

C++程序设计

第二篇 面向过程编程

第6章 函数

第7章 作用域和存储类型

第8章 指针的高级用法

第9章 预处理命令

第6章 函数

6.1 函数定义

利用函数可以简化复杂程序的设计。

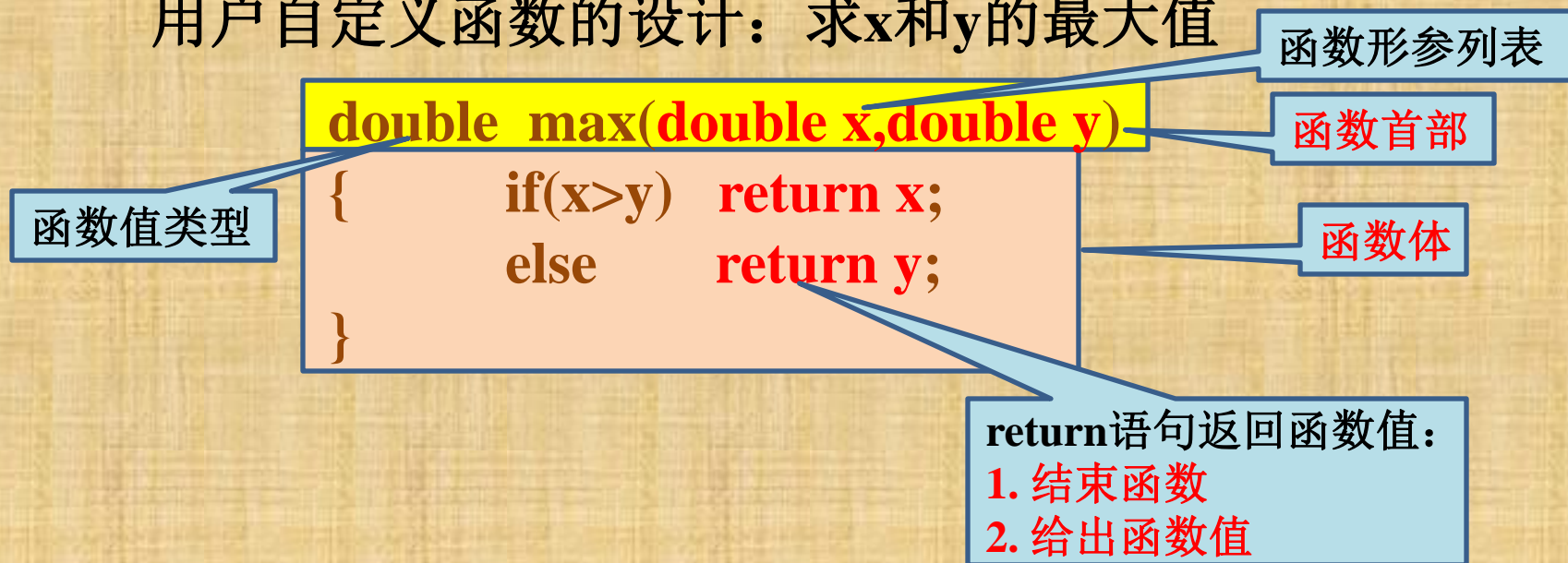
我们曾经使用过函数：

- 程序框架就是一个函数(主函数main)，其它函数均称为子函数
- 库函数，如`sqrt(2.0)`，`fabs(x*x-5*x+9)`，`strlen(s)`

除了上述函数，我们还可以设计自己的函数

- 用户自定义函数

用户自定义函数的设计：求x和y的最大值



用户自定义函数的使用：函数调用

```
double a=3 , b=4 , c ;
c=max(7,-8)+max(a,b);
c=max(a*a+7*a-8,b*b*b-3*b+2.768);
```

函数实参

例A6.1 函数用法1: `double max(double,double)`。

程序: MA6_1A.cpp 函数用法1

```
double max(double x,double y)           函数定义
{ if(x>y) return x; else return y; }
```

```
cout<<"最大值为:"<<max(a,b)<<endl;    函数调用
```

程序: MA6_1B.cpp 函数用法1 使用函数说明

```
double max(double x,double y);          函数说明
```

```
int main( )
```

```
{ .....
  cout<<"最大值为:"<<max(a,b)<<endl;  函数调用
  .....
}
```

函数说明可省略形参名, 如
`double max(double,double);`

```
double max(double x,double y)
```

```
{ if(x>y) return x; else return y; }
```


- 注1: 函数定义可以放在函数调用的前面, 此时不需要函数说明。函数定义通常放在函数调用的后面, 此时函数调用前一定要有函数说明。
- 注2: 函数说明中的形参列表可以没有形参名, 如 “`double max(double, double);`” 等价于 “`double max(double x, double y);`”。甚至函数定义中的形参列表也可以没有形参名, 在这种情况下, 函数体内无法使用相应的形参(具体例子见例B3.14)。
- 注3: 函数定义只能有一个而且必须有一个(库函数的定义已经编译成机器代码存放在库文件中), 但是函数说明可以有多个。
- 注4: 函数定义必须放在其它函数的外面, 函数说明通常也放在其它函数的外面, 但也可以放在调用它的函数(主调函数)函数体内。
- 注5: 函数定义中函数体内的`return`语句后面的表达式类型必须与函数值类型兼容, 即同类型或可以自动转换的类型, 通常保持同类型。

例A6.2 函数用法2: void showTime()。

include <ctime> 程序中用到时间的库函数要加上该行

void showTime(); 函数说明, 函数功能为输出时间

无函数值

int main()

{
.....

showTime(); 函数调用, 无函数值只进行输出

.....
}

void showTime()

函数定义

{

double t;

取时间的库函数: 运行的毫秒数

t=(double)clock()/CLOCKS_PER_SEC;

cout<<"当前时间为: "<<t<<"秒"<<endl;

// return ; 最后结束可加return;也可不加

}

return后不能跟函数值, 因这是void函数

注6：在函数内定义的变量是内部变量，其它函数不能使用。因此主函数不能使用子函数内定义的变量，同样子函数也不能使用主函数内定义的变量。另外，不同函数内可以定义同名的变量，它们是从属于所在函数的不同变量。

注7：除了专用于输入/输出的函数，一般不在子函数内输入/输出数据。通常是将数据传给子函数，用子函数处理数据，然后主函数进行输出或其它处理。

6.2 函数的参数传递

函数间的数据传递对于函数设计是很重要的。

以例A6.1 程序作为说明。函数间的数据传递分为：
参数传递和**函数值返回**。

主调函数：

```
int main( )  
{  
    .....  
    cout<<"最大值为:"<<max(a,b)<<endl;  
    .....  
}
```

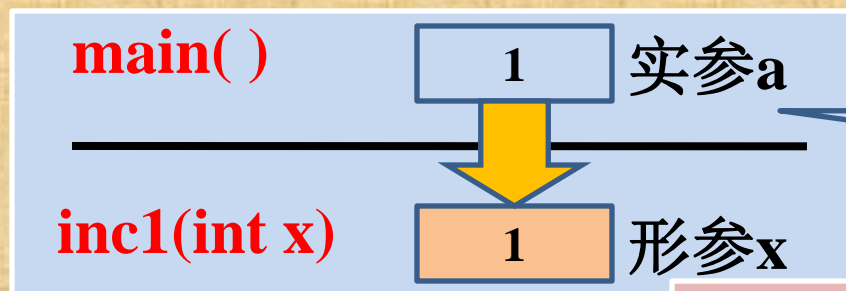
被调函数：

```
double max(double x,double y)  
{ if(x>y) return x; else return y; }
```

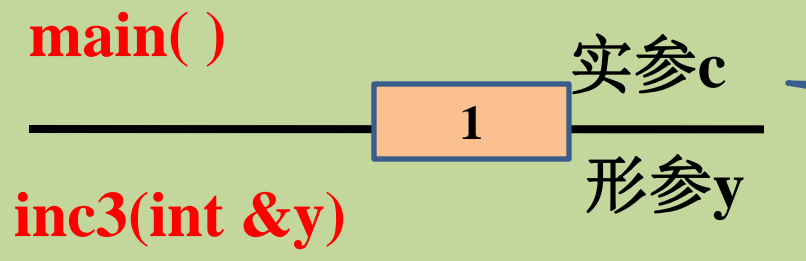
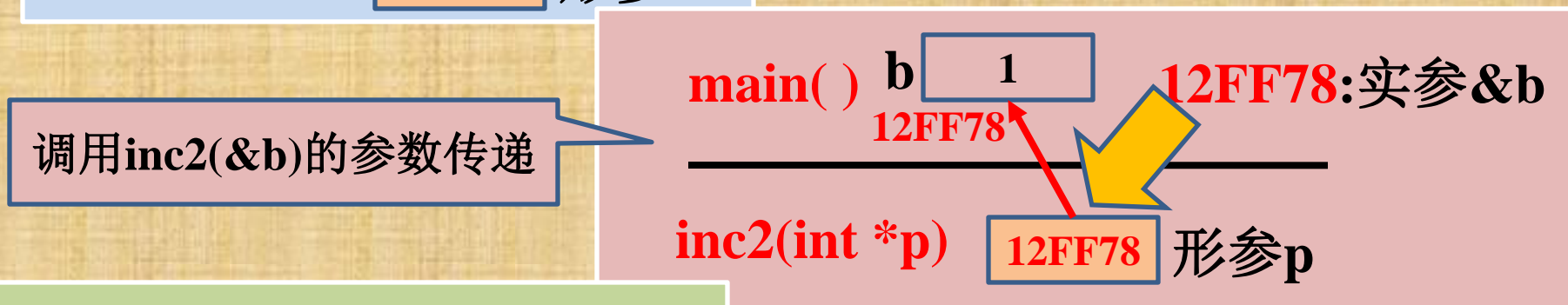


参数传递类型分3种:

- **值传递**: 形参为普通变量, 如 `void inc1(int x) { x++; }`
- **指针传递**: 形参为指针变量, 如 `void inc2(int *p){(*p)++;}`
- **引用传递**: 形参为引用, 如 `void inc3(int &y) { y++; }`



调用inc1(a)的参数传递



调用inc3(c)的参数传递

例B6.3 函数的值参数、指针参数和引用参数。

void inc1(int x),inc2(int *p),inc3(int &y); 3种类型参数

int main()

{ int a=1,b=1,c=1;

.....

inc1(a); 传递值：变量a的值1

inc2(&b); 传递指针：变量b的地址

inc3(c); 传递引用：变量c取别名

..... }

void inc1(int x) { x++; } 值参数函数

void inc2(int *p) { (*p)++; } 指针参数函数

void inc3(int &y) { y++; } 引用参数函数

注1：函数参数若是普通类型，参数传递是值传递，传递是单向的，函数无法改变实参的值。若是引用参数，参数传递是双向的，函数可以改变实参的值。若是指针参数，函数可以改变实参指向的值。

注2：在函数调用中，普通类型参数的实参可以是同类型(可以是兼容类型)的表达式，但是引用类型参数的实参必须是同类型的变量或数组元素，即必须是左值。

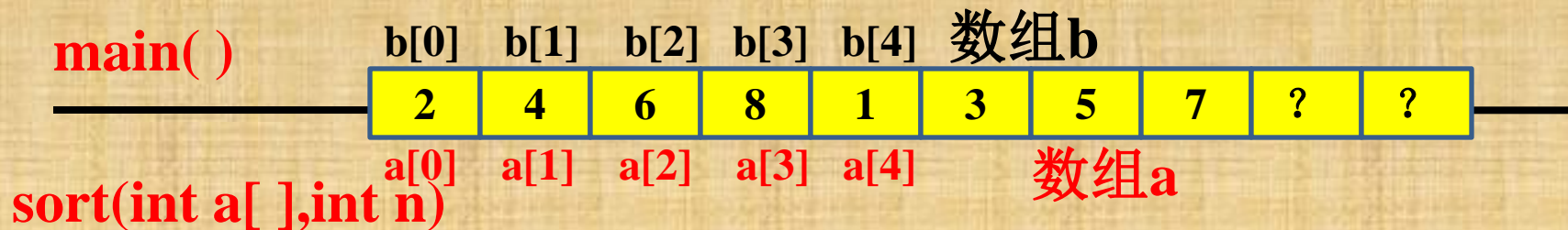
函数经常用于处理数组，实参数组和形参数组的使用如下：

`void sort(int a[],int n);` 形参中`int a[]`为形参数组
若`int b[100];`可用`sort(b,n);` 实参中`b`为实参数组(用数组名)

例C6.4 数组排序子函数。程序MC6_4.cpp

```
void sort(int a[ ],int n);
int main( )
{
    int b[100],n,i; .....
    sort(b,n); 实参数组b，传递给形参数组a
                ..... 于是a与b表示同一个数组
}
void sort(int a[ ],int n) 形参数组a
{
    .....
    if(k!=i) { t=a[k]; a[k]=a[i]; a[i]=t; }
}
```


函数数组参数传递图示：sort(b,n)的参数传递情况



注3：形参数组也可以有数组长度，如函数首部为“void sort(int a[100],int n)”，但通常不写数组长度。

注4：形参数组的本质是形参指针变量，“void sort(int a[],int n)”等价于“void sort(int *a,int n)”。故子函数内若用sizeof(a)，实际上是形参指针变量a所占的字节数，即4个字节，不是实参数组所占的字节数。

函数数组参数传递的本质：

- **sort(b,n) 等价于 sort(&b[0],n) b传递的是指针**
- **void sort(int a[],int n) 等价于 void sort(int *a,int n)**
参数传递后，指针a指向数组b，于是可将数组b看成数组a，
此时数组元素 a[i]就是*(a+i)

