中定义的3个函数
    char c,str[80];
    enter_string(str);                             //调用在其他文件中定义的enter_string函数
    scanf("%c",&c);                                 //输入要求删去的字符
    delete_string(str,c);                          //调用在其他文件中定义的delete_string函数
    print_string(str);                             //调用在其他文件中定义的print_string函数
    return 0;
}
```

file2.c

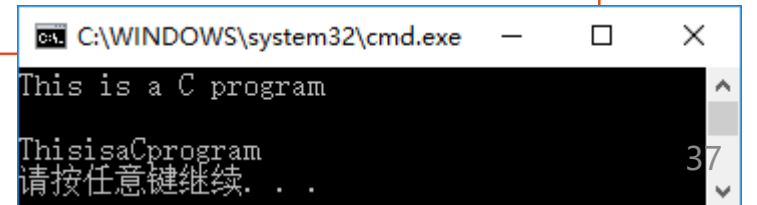
```
void enter_string(char str[80]) //定义外部函数enter_string
{
    gets(str);                  //向字符数组输入字符串
}
```

file3.c

```
void delete_string(char str[],char ch)
//定义外部函数delete_string
{
    int i,j;
    for(i=j=0;str[i]!='\0';i++)
        if(str[i]!=ch)
            str[j++]=str[i];
    str[j]='\0';
}
```

file4.c

```
void print_string(char str[])
//定义外部函数print_string
{
    printf("%s\n",str);
}
```



外部函数

使用extern声明就能够在本文件中调用在其他文件中定义的函数，或者说把该函数的作用域扩展到本文件。extern声明的形式就是在函数原型基础上加关键字extern。

由于函数在本质上是外部的，在程序中经常要调用其他文件中的外部函数，为方便编程，C语言允许在声明函数时省写extern。

用函数原型能够把函数的作用域扩展到定义该函数的文件之外（不必使用extern）。只要在使用该函数的每一个文件中包含该函数的函数原型即可。函数原型通知编译系统：该函数在本文件中稍后定义，或在另一文件中定义。

利用函数原型扩展函数作用域最常见的例子是#include指令的应用。在#include指令所指定的“头文件”中包含调用库函数时所需的信息。