子函数中可以利用指针来处理数组，以提高代码执行效率。
程序例子MC6_4.cpp的sort函数可改写如下：

```
void sort(int *p,int n)
{
    int *pend,*pmin,*pj; int t;
    pend=p+n;
    for( ; p<pend ; p++ ) 从p到pend-1找最小放入p处
    {
        for(pmin=p,pj=p+1;pj<pend;pj++) pj查找最小
        if(*pj<*pmin) pmin=pj;
        if(pmin!=p) { t=*pmin; *pmin=*p; *p=t; } 放最小
    }
}
```

注5：当函数参数是数组时，实参应为数组名或某个地址(表示该地址开始的数组)，例如上述程序中使用sort(b,n)或sort(&b[0],n)甚至sort(&b[3],n-3)都是合法的函数调用，即数组参数对应的实参必须是指针。若使用sort(b[],n)、sort(b[100],n)或sort(b[n],n)进行调用都是错误的，因为b[]不表示什么，而b[100]、b[n]仅仅表示某个数组元素。

函数参数类型也可以是结构类型和枚举类型。

例A6.5 结构变量和枚举值做函数参数。

```
struct intPair { int x,y; }; 结构类型定义
enum mode { Vector, Fraction }; 枚举类型定义
intPair add(intPair a,intPair b,mode m); 参数为变量是值传递
int main( )
{   intPair a={3,4}, b={5,6},c;
    c=add(a,b,Vector); 函数调用,实参为结构变量和枚举值
    .....
}
intPair add(intPair a, intPair b, mode m)
{   intPair c; int i;
    .....
    return c; 结构变量的值作为函数值
}
```

注6: 结构类型和枚举类型值做函数参数是值传递。

注7: 在C++中函数的参数传递次序是反向进行的, 从最后一个参数到第一个参数依次传递。

6.3 递归函数、重载函数和默认参数函数

6.3.1 递归函数

C++函数可以自己调用自己，这就是**递归函数**。

求n!的递推公式：
$$\begin{cases} 0!=1!=1 \\ n!=(n-1)! \times n \quad n=2,3,4, \dots \end{cases}$$

可定义求n!的递归函数如下：

```
double Factorial(int n)
{
    if(n<=1) return 1;
    else return Factorial(n-1)*n;
}
```

n!数值很大,使用double型表示

递归终止条件

递归调用

例A6.7 递归函数的使用。

```
int main( )
{
    .....
    cout<<n<<"!="<<Factorial(n)<<endl;
    .....
}

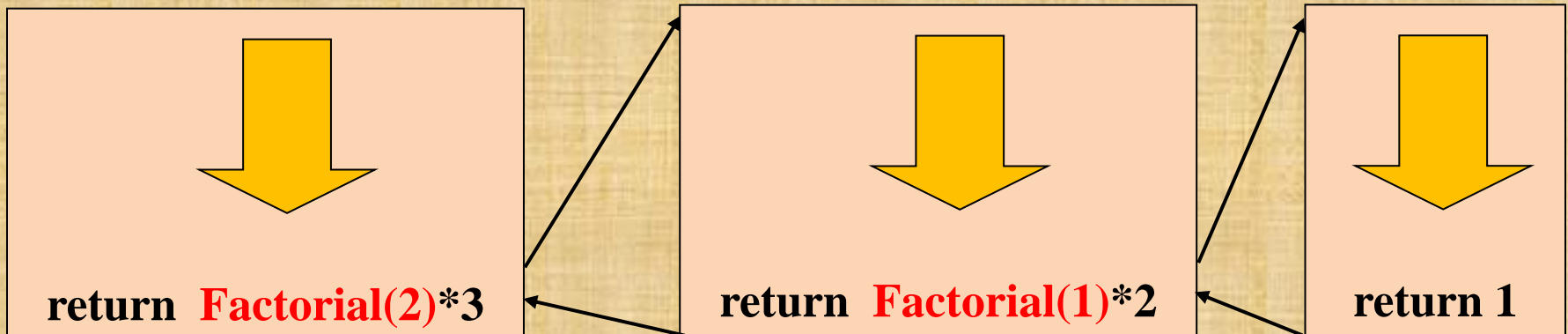
double Factorial(int n)
{
    if(n<=1) return 1;
    else     return Factorial(n-1)*n;
}
```

递归函数 **Factorial(3)** 的执行过程:

Factorial(3)

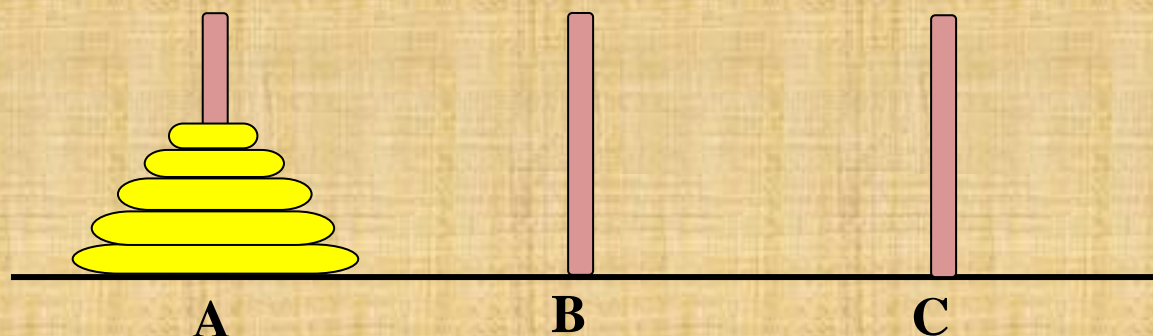
Factorial(2)

Factorial(1)



求解河内塔问题:

所谓河内塔问题,就是将一个柱子上的 n 个大小不一的中空的盘子,全部移动到另一个柱子上。要求每次只能移动一个盘子,而且要保证任何一个柱子上都是下面的盘子比上面的盘子大。



河内塔问题求解:

$\text{Hanoi}(n, x, y, z)$

- 若 $n=0$, 则什么也不做

- 若 $n \geq 1$, 分3步: 1. A上面 $n-1$ 个盘子移到B柱

$\text{Hanoi}(n-1, x, z, y)$

- 2. A中大盘子移到C柱

$\text{move}(x, z)$

- 3. B柱 $n-1$ 个盘子移到C柱

例C6.8 求解河内塔问题。

```
void Hanoi(int n,char x,char y,char z) 递归求解河内塔
{
    if(n>=1)
    {
        Hanoi(n-1,x,z,y);    递归调用解n-1层
        cout<<x<<"---"<<z<<endl;
        Hanoi(n-1,y,x,z);    递归调用解n-1层
    }
}
```

○ 上述是直接递归，还有间接递归

```
int f1(int x)
{
    .....
    y=f2(x+2,x-2);
    .....
}

int f2(int x1,int y1)
{
    .....
    z=f1(x1)+f1(y1);
    .....
}
```

注1：不管是直接递归还是间接递归，都要避免无限递归的情况出现，即都要趋于递归终止点。

6.3.2 重载函数

不同的函数使用同一个函数名，就是**函数重载**。

同名函数就是**重载函数**。

以求绝对值为例：

absolute(x)

一个功能一个名：求绝对值

处理int, int absolute(int x)

处理double, double absolute(double x)

处理complex, double absolute(complex x)

例C6.9 重载函数的使用。

```
struct complex { double x,y; };  
int absolute(int x);  
double absolute(double x);  
double absolute(complex c); } 重载函数说明  
  
int a= -10; double b= -3.14; complex c={3,-4};  
cout<<absolute(a)<<absolute(b)<<absolute(c)<<endl;  
  
int absolute(int x) { return x>=0?x:-x; } 函数定义  
double absolute(double x) { return x>=0?x:-x; }  
double absolute(complex c) { return sqrt(c.x*c.x+c.y*c.y);} }
```

注2：重载函数的参数列表必须不同，否则会出错。

6.3.3 默认参数函数

C++函数可以提供**默认参数**，当实参个数不足时用来**补充缺少的实参**。

注3：函数给出的默认参数应该是靠后面的参数，多个默认参数应该是靠右连续的。

注4：默认参数通常在函数说明中给出。

例C6.10 默认参数的使用。

```
void showNumber(unsigned int n,int idx=10);  
showNumber(a,2); showNumber(a,16); showNumber(a);  
void showNumber(unsigned int n,int idx)  
{ char digit[ ]="0123456789ABCDEF";  
  if(n>=idx) showNumber(n/idx,idx); 递归输出idx进制数  
  cout<<digit[n%idx]; 输出idx进制数的个位数字  
}
```

默认参数

使用默认参数

6.4 内联函数

使用**内联函数**实现判断闰年的功能：

```
inline bool isLeapYear(int year)    内联函数
{ return (year%4==0&&year%100!=0||year%400==0); }
int main()
{ ..... if(isLeapYear(yy)) cout<<"闰年"; ..... }
```

实际实现相当于作了如下替换：

```
int main()
{...if(yy%4==0&&yy%100!=0||yy%400==0) cout<<"闰年"; }
```

内联函数使用函数的格式，实际是插入程序片段

例A6.11 内联函数的使用。

```
inline bool isLeapYear(int year)    内联函数
{ return (year%4==0&&year%100!=0||year%400==0); }
if(isLeapYear(yy)) cout<<yy<<"是闰年"<<endl;
else                cout<<yy<<"不是闰年"<<endl;
```

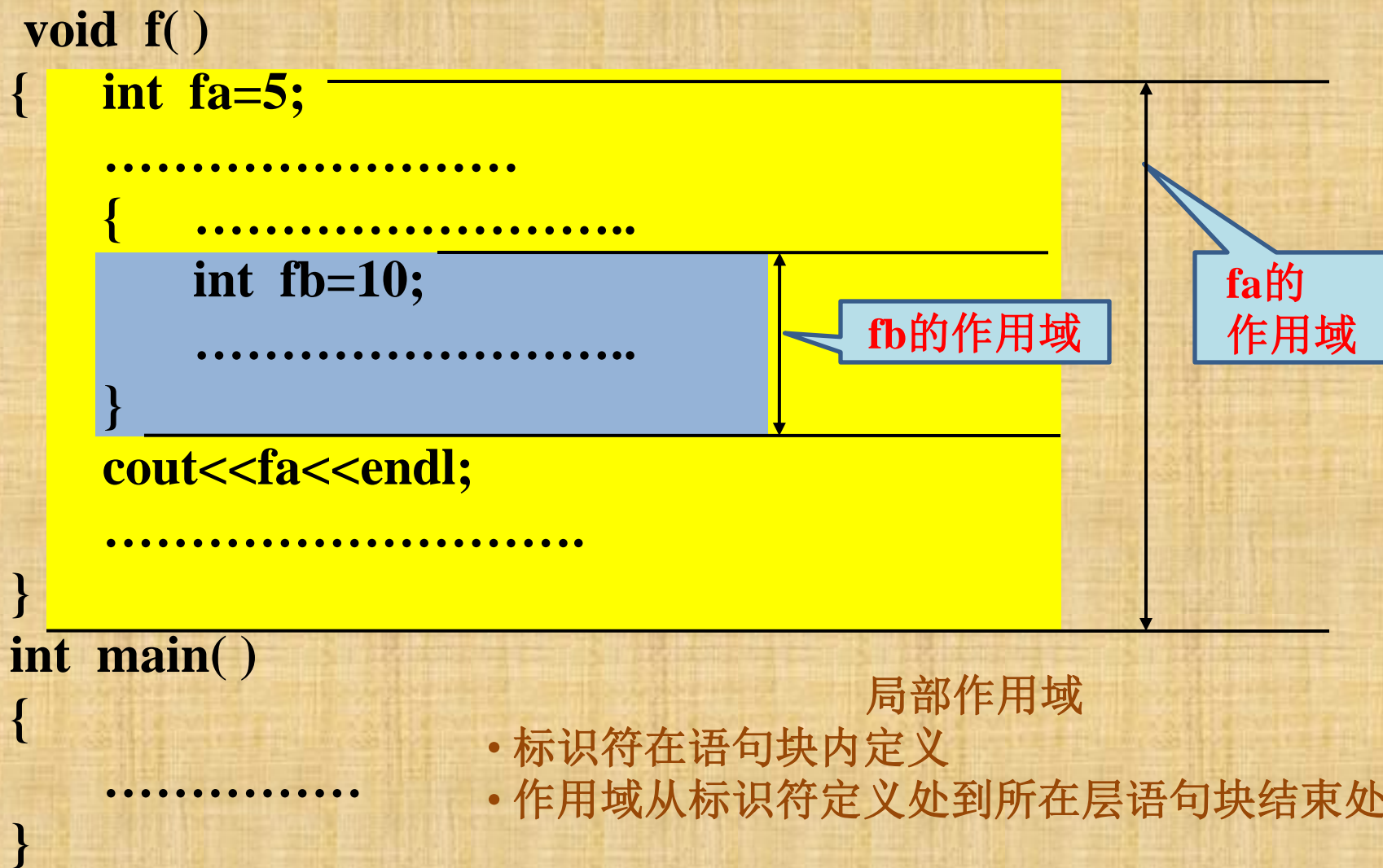
第7章 作用域和存储类型

7.1 标识符的作用域

标识符的作用域——标识符在程序中的有效范围

标识符的 作用域	局部作用域	复合语句(语句块)内有效
	全局作用域	源程序文件内有效
	语句作用域	结构性语句内有效
	名空间作用域	名空间内有效
	类作用域	类内有效

7.1.1 局部作用域和全局作用域




```
# include <iostream>
```

```
.....
```

```
int a=100;
```

```
void f( )
```

```
{
```

```
.....
```

```
}
```

```
int b=200;
```

```
int main( )
```

```
{
```

```
.....
```

```
}
```

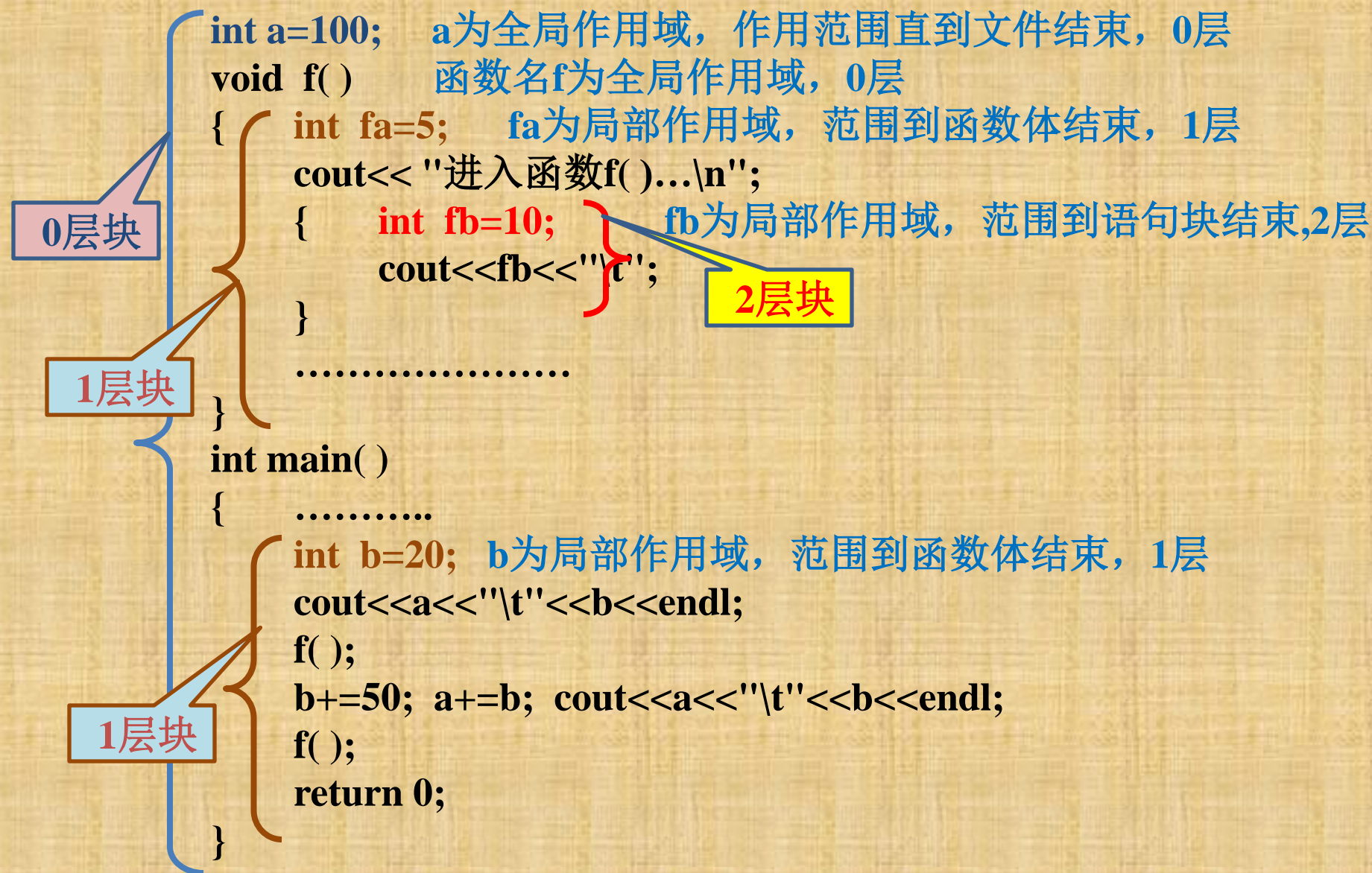
a的作用域

b的作用域

全局作用域

- 标识符在语句块的外面定义
- 作用域从标识符定义处到所在源程序文件结束处

例B7.1 演示局部作用域和全局作用域。



注1：作用域是指标识符的作用域，是名字的作用域。有时说变量的作用域是指变量名的作用域。

注2：变量名已经超出作用域范围，但是变量可能仍然存在，只是不能使用该变量名了。

例B7.2 演示函数形参的作用域。

形参作用域在1层块

```
.....  
void f( int a , double b )  
{  
    cout<< "f(a,b) { a="<<a<<"\tb="<<b<<"}\n";  
    int fa=5;  
    cout<< "f(a,b) { fa="<<fa<<"}\n";  
}
```

形参a、b
的作用域

- 函数形参是局部作用域
- 作用域从形参说明到函数体结束处

7.1.2 ○标识符的定义与说明以及混合作用域

函数说明和变量说明可以扩展函数名和变量名的作用域

例B7.3 演示变量说明及混合作用域。



7.1.3 语句作用域

if、switch、while、for首部的表达式表内定义的标识符的作用域就是**语句作用域**，范围为从定义处到该语句结束。

例**B7.4** ○if、switch、while条件中定义的标识符的作用域。

```
int main( )
{
    .....
    if( int k=3 )
        cout<<k<<endl;
    .....
    switch( int s=n+1 )
    {
        .....
    }
    while( int t=7 )
    {
        t+=5; cout<<t<<endl; if(--n<0) break; }
}
```

The diagram illustrates the scope of variables defined in the code blocks:

- k的作用域** (Scope of k): Indicated by a yellow box around the `if(int k=3)` block and its body.
- s的作用域** (Scope of s): Indicated by a blue box around the `switch(int s=n+1)` block and its body.
- t的作用域** (Scope of t): Indicated by a green box around the `while(int t=7)` block and its body.

例B7.5 演示for中定义的标识符的作用域。

```
int main( )
{
    .....
    for( int i=0; i<3 ; i++ )
    {
        for( int j=0 ; j<3 ; j++ )
        {
            cout<<a[i][j]<< "\\t"; }
            cout<<endl; // cout<<j<<endl; 错误
        }
        // cout<<i<<endl; 错误
        .....
    }
}
```

i的作用域

j的作用域

注4: 由于for语句表达式1定义的标识符在C++标准和旧版本C++中作用域是不同的, 为了避免因此引起的程序移植问题, 尽量不要在表达式1中定义变量, 除非不会引起问题。

7.1.4 ○ 名空间及名空间作用域

将程序中的各种标识符分组,每个组有组名,这些组称为**名空间**。

名空间定义如下:

namespace group

```
{    double area=100;
      void sort(int x[ ],int n)
      {    int i,j,k,t;
          for(i=0;i<n;i++)
          {    for(k=i,j=i+1;j<n;j++) if(x[j]<x[k]) k=j;
              if(k!=i) { t=x[k]; x[k]=x[i]; x[i]=t; }
          }
      }
}
```

area的作用域

sort的作用域

名空间外使用**area**和**sort**:

```
cout<<group::area<<endl; group::sort(a,8);
```

例A7.6 名空间的使用及名空间作用域的范围。

```
namespace group1
```

```
{  
    const float pi=3.14;  
    void sort(int x[ ],int n);  
}
```

pi 的作用域

```
namespace group2
```

```
{  
    .....  
}
```

sort 扩展的作用域

```
int main( )
```

```
{  
    ..... std::cin>>r; area1=group1::pi*r*r; area2=group2::pi*r*r;  
    group1::sort(a,8); .....  
}
```

```
namespace group1
```

```
{  
    void sort(int x[ ],int n)  
    {  
        .....  
    }  
}
```

pi的作用域

sort 的作用域

```
.....
```

程序中的`cin`和`cout`也是名空间中的标识符，名空间为`std`。所以使用时要写成`std::cin`和`std::cout`。

使用`using namespace std;`可以在程序中省略`std::`这样的前缀。

7.1.5 标识符的重名

一般来讲，除了重载函数使用同样的函数名外，在程序中应尽可能避免名称相同，即所谓的重名，因为重名容易造成冲突，产生二义性。

当相同名称的标识符的作用域有重叠时，C++规定，标识符的作用域必须在不同层的块上(包括全局作用域，即0层的块)，否则将产生二义性。

注7：在内层块中定义或说明的标识符将隐藏外层的同名标识符，故重名时标识符默认为内层的标识符。

例B7.8 标识符重名问题。

全局变量**a**的作用域：0层

```
int a=100;
```

```
void f(int a)
```

```
{ int s= -123;
```

```
{ char a= 'A';
```

```
cout<<s<<"\t"<<a<endl;
```

```
}
```

```
cout<<s<<"\t"<a<<"\t"<<::a<<endl;
```

```
}
```

```
int main( )
```

```
{ int i=3,j=4,s;
```

```
s=i*j; .....
```

```
{ double i=7.9;
```

```
s=i*j; .....
```

```
}
```

```
.....
```

```
}
```

形参**a**和局部变量**s**
的作用域：1层

局部变量**a**的作用域：2层

::a指全局变量**a**,
即0层的变量**a**

局部变量 **i,j,s**
的作用域：1层

局部变量**i**的作用域：2层

7.1.6 ○函数说明的隐藏规则

函数说明可以放在块内，并且可以在不同层的块内出现。当内层块中的函数说明与块外函数说明同名时，将隐藏块外的**所有同名函数**的说明。

例**B7.9** 函数说明的隐藏特性。

```
void f( ),f(int x), f(double x);
```

```
int main( )
```

```
{    void f( ); void f(int x);  
    f( ); f(3); f(3.1); //实际调用f(int x)  
    {    void f(double x);  
        f(3.1); // f( );错误：函数未说明  
    }  
    return 0;  
}
```

函数f()和f(int x) 的作用域
隐藏了f(double x)

函数f(double x) 的作用域
隐藏了f()和f(int x)

```
void f( ) { ... }
```

```
void f(int x) { ... }
```

```
void f(double x) { ... }
```

注8：函数说明是函数集的说明，内层说明的函数集将隐藏外层同名的函数集。

7.2 程序的内存映像

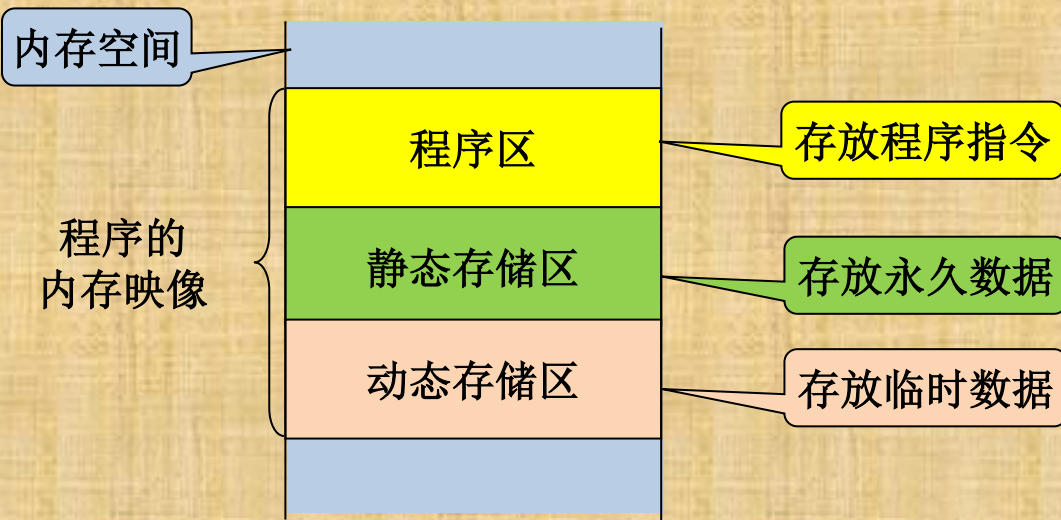
计算机运行程序过程：

1. 将可执行文件代码从外存储器装载到内存
2. 运行内存中的代码

程序的内存映像

程序的内存映像分为：

- 程序区
- 静态存储区
- 动态存储区

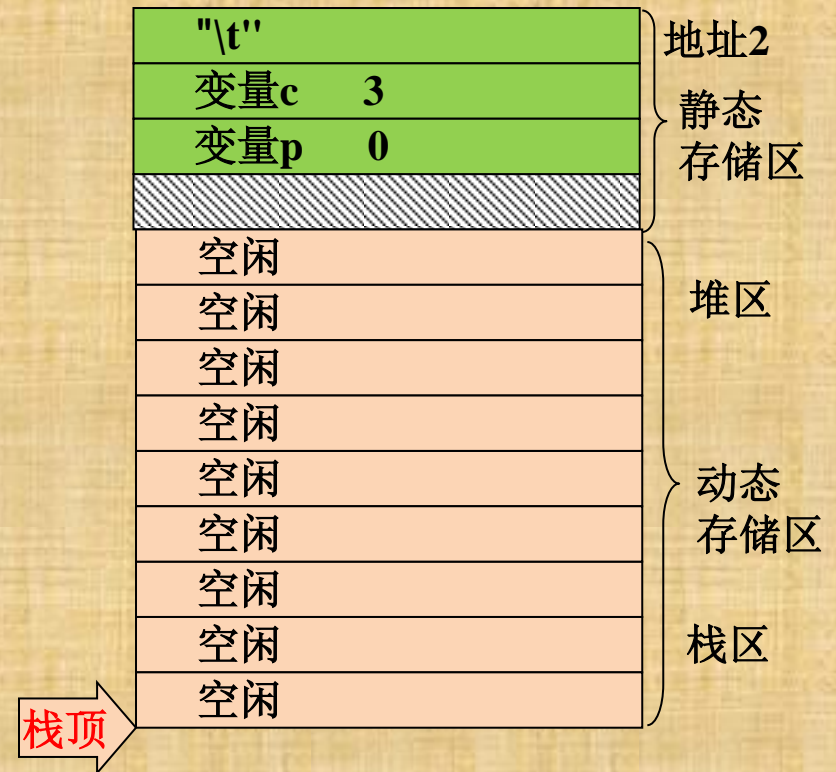


例B7.10 演示程序的内部运行情况。

```
# include <iostream>
using namespace std;
int max(int x,int y);
int main( )
{   static int  c=3,*p;
    int  a=1518,b=97;
    c=max(a,b);
    p=new int;
    *p=100;
    cout<<c<<"\t"<<*p<<endl;
    delete p;
    return 0;
}
int max( int x , int y )
{   int t;
    if(x>y) t=x;
    else    t=y;
    return t;
}
```

例B7.10 的图示

main	1	int a=1518	地址1
	2	int b=97	
	3	push b	
	4	push a	
	5	push 地址1	
	6	call max 跳转到16步	
	7	c←EAX	
	8	p←new分配的堆空间地址	
	9	*p←100	
	10	cout<< c	
	11	cout<<地址2中字符串	
	12	cout<< *p	
	13	cout<< endl	
	14	delete p释放p指向的堆空间	
	15	return 0	
max	16	int t	程序区
	17	if(x>y) t ← x	
	18	else t ← y	
	19	return t t值传给EAX	



7.3 变量的存储类型

7.3.1 auto类型和static类型

- **auto类型**：局部变量的一种类型，在动态存储区，定义时生成，出变量名作用域时撤销。
- **static类型**：可以为局部变量，也可以是全局变量，在静态存储区，程序运行前生成，程序结束后撤销。

static类型变量在变量名出作用域后依然存在，可以永久保存数据。

另外静态存储区中的变量**若没有赋初值，则初值自动设为0**，这一点与动态存储区中变量未赋初值的情况不同。

例B7.11 演示auto和static 类型的差异。

```
void f1()
```

每次生成新变量t，赋初值100

```
{ auto int t=100; t++; cout<<t<<endl; }
```

```
void f2()
```

程序运行前就存在，初值为100

```
{ static int t=100; t++; cout<<t<<endl; }
```

```
int main()
```

```
{ auto int x;
```

auto变量未赋初值，值不确定，
但VC为0xcccccccc

```
static int y;
```

static变量未赋初值，值默认0

```
int *p=&x;
```

```
cout<<*p<<"\t"<<y<<endl; VS2008要间接输出没赋值变量
```

```
f1(); f2(); 此处f2()中的变量存在，但没有变量名
```

```
f1(); f2();
```

```
return 0;
```

```
}
```

注1：函数参数属于自动数据。

例C7.12 统计函数调用次数。 static变量可做内部计数器

int Fibonacci(int n); **n>0**计算, **n<=0**统计

```
int main( )
```

```
{ int n=10;
```

```
  cout<<Fibonacci(n)<<endl;
```

```
  cout<<Fibonacci(0)<<endl;
```

```
  .....
```

```
}
```

```
int Fibonacci(int n)
```

```
{ static int count=0;
```

static变量count用于记录调用函数次数

```
  if(n<=0) return count;
```

```
  count++;
```

统计调用次数

```
  .....
```

```
}
```

7.3.2 ○register类型

register类型用于**申请使用CPU中的寄存器作为变量**，这样使用变量，可以大大提高存储速度。

例**C7.14** register存储类型变量的使用。

```
int main( )
{  int n;  register int F0,F1,Fn,i;
    F0=F1=Fn=1;
    cin>>n;
    for(i=3;i<=n;i++)
    {  Fn=F0+F1;
        F0=F1; F1=Fn;
    }
    cout<<Fn<<endl;
    return 0;
}
```

频繁使用的变量
用register类型

变量i,F0,F1,Fn在此处
反复大量使用

7.3.3 无名变量——函数值

函数值是一个**无名变量**：

- 函数值是**基本类型**时，函数值是CPU中的某个**寄存器**
- 函数值是**复合类型**时，函数值是内存中的**某个单元**

例**B7.15** ○演示函数值是一个无名变量。

```
struct date { int year,month,day; date() {} 定义结构类型date
date(const date&) { cout<< "无名变量地址: "<<this<<endl; } };
date f(int yy,int mm,int dd);
int main( )
{   date d1;  d1=f(2016,10,1);
    cout<<"&d1="<<&d1<<endl;
    return 0;
}
date f(int yy,int mm,int dd)
{   date d;
    d.year=yy; d.month=mm; d.day=dd;
    cout<<"&d="<<&d<<endl;
    return d;
}
```

运行结果：

&d=0024FA70

无名变量地址：0024FAA8

&d1=0024FB7C

函数值的地址

返回d的值：d传值给函数值

7.3.4 ○多文件程序中的变量和函数

C++程序可以由多个源文件组成。各个源文件的函数和全局变量可以相互使用，也可以限制在源文件内使用。多文件组成程序的规则为：

- 所有的源程序文件中**只能且必须有一个main函数**，程序总是从main函数开始执行。
- 如果函数或全局变量定义前有**static说明**，则该函数或全局变量只能在所在源文件中使用，**其它文件禁止使用**。
- 如果函数或全局变量定义前没有static限制，则其它源文件中可以使用该函数或变量，**使用前要有函数说明或变量说明**。
- **变量说明要用extern说明**，函数也可前置extern，但通常省略。

第8章 指针的高级用法

8.1 指针与函数

8.1.1 指针作参数

通过指针参数，函数可以改变主调函数中变量的值，也可以将多个值传给主调函数。

例A8.1 用指针传出多个数据。

```
int main( )
{ double x,y; int k; .....
  k=integer(x,&y);   integer传出两个值，函数值和y
  .....
}

int integer(double x,double *pfraction)
{ int k=x; *pfraction=x-k;
  if(*pfraction<0) { (*pfraction)++; k--; } return k;
}
```

指针参数还常常用于交换主调函数中的两个变量的值。

例C8.2 用指针交换两个变量的值。

```
int main( )
{  double d[200]; int i,j,n; .....
    for(i=0,j=n-1;i<j;i++,j--) swap(&d[i],&d[j]);
    .....
}

void swap(double *x,double *y)  交换 *x和*y
{  double t=*x; *x=*y; *y=t; }
```

d[i],d[j]指针传给形参x,y, 于是*x即d[i],*y即d[j],交换*x和*y即交换d[i]和d[j]

swap的3种错误写法:

```
void swap(double x,double y)
{ double t=x; x=y; y=t; }
```

只交换形参
不交换实参

```
void swap(double *x,double *y)
{ double *t=x; x=y; y=t; }
```

只交换指针
不交换实参

```
void swap(double *x,double *y)
{ double *t; *t=*x; *x=*y; *y=*t; }
```

交换实参
用了野指针

程序中避免使用野指针、0指针指向的变量。

8.1.2 指针传递数组和返回指针

函数经常用于处理数组，而**数组参数本质上就是指针参数**。

函数有时也会**返回一个指针**作为函数值。

例C8.3 编写函数实现字符串求长度、比较、复制及合并操作(返回指针)。

```
int main( )
{  char ss1[80],ss2[80]; .....
   strCopy(ss1, "第二个字符串为:");
   cout<< strConcate(ss1,ss2)<<endl;    .....
}

char *strCopy( char *s1,char *s2)
{  char *ret=s1; while(*s1=*s2) { s1++;s2++; } return ret;  }

char *strConcate( char *s1,char *s2 )
{  char *ret=s1; ..... return ret;  }
```


再看一个函数返回字符指针的例子，其实就是返回字符串。

例C8.4 将一个int型数转换成二进制字符串。

```
int main( )
{ int n; cin>>n;
  cout<<n<< "="<<intToBinaryString(n)<<endl;
  .....
}

char *intToBinaryString( int n )
{ static char binary[33]; unsigned int k; k=(unsigned int)n;
  binary[32]= '\0';
  for(int i=31;i>=0;i--)
  {   binary[i]=k%2+'0'; k/=2; }
  return binary;  实质上是返回一个字符串
}
```

binary数组必须是在静态存储区

注1：局部数组binary在子函数结束后还要使用，必须定义为静态数组。

8.3 各类指针

8.3.1 字符指针

在C++中字符串用字符串的首地址表示，即用字符指针表示从指针指向的位置直到串结束标志'\0'的这一串字符。

具体表示方式上，字符串可以有多种方式。

- 字符串常量——表示常量首地址所指字符串
- 字符数组名——表示数组所保存的字符串
- 字符指针变量——表示指针值所指的地址开始的字符串

注1：字符串用字符指针表示，表示从指向位置开始到遇到的第一个'\0'结束的串。

注2：在C++中可以输入/输出字符指针，实际上是输入/输出字符串。但其它类型的指针不能输入，只能输出，此时输出的是一个8位的十六进制的指针值。如果要输出字符指针的指针值，可以显式转换成其它指针类型(见5.2节注9)。

```
char s[ ]="Apple";  字符数组s赋初值 "Apple"  
char ss[3][80]={"Apple", "Pear", "Banana"};  
                二维字符数组ss赋初值"Apple"、"Pear"、"Banana"  
char *p1="Pear";  字符指针变量赋初值 "Pear"  
char *p2,*p3=s;   字符指针变量赋初值s  
p2="Banana";      字符指针变量赋值 "Banana"
```

上述定义和赋值得到了字符串：

s "Apple"

ss[0] "Apple"

ss[1] "Pear"

ss[2] "Banana"

p1 "Pear"

p2 "Banana"

p3 "Apple"

字符串的各种用法：

```
cout<<&s[2]<< "\\t"<<ss[2]<< "\\t"<<p1<< "\\t"<<p2<<endl;
```

```
cout<<"Hello\0World"+6<<endl;
```

输出结果：

```
ple Banana Pear Banana  
World
```


8.3.2 ○行指针

二维数组可以看成由行构成的数组，而以行为单位的地址就是**行指针**。

LINE类型：行类型

`int a[3][4];`  `typedef int LINE[4]; LINE a[3];`

数组 a $\left\{ \begin{array}{l} \mathbf{a[0]} \quad \{ \mathbf{a[0][0]}, \mathbf{a[0][1]}, \mathbf{a[0][2]}, \mathbf{a[0][3]} \} \\ \mathbf{a[1]} \quad \{ \mathbf{a[1][0]}, \mathbf{a[1][1]}, \mathbf{a[1][2]}, \mathbf{a[1][3]} \} \\ \mathbf{a[2]} \quad \{ \mathbf{a[2][0]}, \mathbf{a[2][1]}, \mathbf{a[2][2]}, \mathbf{a[2][3]} \} \end{array} \right.$

`a[0]`表示一行，`LINE`类型(即`int [4]`类型)，相当于一维数组名，通常作为一维数组的首地址，即`a[0] ≡ &a[0][0]`，为`int*`类型。

`a`表示`&a[0]`，`LINE*`类型(即`int(*)[4]`类型)，为行数组的首地址。

`int(*p)[4];`定义行指针变量`p`。

注3: `int(*)[4]`是行指针类型，`int*[4]`是指针数组类型。`int (*p)[4];`定义了一个行指针变量`p`，`int*pt[4];`定义了一个指针数组`pt`，元素类型是`int*`。

注4: 二维形参数组就是形参行指针变量。

例A8.6 用行指针使用二维数组。

```
void magicSquare(int (*x)[11],int n);  
void print(int (*x)[11],int n); 等价于 void print(int x[ ][11],int n);  
int main( )  
{ int a[11][11]; magicSquare(a,5); print(a,5); return 0; }  
void magicSquare(int (*x)[11],int n)  
{ ..... for(i=0;i<n;i++) for(j=0;j<n;j++) x[i][j]=0;  
  for(i=0,j=n/2,s=1; s<=n*n;s++)  
  {   x[i][j]=s; i=(i-1+n)%n; j=(j+1)%n;  
      if(x[i][j]) { i=(i+2)%n; j=(j-1+n)%n; }   }  
}  
void print(int (*x)[11], int n)  
{ for(int i=0;i<n;i++)  
  { for(int j=0;j<n;j++) cout<<x[i][j]<<"\t"; cout<<endl; }  
}
```

行指针变量 **p** 指向某个二维数组后，可以看成该二维数组就是 **p** 数组。

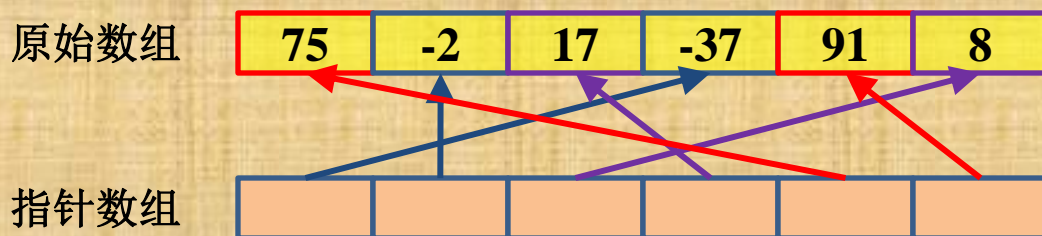
8.3.3 指针数组与指向指针的指针

数组可以是基本类型、结构类型，也可以是指针类型。

当用指针处理的是指针类型的数组时，该指针就是指向指针的指针。

指针数组可以用于进行索引排序，也可以用于处理成批的字符串。

索引排序图示：



例C8.7 数组的索引排序。

```
int main( )
{  double d[100],*idx[100];    .....
    for(i=0;i<n;i++)  cin>>d[i];
    for(i=0;i<n;i++)  idx[i]=&d[i];
    sort(idx,n);
    for(i=0;i<n;i++)  cout<<d[i]<<"\t";  cout<<endl;
    for(i=0;i<n;i++)  cout<<*idx[i]<<"\t";  cout<<endl;
}

void sort(double *p[ ],int n)
{  double *t;  int i,j,k;
    for(i=0;i<n;i++)
    {  for(k=i,j=i+1;j<n;j++) if(*p[j]<*p[k]) k=j;
        if(k!=i) { t=p[k]; p[k]=p[i]; p[i]=t; }
    }
}
```

指针数组赋值

输出原始数组

按索引输出数组(有序)

实际数据比较

交换指针

使得p[i]指向后面的最小值

一个字符指针数组可以表示和处理一批字符串。实际的字符则保存在其它的内存空间中，如常量区。

例A8.8 字符指针数组赋初值与指向指针的指针。

```
int main( )
```

```
{ char *str[ ]={ "Red", "Orange", "Yellow", "Green", 0};
```

```
  for(int i=0;str[i]!=0;i++) cout<<str[i]<<endl;
```

```
  for(char **p=str; *p!=0; p++) cout<<*p<<endl;
```

```
  return 0;
```

```
}
```

字符串常量实际表示
该常量的首地址

0指针作为结束标志

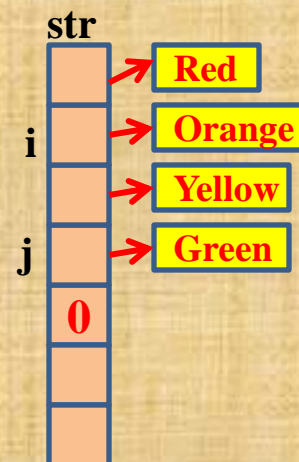
指针p处理数组str，为指向
str[i]的指针，char**类型

*p为p指向的元素
str[i]，表示字符串

字符指针数组处理成批字符串很方便。

例C8.9 用字符指针数组处理字符串：输入、排序和输出。

```
int main( )
{  char *str[100]; input(str); output(str); sort(str); .....  }
void input(char *s[ ])  相当于 void input(char **s)
{  char line[80]; int i=0;
  while(cin.getline(line,80))
  {s[i]=new char[strlen(line)+1];strcpy(s[i],line);i++;}
  s[i]=0;
}
void sort(char *s[ ])
{  char *t; int i,j,k
  for( int i=0 ;s[i] ; i++)
  {  for(k=i,j=i+1;s[j];j++) if(strcmp(s[j],s[k])<0) k=j;
    if(k!=i) {  t=s[k]; s[k]=s[i]; s[i]=t;  }
  }
}
.....
```



s[i],s[j],s[k]均表示字符串

例C8.9程序改成指针的等价程序

```
int main( )
{   char *str[100]; input(str); output(str); sort(str); ..... }

void input(char **s)
{   char line[80];
    while(cin.getline(line,80))
    { *s=new char[strlen(line)+1];strcpy(*s,line);s++;}
    *s=0;
}

void sort(char **s)
{   char **q,**min,*t;
    for( ;*s ; s++)
    {   for(min=s,q=s+1;*q;q++) if(strcmp(*q,*min)<0) min=q;
        if(min!=s) { t=*min; *min=*s; *s=t; }
    }
}
.....
```

8.3.4 函数指针

指针可以用来间接使用函数。而指向函数代码的指针就是**函数指针**。

函数指针使用如下：

若有函数

```
int max(int x,int y)
{
    .....
}
```

max为该类型的指针常量

则该函数的类型为：**int (int,int)**

指向该类型函数的指针类型为：**int (*)(int,int)**

定义该类型的变量 **pf** 的形式为：**int (*pf)(int,int)**

若有定义 `int (*pf)(int,int)=max;` **pf指向函数max**
则可如下使用函数max: `pf(a,b)` 由于与C语言兼容, 也可用 `(*pf)(a,b)`

注7: 不要混淆函数指针变量的定义 `int (*pf)(int,int)` 与返回指针值的函数说明 `int *f(int,int)`。

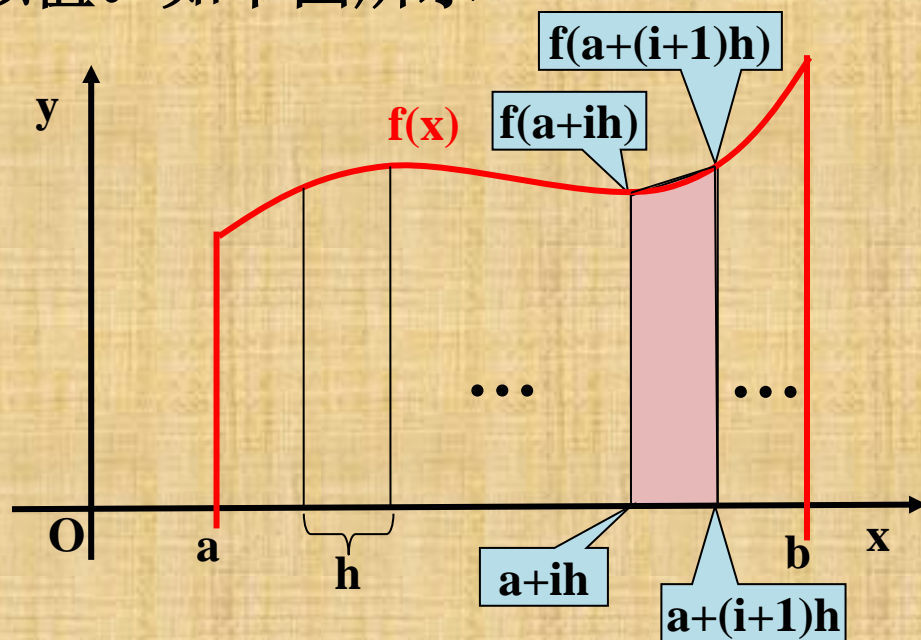
例A8.11 函数指针的使用。

```
int max(int x,int y),min(int x,int y);
int main( )
{   int a=3,b= -4;  int (*pf)(int,int);
    cout<<max(a,b)<<"\t"<<min(a,b)<<endl;
    pf=max; cout<<pf(a,b)<<endl;    pf(a,b)=max(a,b)
    pf=min; cout<<pf(a,b)<<endl    pf(a,b)=min(a,b)
}
```

注8: 类似于指针指向数组, 当函数指针变量pf指向某个函数时, 我们也称该函数是pf函数。

函数指针可以做函数参数用以编写通用的函数，如编写通用的积分函数。

积分的近似计算是计算函数在积分区间中的曲边梯形的面积近似值。如下图所示



积分的近似计算公式为：

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} \frac{1}{2} (f(a+ih) + f(a+(i+1)h))h = \left[\frac{f(a) + f(b)}{2} + \sum_{i=1}^{n-1} f(a+ih) \right] h$$

例C8.12 编写通用积分函数。

```
double f1(double x), f2(double x);
double integral(double (*f)(double), double a, double b);
int main( )
{ cout<<"integral(f1)="<<integral(f1,0,5)<<endl;
  .....
  cout<<"integral(sin)="<<integral(sin,0,3.1415927)<<endl;
  .....
}
.....
double integral(double (*f)(double), double a, double b)
{ double s,h; int i,n=20;
  h=(b-a)/n; s=(f(a)+f(b))/2; f(a),f(b)为函数指针调用函数
  for(i=1;i<n;i++) s+=f(a+i*h);
  return s*h;
}
```

8.3.6 const与指针

在C++中**const**用以说明常量，我们可以将const理解为**冻结**了所说明的量，使之不能改变数值。

const使用规则：

- **const**向左冻结变量，即 **Type const**
- **Type const** 类型习惯上写成**const Type**类型，强调常量
- **const** 冻结的变量即为常量，必须有初始值

常量例子：

```
const int MaxInt=0x7FFFFFFF;
```

```
const double pi=3.141592653589793;
```

```
const int a[4]={1,2,3,4};
```

```
const char s[ ]= "Hello";
```

指针中的常量:

```
int a=3,b=4;
```

```
const int *p2=&a; //等价于 int const * p2=&a;
```

```
int * const p3=&a;
```

```
const int * const p4=&a; //等价于 int const *const p4=&a;
```

注11: 若定义“int a=3; const int *p2=&a;”, 则对于同一个变量, 当它以a的身份使用时就是变量, 以*p2的身份使用时就是常量, 因为经过p2时受到了不能改动的约束。

例B8.14 ○演示指针常量与指向常量的指针。

```
int main( )
```

```
{  int a=3,b=4;  .....
```

```
    const int *p2=&a;  *p2为常量,但p2为变量,  *p2=3; 错误
    p2=&b;
```

```
int *const p3=&a;  p3为常量,但*p3为变量,  p3=&b; 错误
    *p3=5;
```

```
const int *const p4=&a;  p4和*p4均为常量
                        *p4=10; 和 p4=&b; 都错误
```

```
cout<<*p4<<endl;  a=10;  *p4是常量, 但a不是常量
cout<<*p4<<endl;
return 0;
```

```
}
```

可以使用const来**确保**函数参数在函数内不被改动。

例**C8.15** const在函数中的保护作用。

```
int main()  
{ double d[200]; input(d,5); print(d,5); cout<< max(d,5)<<endl; ... }  
void input(double *x,int n)  
{ for(double *p=x;p<x+n;p++) cin>>*p; }  
void print(const double *x,int n) 函数内禁止改动x指向的数组  
{ for(const double *p=x;p<x+n;) cout<<*p++<<"\t" ; cout<<endl; }  
double max(const double *x,int n) 函数内禁止改动x指向的数组  
{ double mx=*x;  
  for(const double *p=x;p<x+n;p++) { if(*p>mx) mx=*p; } return mx;  
}
```

注12：冻结后的数据无法再解冻，即指向变量的指针值可以赋给指向常量的指针变量，但反过来不可以，以保证常量不会被非法改动。若定义“int a=3,*p1; const int *p2=&a;”则赋值“p1=&a; p2=p1;”是可以的，但是赋值“p1=p2;”不可以。另外要注意，若还有定义“int *const p3=&a;”则赋值“p1=p3;”是可以的，因为这是将int*类型的常量p3赋值给int*类型的变量。

注13：字符串常量(如 "Hello")是指保存在常量区的字符串的首地址，为char*类型的指针，而非const char*类型的指针。但实际上该字符串是常量，不能改动。

8.3.7 容易混淆的指针使用

1、野指针与0指针

double *pd;

for(int i=0;i<3;i++) cin>>pd[i]; pd是野指针, 使用pd指向的数组非法

int a[3]={1,2,3},b=4,*c[3]={&a[0],&b,0},*d;

***c[1]=*c[2];** c[2]是0指针, 使用c[2]指向的数据*c[2]非法

***d=*c[1];** d是野指针, 使用d指向的变量非法

2、字符串的使用

char s1[]="Apple", s2[80]="Banana";char *p1= "Orange",*p2=s2;

使用正确:

cout<<s2; strcpy(s2,s1); cout<<p2; cin>>p2; p2= "Apple";

if(strcmp(s1, "Apple")==0) cout<< "OK";

使用有隐患:

cin>>s1;

输入的字符串长度必须小于5

if(s1==s2) cout<< "OK";

实际比较的是字符串的首地址的大小

使用错误:

cin>>p1;

p1指向字符串常量, 输入字符串到常量中非法

s1=s2;

数组不能直接赋值, 因为两边都是首地址

s2="Pear";

数组不能直接赋值

strcpy(s1,s2);

s1数组长度是6, 而保存字符串s2的长度为7, 超界

3、行指针与指针数组的关系

```
int a[3][4], (*p)[4]=a, *b[3]={&a[0][0],a[1],a[2]}, **c=b;
```

下列使用正确:

```
p=a;
```

```
c=b;
```

```
b[0]=a[0];
```

```
c[0]=p[0];
```

下列使用错误:

```
a=p;
```

数组不能直接赋值

```
b=p;
```

数组不能直接赋值

```
p=b;
```

指针类型不同, p为int(*)[4]类型, b为int**类型

```
c=a;
```

指针类型不同, c为int**类型, a为int(*)[4]类型

```
a[0]=b[0];
```

a[0]表示数组, 数组不能直接赋值

```
p[0]=c[0];
```

p[0]表示数组, 数组不能直接赋值

各种量所占字节数

```
sizeof(a): 48
```

```
sizeof(p): 4
```

```
sizeof(b): 12
```

```
sizeof(c): 4
```

```
sizeof(a[0]): 16
```

```
sizeof(p[0]): 16
```

```
sizeof(b[0]): 4
```

```
sizeof(c[0]): 4
```

```
sizeof(a[0][0]): 4
```

```
sizeof(p[0][0]): 4
```

```
sizeof(b[0][0]): 4
```

```
sizeof(c[0][0]):4
```


4、const指针

```
int a[4]={1,2,3,4};  
int *p0=&a[3];  
const int *p1=&a[0];  
int * const p2=a+1;  
const int * const p3=&a[2];
```

使用正确:

```
*p0=5; *p2=5;
```

p0=p2; p2为int*类型常数，可以赋值给int*类型变量p0

p1=p0; p0为int*类型，p1为const int*类型，可以通过赋值冻结数据

const int *pp=p3; pp为const int*类型，p3为同类型，可做初值

使用错误:

p0=p1; p1是const int*类型，p0是int*类型，赋值将解冻，非法

int *pp2=p1; p1是const int*类型，pp2是int*类型，赋值将解冻，非法

*p1=5; *p1是常量，不能赋值

p2=p0; p2是常量

p2=p1; p2是常量

p3=&a[2]; p3是常量

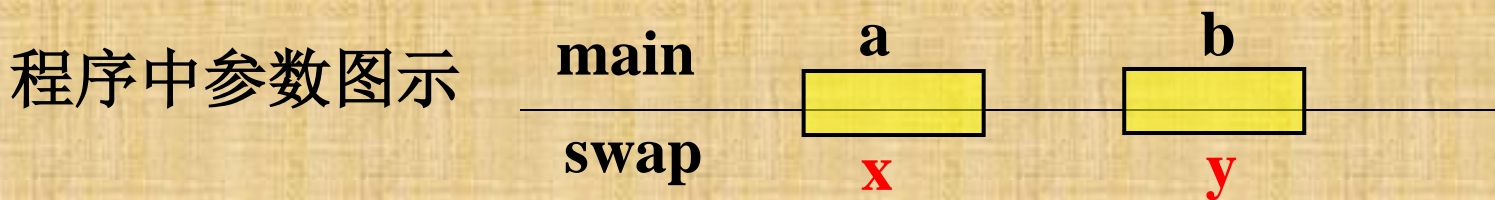
8.4 跨函数引用

8.4.1 引用型参数

函数通过引用参数可以直接使用主调函数中的变量，具有指针参数类似的功能，但比指针参数更加直接。

例**B8.16** 用引用交换两个变量的值。

```
int main( )  
{  double a=0.123,b= -35.7;  
    swap(a,b);  
    cout<< "a="<<a<< "\tb="<<b<<endl; return 0;  
}  
void swap(double &x,double &y)    x,y是实参的别名  
{  double t=x; x=y; y=t; }
```



8.4.2 返回引用

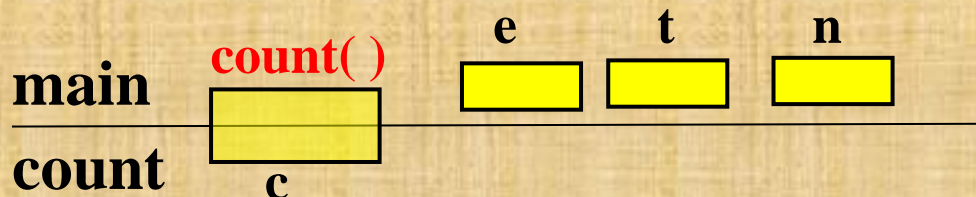
函数值也可以是引用，表示return后面变量的别名。

例B8.17 演示返回引用的特殊作用。

```
int main( )
{
    .....
    for(e=1,t=1,n=1;t>=1e-3; ) { ..... count(); }
    cout<< "e="<<e<< "\\tcount="<<count( )-1<<endl;
    count( )=0;    统计值清0
    for(e=1,t=1,n=1;t>=1e-7; ) { ..... count(); }
    cout<< "e="<<e<< "\\tcount="<<count( )-1<<endl;
    return 0;
}

int &count( ) { static int c=0; c++; return c; }
```

程序中函数值图示



注1: 函数参数为引用, 则实参必须为同类型的变量或数组元素(即左值)。函数值为引用, 则return后面的表达式必须为同类型的变量或数组元素。

注2: 若函数参数为const引用, 则实参可以为左值, 也可为右值, 特别是常量。函数值为const引用, return后面的表达式也可作为右值。如

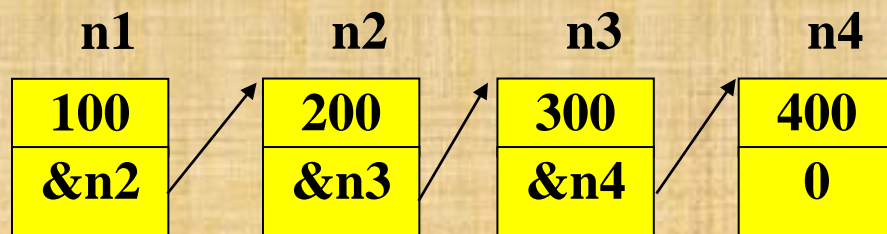
```
const int &f(const int &x,const int &y) { return x+y; }
```

则在定义 “int a=3,b;” 下可使用调用 “b=f(a,9);”。

8.5 链表和二叉树

8.5.1 单链表

用指针将一批数据串成一串，每一个数据都有一个指向下一个数据的指针，这就构成了单链表，如图。



单链表中每个数据称为链表结点，通常用头结点的指针表示链表，末结点指向下一个数据的指针为0。

上述单链表的结点类型可定义为：

```
struct node  
{  
    int val; node *next; };
```

例A8.18 构造和使用单链表。

```
struct node { int val; node *next; };  结点类型
```

```
int main( )
```

```
{ node n4={400,0},n3={300,&n4},n2={200,&n3},n1={100,&n2},*p;
```

```
  for(p=&n1;p!=0;p=p->next) cout<<p->val<<"\t";
```

```
  cout<<endl;
```

遍历链表，图示见后页

```
  node s[ ]={{10,s+1},{20,s+2},{30,s+3},{40,s+4},{50,s+5},{60,0}};
```

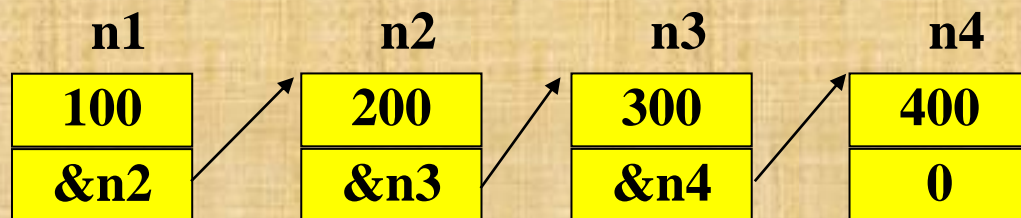
```
  for(p=s;p!=0;p=p->next) cout<<p->val<<"\t"; cout<<endl;
```

```
  return 0;
```

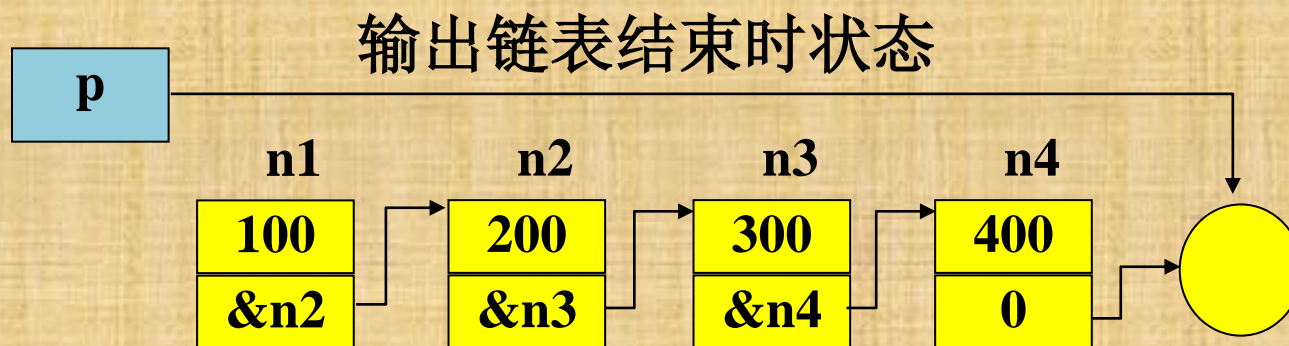
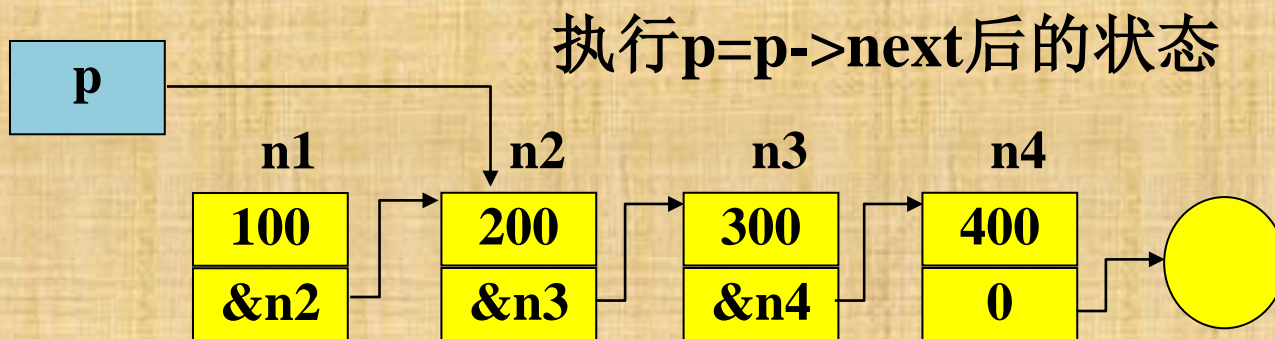
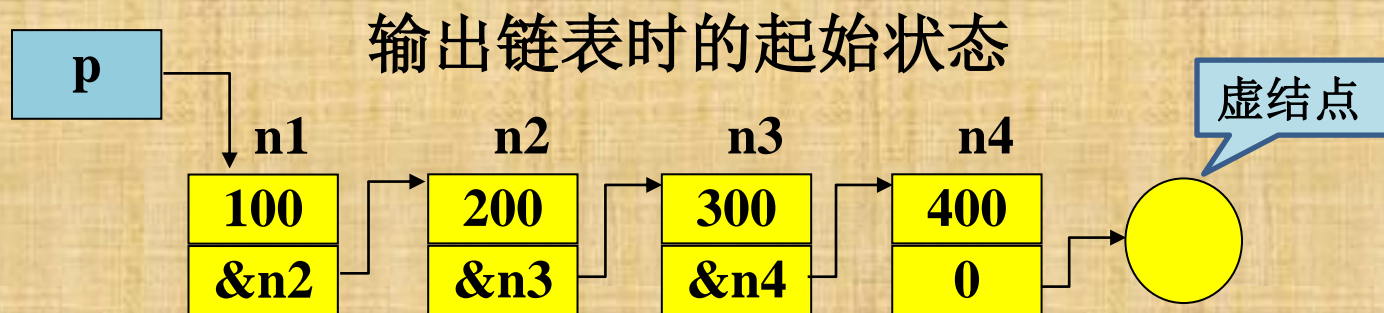
```
}
```

构造静态链表

构造静态链表



注1：在程序中通过指向结点的指针p使用成员val和next时一般不用(*p).val和(*p).next，而是习惯上使用p->val和p->next。



注2: 链表的指针从一个结点移到后一个结点不能用操作“ $p++$ ”，而要用操作“ $p=p \rightarrow next$ ”。

注3: 链表末结点的后向指针 $next$ 必须是0指针，表示后面不指向其它结点。

下面是一个比较完善的处理单链表的程序例子。

例C8.19 单链表的处理。 (主函数)

```
struct node { int val; node *next; };
node *insert(node *h,node *p); 在链表末尾添加结点
node *create( );               创建链表
void print(node *h);           输出链表
node *delNode(node *h,int x);  删除链表结点
void delAll(node *h);          删除链表

int main( )
{ node *head; int value;
  head=create( ); print(head); .....
  head=delNode(head,value); print(head); .....
  node *p=new node; p->val=value;
  head=insert(head,p); print(head);
  delAll(head); .....
}
```


例C8.19 单链表的处理。 (子函数1)

node *insert(node *h,node *p) 在链表末尾添加结点

```
{ p->next=0;
  if(h==0) h=p;
  else
  { node *q; for(q=h;q->next!=0;q=q->next); q->next=p; }
  return h;
}
```

找末结点

node *create() 创建链表

```
{ int value; node *h=0,*p;
  while(cin>>value)
  { p=new node; p->val=value; h=insert(h,p); }
  cin.clear(); 清除输入出错标志
  return h;
}
```

例C8.19 单链表的处理。 (子函数2)

void print(node *h) 输出链表

```
{ while(h!=0) { cout<<h->val<< "\\t"; h=h->next; } ..... }
```

node *delNode(node *h,int x) 删除链表结点

```
{ node temp={0,h}; temp为引导结点
```

```
node *q=&temp,*p=h;
```

扫描链表查找结点

```
while(p!=0&& p->val!=x) { q=p; p=p->next; }
```

```
if(p!=0) { q->next=p->next; delete p; }
```

```
else cout<< "链表中没有值: "<<x<<endl;
```

```
return temp.next;
```

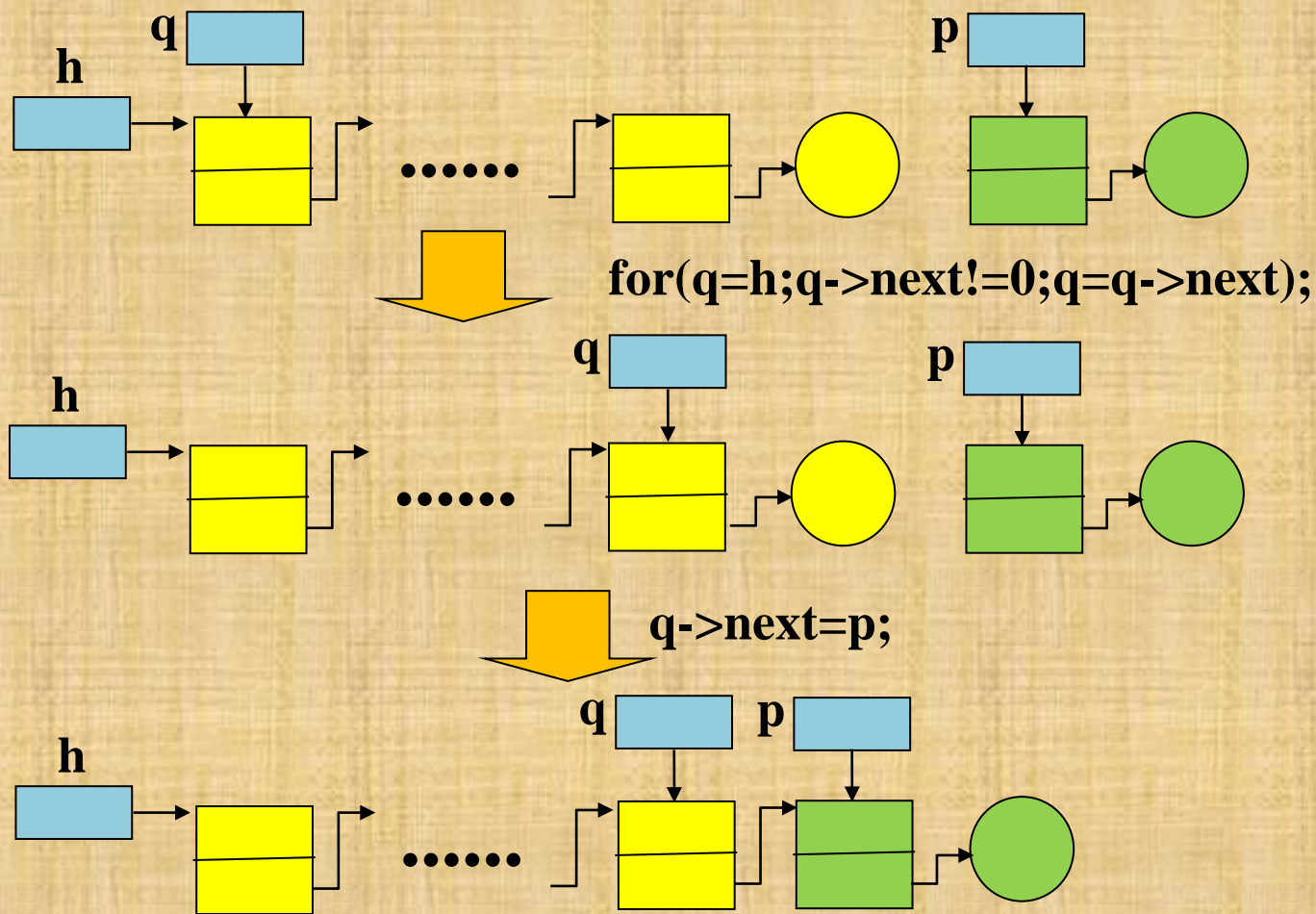
```
}
```

q为p的前一个结点,删p结点

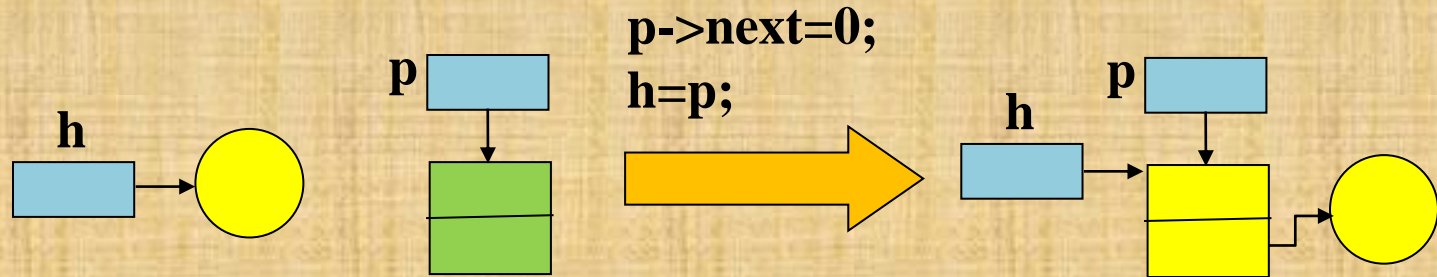
void delAll(node *h) 删除链表

```
{ node *p; while(p=h) { h=h->next; delete p; } }
```

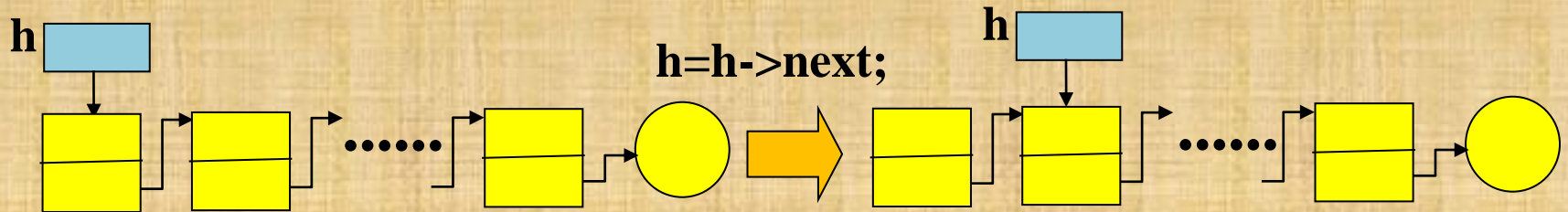
insert中 $h \neq 0$ 时的操作



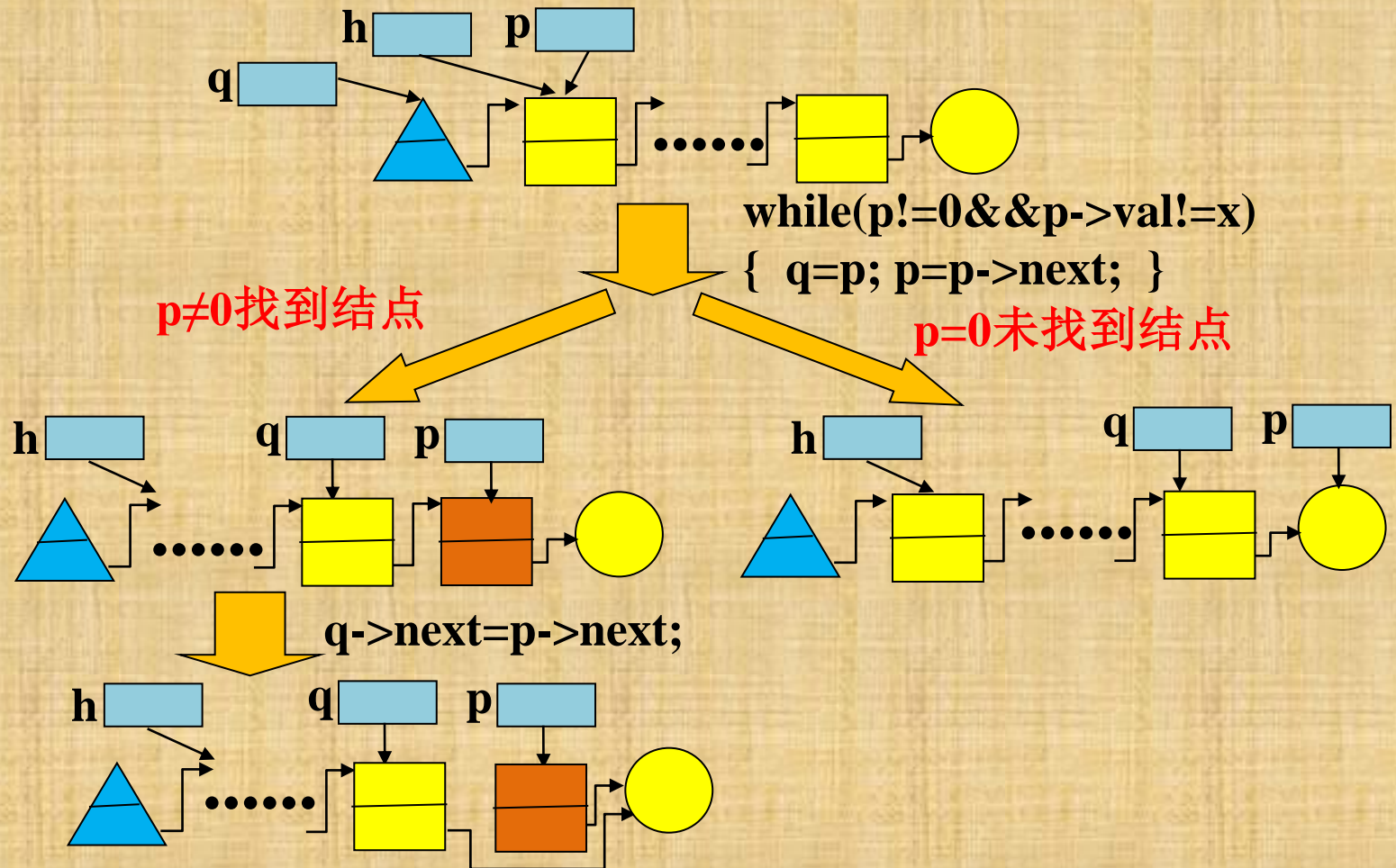
insert中h=0时的操作



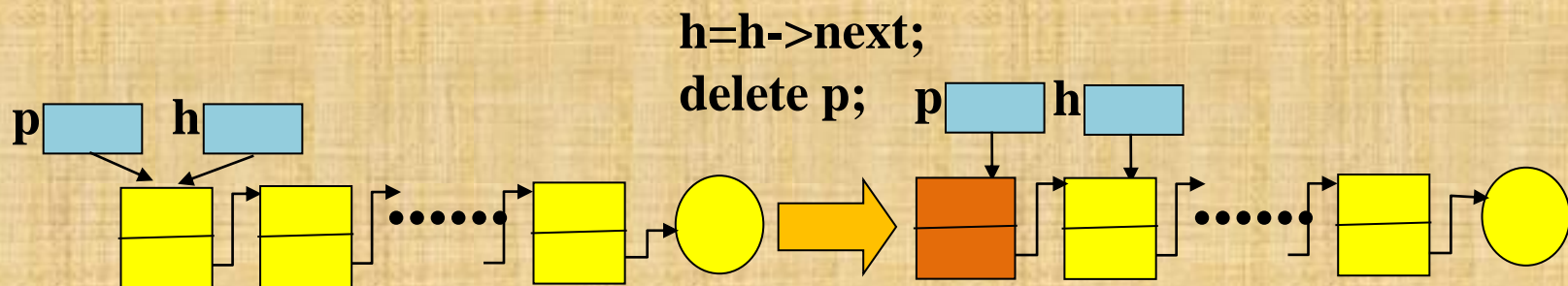
print中移向下一结点操作



delNode中查找及删除操作



delAll中删除头结点操作



注4：在链表中插入和删除结点需要使用和改动前驱结点，若插入和删除的是头结点，则要使用和改动头指针。

8.6 程序中的指针错误用法

本节介绍一些涉及程序上下文的指针使用错误

1、利用指针参数传出指针值

利用指针参数传出指针值错误程序

```
void getInt(int *p)
{ cin>>*p; }
void getString(char *s)
{ char *p=new char[80]; cin>>p; s=p; }
int main( )
{ int a; char *ss= "Hello";
  getInt(&a); getString(ss);
  cout<<a<< "\t"<<ss<<endl;
  return 0;
}
```

单向传递实参指针值

8.6 程序中的指针错误用法

本节介绍一些涉及程序上下文的指针使用错误

1、利用指针参数传出指针值

利用指针参数传出指针值错误程序

```
void getInt(int *p)
{ cin>>*p; }
void getString(char *&s)
{ char *p=new char[80]; cin>>p; s=p; }
int main( )
{ int a; char *ss= "Hello";
  getInt(&a); getString(ss);
  cout<<a<< "\t"<<ss<<endl;
  return 0;
}
```

改动为

2、忽略指针变动

忽略指针变动错误程序1

```
int main( )  
{ int a[10],*p=a;  
  while(p<a+10) cin>>*p++;  
  while(p<a+10) cout<<*p++<< "\\t";  
  cout<<endl; return 0;  
}
```

p指针已发生改动，
不再指向a[0]

2、忽略指针变动

忽略指针变动错误程序1

```
int main( )  
{  int  a[10],*p=a;  
    while(p<a+10) cin>>*p++;  
    while(p<a+10) cout<<*p++<< "\\t";  
    cout<<endl; return 0;  
}
```

增加该语句

p=a;

忽略指针变动错误程序2

```
int sum(int *x,int n)
{  int s=0;
   while(x<x+n) s+=*x++;
   return s;
}
int main( )
{  int a[10]={1,3,5,7,9,2,4,6,8,0};;
   cout<<sum(a,10)<<endl;
   return 0;
}
```

此**x**随着循环而发生变化，表达式**x<x+n**永远成立

忽略指针变动错误程序2

```
int sum(int *x,int n)
{  int s=0,*pend=x+n;
    while(x<pend) s+=*x++;
    return s;
}
int main( )
{  int a[10]={1,3,5,7,9,2,4,6,8,0};;
    cout<<sum(a,10)<<endl;
    return 0;
}
```

增加

改动为

或sum改为:

```
int sum(int *x,int n)
{  int s=0; while(n-->0) s+=*x++; return s; }
```


3、函数返回撤销的量

函数返回撤销的量错误程序

```
char *toHex(unsigned int x)
```

```
{  char s[40]="",hex[ ]="0123456789ABCDEF"; int i=0,j;  
  do{ s[i++]=hex[x%16]; x/=16; } while(x!=0);  
  for(j=0,i--; j<i; j++, i-- ) { char ch=s[j]; s[j]=s[i]; s[i]=ch; }  
  return s;  
}
```

该数组即将撤销

```
int &f(int n)
```

```
{  int s=0; while(n>0) s+=n--;  return s; }
```

```
int g( ) { return 0; }
```

```
int main( )
```

```
{  cout<<toHex(1234)<<endl;
```

```
  cout<<f(100)+g( )<<endl;
```

```
  return 0;
```

```
}
```

表示已撤销不存在的数组

该变量即将撤销

表示已撤销的变量(f中的s)

3、函数返回撤销的量

函数返回撤销的量错误程序

改为static类型

```
char *toHex(unsigned int x)
```

```
{  static char s[40],hex[ ]="0123456789ABCDEF"; int i=0,j;  
  do{ s[i++]=hex[x%16]; x/=16; } while(x!=0); s[i]='\0';  
  for(j=0,i--; j<i; j++, i-- ) { char ch=s[j]; s[j]=s[i]; s[i]=ch; }  
  return s;  
}
```

添加串结束标志

```
int &f(int n)
```

改为static类型

```
{  static int s=0; while(n>0) s+=n--; return s; }
```

```
int g( ) { return 0; }
```

```
int main( )
```

```
{  cout<<toHex(1234)<<endl;
```

```
  cout<<f(100)+g( )<<endl;
```

```
  return 0;
```

```
}
```

4、一次性函数

一次性函数错误程序

```
char *toHex(unsigned int x)
{
    static char s[40]="",hex[ ]="0123456789ABCDEF"; int i=0,j;
    do{ s[i++]=hex[x%16]; x/=16; } while(x!=0);
    for(j=0,i--; j<i; j++, i-- ) { char ch=s[j]; s[j]=s[i]; s[i]=ch; }
    return s;
}

int main( )
{
    cout<<toHex(0xABCD)<<endl;
    cout<<toHex(0x10)<<endl;
    return 0;
}
```

数组s在程序运行中只初始化一次，第2次调用函数就是残留值了

该调用时toHex的静态局部数组s具有残留字符串 "ABCD"

4、一次性函数

一次性函数错误程序

```
char *toHex(unsigned int x)
{
    static char s[40],hex[ ]="0123456789ABCDEF"; int i=0,j;
    do{ s[i++]=hex[x%16]; x/=16; } while(x!=0); s[i]='\0';
    for(j=0,i--; j<i; j++, i-- ) { char ch=s[j]; s[j]=s[i]; s[i]=ch; }
    return s;
}

int main( )
{
    cout<<toHex(0xABCD)<<endl;
    cout<<toHex(0x10)<<endl;
    return 0;
}
```

新的字符串后面加上
串结束标志

5、○行指针参数与指针数组参数混淆

行指针参数与指针数组参数混淆错误程序

```
void print(double **pA,int m,int n)
{   for(int i=0;i<m;i++,cout<<endl)
        for(int j=0;j<n;j++) cout<<pA[i][j]<< "\\t";
}

int main( )
{   double A[3][4]={1,2,3,4},{2,0,-1,5},{0,-2,1,3}};
    print(A,3,4);
    return 0;
}
```

实参A为行指针double(*)[4]类型
形参pA为指向指针的指针double**

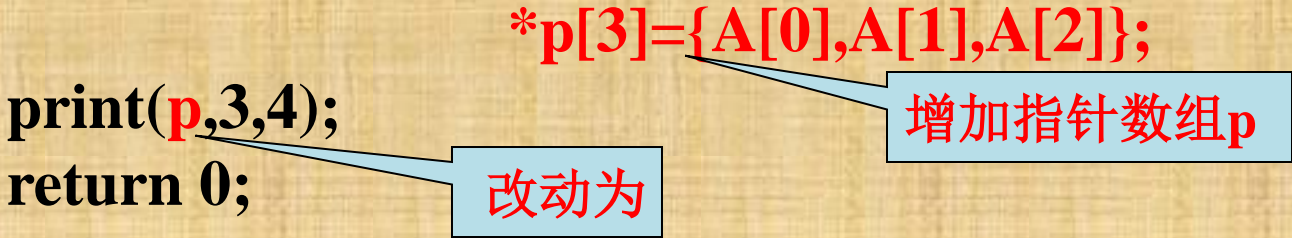
5、○行指针参数与指针数组参数混淆

行指针参数与指针数组参数混淆错误程序

```
void print(double **pA,int m,int n)
{  for(int i=0;i<m;i++,cout<<endl)
    for(int j=0;j<n;j++) cout<<pA[i][j]<< "\\t";
}

int main( )
{  double A[3][4]={1,2,3,4},{2,0,-1,5},{0,-2,1,3}},
    *p[3]={A[0],A[1],A[2]};

    print(p,3,4);
    return 0;
}
```



The diagram includes two callout boxes with arrows pointing to the code. One box points to the variable **p** in the `print` function call, and the other points to the pointer array declaration `*p[3]`.

改动为

增加指针数组p

第9章 预处理命令

通常C++程序与C语言程序在转化过程中，在编译之前多了一步源程序的替换和选择的过程，称为**预处理**，也称预编译。

常用的预处理命令有以下3种命令

- 包含命令
- 宏命令
- 条件编译命令

包含命令形式:

include <iostream>

宏命令形式

define MAXINT 0x7fffffff

条件编译命令形式

ifndef MAXINT

define MAXINT 0x7fffffff

endif

注1: 预处理命令总是以#引导, 并且单独占一行, 后面不加分号“;”。

9.1 包含命令

包含命令就是在预处理时将include命令后的文件内容嵌入到该命令的位置，即用文件内容替换该命令。

包含命令的3种形式：（包含的文件称头文件）

`# include <iostream>` 指定目录中的头文件

`# include "MB9_1.cpp"` 当前目录中的头文件

`# include "D:\\VC++\\file1.h"` 指定路径中的头文件

注1：包含命令的头文件用双引号括起来，则预处理时先在源程序当前目录中查找头文件，若查找不到，再到软件指定的头文件目录中去查找。

例**B9.1** 演示自定义头文件的用法。

// 头文件 MB9_1.h

```
int monthDay[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};  
inline bool isLeapYear(int year)  
{ return (year%4==0)&&(year%100!=0)||(year%400==0); }
```

// 源程序文件 MB9_1.cpp

include <iostream>

using namespace std;

include "MB9_1.h"

int main()

```
{ .....  
    for(total=day,i=1;i<month;i++) total+=monthDay[i];  
    if(month>2&&isLeapYear(year)) total++;  
    ..... }
```

注2： 当一个程序中需要包含多个文件时必须写多个包含命令，每个命令占一行。一般来讲，多个头文件的包含命令与次序无关。

注3： 当查看预处理结果时，若保留包含命令“#include <iostream>”，头文件会在预处理结果文件中插入很多我们不想看的内容，所以删去该包含命令及配套的using说明再查看预处理结果比较好。

9.2 宏定义

宏定义就是给一些数据或式子取名字，所取的名字就是宏名。宏定义分**无参宏定义**和**带参宏定义**。

无参宏定义形式：

```
# define MAXINT 0x7fffffff
```

带参宏定义形式

```
# define mul(x,y) x*y
```

取消宏定义形式

```
# undef MAXINT
```

```
# undef mul
```

注1：一个宏定义在程序中单独占一行，多个宏定义必须占多行。

9.2.1 无参宏定义

例B9.2 演示无参宏定义的使用。

```
# define PI 3.141592653589793    定义无参宏 PI
# define R 50                    定义无参宏 R
# define CIRCUMFERENCE 2*PI*R    定义宏CIRCUMFERENCE
int main( )
{
    cout<< "PI="<<PI<<endl;    "PI="不作宏替换,PI进行宏替换
    cout<< "R="<<R<<endl;      "R="不作宏替换,R进行宏替换
    cout<< "2*PI*R="<<2*PI*R<<endl;    PI和R作宏替换
    cout<<"CIRCUMFERENCE="<<CIRCUMFERENCE<<endl;
# undef R                        取消宏定义 R
# define R doubleVar            重新定义宏名 R为 doubleVar
    double doubleVar=1.41428;
    cout<<"R"<<"\t"<<R<<endl;    R作宏替换,替换为doubleVar
}
```

注2: 在宏定义中可以使用已定义的宏名。

注3: 宏替换只是单纯的文本串替换, 不做语法检查。

宏替换只是单纯的文本串替换，不作语法检查

```
# define e 2.71828 ;
```

```
# define A 3+5
```

```
# define B A*A
```

```
int main( )
```

```
{    cout<<e<<endl;
```

```
    cout<<B<<endl;
```

```
    return 0;
```

```
}
```



```
int main( )
```

```
{    cout<<2.71828; <<endl;
```

```
    cout<<3+5*3+5<<endl;
```

```
    return 0;
```

```
}
```

9.2.2 带参宏定义

带参宏定义除了替换文本，也替换参数。

若有宏定义

```
# define area(r) 3.1416*r*r
```

宏替换

```
s=area(100);
```

过程为

宏名替换

```
3.1416*r*r
```

再参数替换

```
3.1416*100*100
```

最后效果为

```
s=3.1416*100*100;
```

带参宏定义除了替换文本，也替换参数。

例**B9.3** 演示带参宏定义的使用。

```
# define PI 3.141592653589793
```

```
# define AREA(r) PI*r*r
```

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

```
# define AREA2(r) PI*(r)*(r)
```

```
# define MAX2(x,y) ( (x)>(y)?(x):(y) )
```

```
# define AREA3 (r) PI*r*r
```

```
int main( )
```

```
{  cout<< ...<<AREA(10)<<endl; 替换为3.14....*10*10
```

```
    cout<<...<<AREA(4+6)<<endl; 替换为3.14....*4+6*4+6
```

```
    cout<<...<<AREA2(4+6)<<endl; 替换为3.14....*(4+6)*(4+6)
```

```
    int a =MAX(3,-2)+7;      替换为 a=3>-2?3:-2+7
```

```
    .....
```

```
    cout<<...<<MAX2(3,-2)+7<<endl; 替换为((3)>(-2)?(3):(-2))+7
```

```
    .....
```

```
    return 0;
```

```
}
```

上述程序中，若有语句

```
cout<<"MAX(3,-2)+7="<<MAX(3,-2)+7<<endl;  
cout<<AREA3(10)<<endl;
```

将产生语法错误，因为宏替换后成为如下语句

```
cout<<"MAX(3,-2)+7="<<3>-2?3:-2+7<<endl;  
cout<<(r) 3.141592653589793*r*r (10)<<endl;
```

替换后的语句中第2句显然错误，第1句按照运算符的优先级等价地加上括号成为如下语句

```
((cout<<"MAX(3,-2)+7=")<<3)>-2)?3:((-2+7)<<endl);
```

语句中表达式 `((-2+7)<<endl)` 显然错误，而表达式

`((cout<<"MAX(3,-2)+7=")<<3)` 结果为 `cout`，由于 `cout>-2` 有错，故整句语句有严重错误。

注4：定义带参数的宏时，参数表的左括号“(”必须紧跟在宏名后面，中间不能有空格。

注5：带参宏定义中的带参字符串若加上括号，串中的参数也加上括号，就可以避免宏替换后由于运算符优先级原因造成的错误计算。

○预处理命令必须单独占一行，可以使用续行符(\)将下一行接在上一行后面，如

宏定义

```
# define out(a,n) { for(int i=0;i<n;i++) \  
cout<<a[i]<< "\t"; \  
cout<<endl; }
```

等价于宏定义

```
# define out(a,n) { for(int i=0;i<n;i++)cout<<a[i]<< "\t";cout<<endl;}
```

带参宏与函数的比较

	带参宏	函数
定义形式	#define M(a,b) (a)*(b)	int M(int a,int b) { return a*b; }
处理过程	预处理阶段	编译阶段
语法检查	源程序级的单纯替换， 无语法检查	按语法编译成指令， 有语法检查
效 果	插入代码	使用调用指令

9.3 条件编译命令

条件编译可以根据所提供的条件选择哪部分需要编译，哪部分忽略。

条件编译命令常见的3种形式：

1、 # ifdef TOUPPER

```
ch=(ch>= 'a' && ch<= 'z') ? ch-32:ch; TOUPPER定义编译
```

```
# else
```

```
ch=(ch>= 'A' && ch<= 'Z') ? ch+32:ch; TOUPPER未定义编译
```

```
# endif
```

2、 # ifndef TOUPPER

```
ch=(ch>= 'A' && ch<= 'Z') ? ch+32:ch; TOUPPER未定义编译
```

```
# else
```

```
ch=(ch>= 'a' && ch<= 'z') ? ch-32:ch; TOUPPER定义编译
```

```
# endif
```

3、 # if LETTER // 0 toupper ; 1 tolower

```
ch=(ch>= 'A' && ch<= 'Z') ? ch+32:ch; LETTER非零编译
```

```
# else
```

```
ch=(ch>= 'a' && ch<= 'z') ? ch-32:ch; LETTER为零编译
```

```
# endif
```

例A9.4 条件编译的使用：用凯撒密码加密与解密。

```
// #define ENCRYPTION
```

```
int main( )
```

```
{  char s[80]; int i;
```

```
    cin.getline(s,80);
```

```
    cout<< "原始字符串为: \n"; cout<<s<<endl;
```

```
# ifdef ENCRYPTION    若宏名ENCRYPTION已定义，编译下面加密代码
```

```
    for(i=0;s[i];i++)
```

```
        if(s[i]>='A' && s[i]<='W' || s[i]>='a' && s[i]<='w') s[i]+=3; 字母循环右移3位
```

```
        else if(s[i]>='X' && s[i]<='Z' || s[i]>='x' && s[i]<='z') s[i] -=23;
```

```
    cout<< "加密后字符串为: \n"; cout<<s<<endl;
```

```
# else    若宏名ENCRYPTION未定义，编译下面解密代码
```

```
    for(i=0;s[i];i++)
```

```
        if(s[i]>='D' && s[i]<='Z' || s[i]>='d' && s[i]<='z') s[i] -=3; 字母循环左移3位
```

```
        else if(s[i]>='A' && s[i]<='C' || s[i]>='a' && s[i]<='c') s[i]+=23;
```

```
    cout<< "解密后字符串为: \n"; cout<<s<<endl;
```

```
# endif
```

```
    return 0;
```

```
}
```

9.4 ○程序的编译与连接

C++源程序生成可执行文件的过程:

- **预处理:** 编译之前在源程序上做一些替换和选择, 预处理后依然是源程序文件。注解在预处理之后替换成空格。
- **编译:** 以函数为单位逐个函数编译成相应的指令, 生成中间代码文件(文件扩展名为.obj)。
- **连接:** 将各个函数指令、全局变量和静态局部变量进行拼装, 生成一段完整的程序可执行代码, 并存放在可执行文件中(文件扩展名为.exe)。

C++源程序生成可执行文件过程图示:

