

第6章 函数

代码的封装方法

- 独立性 & 通用性 -



计算机与程序设计基础 (C++)

1 函数的基本语法

函数的基本语法

- 函数是一系列C++语句的集合
- 一个函数通常完成一个特定的功能
- 利用函数组织程序可以简化代码，实现代码独立性和重用

声明

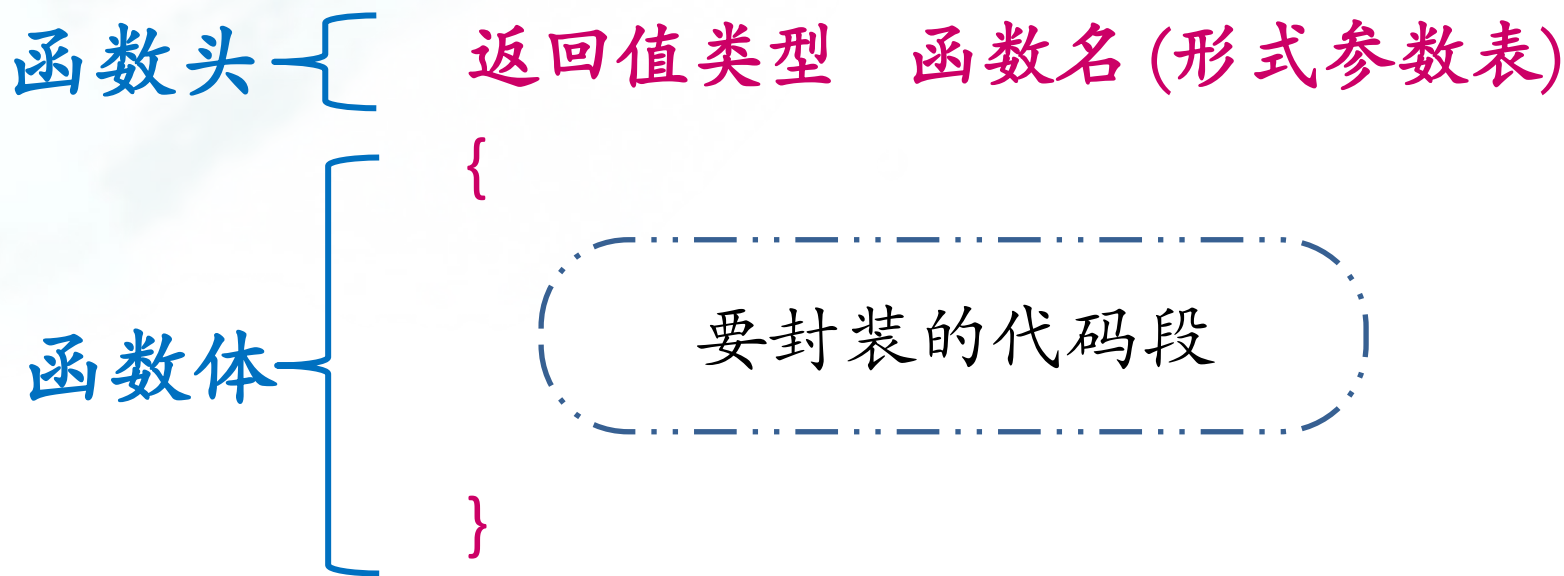
定义

调用

函数定义de 语法框架

■ 函数定义：封装实现该函数功能的代码

— 语法框架：函数头+函数体(块定义符{}+代码段)



函数间的数据流动

■ 函数被调用时:

■ 从主调函数接收要处理的数据

■ 函数被执行时

■ 处理从主调函数接收的数据

■ 函数调用结束时:

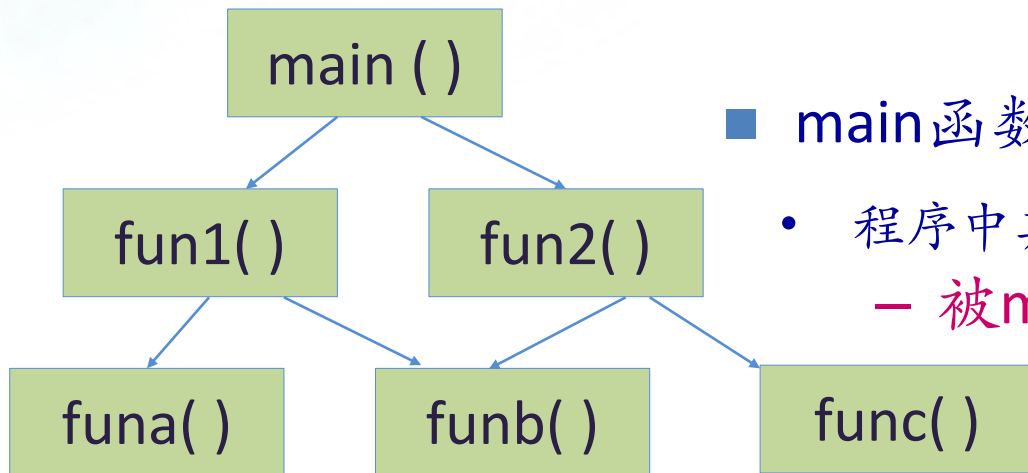
■ 向主调函数返回数据的处理结果

- 语法框架
 - 形式参数表

- 语法框架
 - 返回值类型
- 代码编写
 - **return** 返回值;

■ main函数是程序执行入口

- 程序中其他函数的执行方法
 - 被main函数直接或间接调用



函数头：形式参数表

- 形式参数表：代码运行所需要的外部数据的入口
- 语法格式：(类型 参数1, 类型 参数2,)

返回值类型 函数名 (形式参数表)

{

要封装的代码段

}

- 如果不需要外部数据
 - 语法格式：(void)或()

函数头：返回值类型

- **返回值**：被调函数向主调函数提供的数据
- 函数是否有返回值取决于函数的功能
 - 若有，返回值类型=函数返回值的数据类型
 - ✓ 必须在代码中使用“**return 返回值;**”语句
 - 若无，返回值类型=**void**

返回值类型

函数名 (形式参数表)

{

要封装的代码段

}

注意：函数返回值的类型一致性

- 对于有返回值的函数
 - `return` 语句中的返回值类型和函数类型（函数头部定义的返回值类型）应保持一致
 - 如果两者不一致，返回值自动强制转换成函数类型提供给主调函数

return 语句在void函数中的使用

- 对于没有返回值的函数
 - 当需要在程序指定位置退出时，可以在该处放置一个：`return ;`

函数调用

函数名(实参1, 实参2, 实参3.....);

□ 函数调用要点:

- 实际参数是主调函数在调用函数时依照函数的形参所提供的参数
- 实际参数可以是常量、变量或表达式，之间用逗号隔离
- 只有在完成函数的定义或给出函数声明后，才能调用函数



函数声明、定义、调用

函数声明：函数头，一条语句

数据类型 函数名 (类型 1 形参1, 类型 2 形参2,);

函数定义：函数头和函数体，完整的代码

数据类型 函数名 (类型 1 形参1, 类型 2 形参2,)

{ }

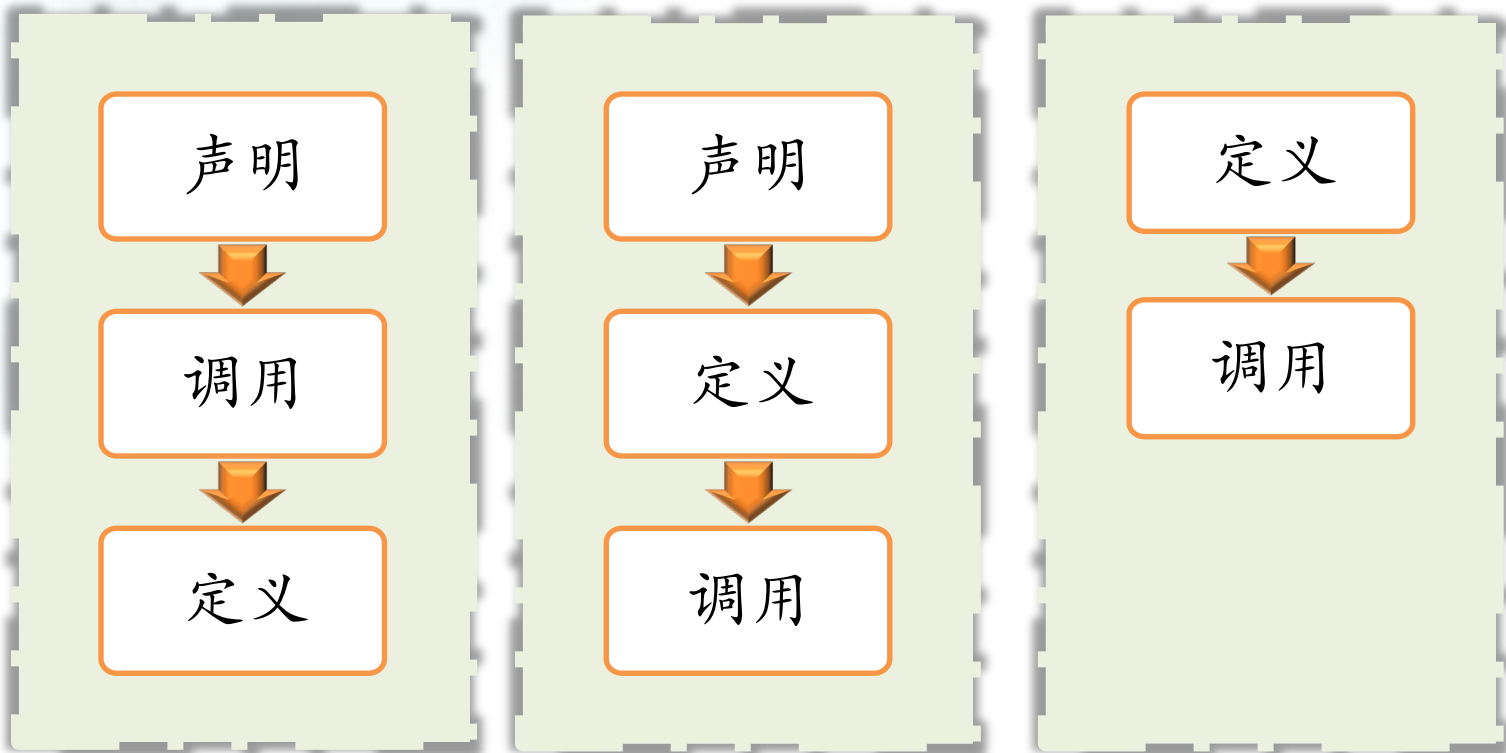
- 函数必须独立定义，不可以定义在其它函数的函数体内

函数调用：表达式

函数名 (实参1, 实参2, 实参3,)

小结：应用顺序

- 函数调用必须在函数声明或定义之后才合法
- 先给出函数声明后，函数定义可以出现在函数调用之后



小结:

- 有参函数

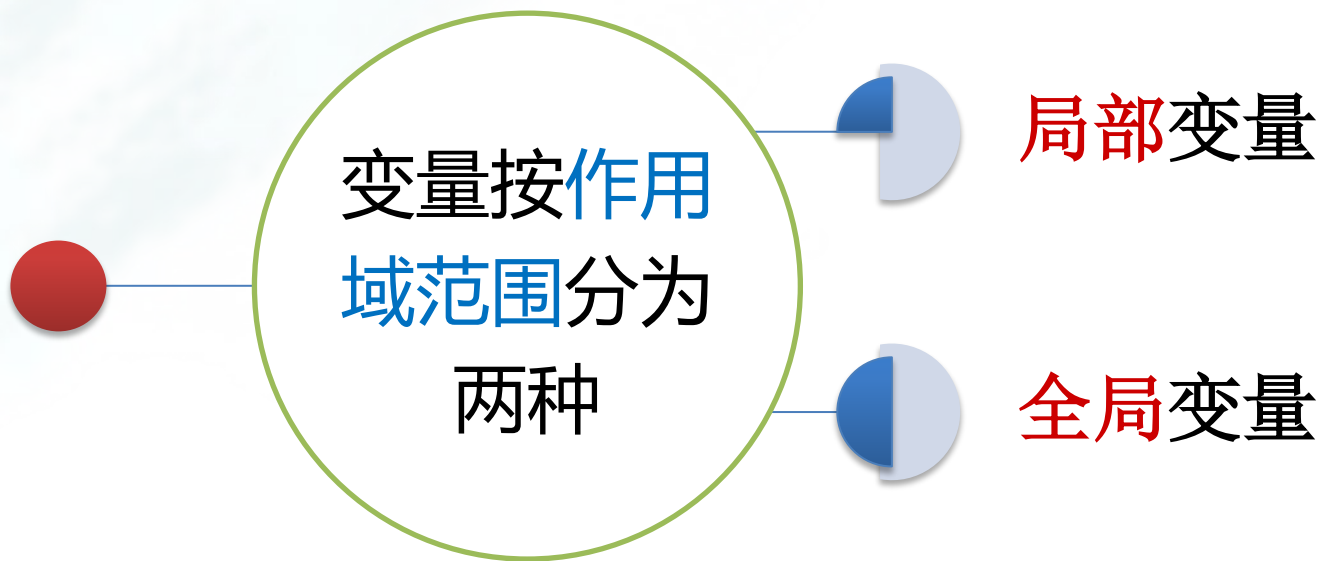
- 定义: 数据类型 函数名(形参表){.....}
- 调用: 函数名(实参表);

- 无参函数

- 声明: 数据类型 函数名();
- 定义: 数据类型 函数名(){.....}
- 调用: 函数名()

2 变量的作用域和生存期

变量的作用域



局部变量

- 局部变量是在函数内或者块内{}定义说明的，其作用域仅限于函数内或者块内，离开该函数或块后再使用这种变量是非法的。

```
int fun1(int a)
{
    int b,c;
    .....
}

int main()
{
    int a,b,c;
    .....
}
```

- a,b,c 三个变量是局部变量，且作用域在函数fun1体内
- main函数中的a,b,c3个变量也是局部变量，且作用域在main函数内
- 虽然变量名与函数fun1中的变量名相同，但其作用域不同，所以不会出现同名冲突，也不可以相互替代。

全局变量

- 全局变量是在函数体之外定义的变量
- 作用域是从“定义处”到“文件结束”处
- 如果用户在定义时不显式给出初始化值，全局变量初始化为0

```
int a, b;  
void fun1()  
{ ..... }  
double x, y;  
int fun2()  
{ ..... }  
int main()  
{ ..... }
```

• 在函数体外定义，全局变量

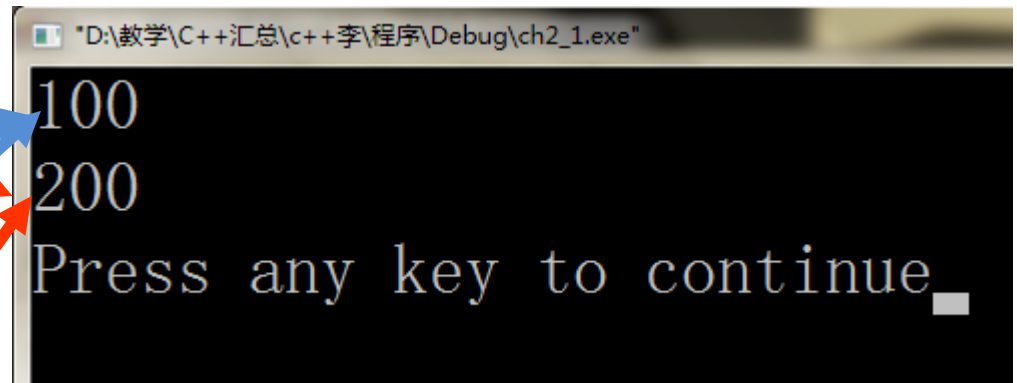
• 在函数体外定义，全局变量

- a,b定义在源程序最前面，在fun1,fun2 及main内不加说明也可使用
- x,y 定义在函数fun1之后，在fun2及main内不加说明也可使用

全局变量

```
#include<iostream>
using namespace std;
int n;//全局变量
void func(){
    n*=2;
}
int main(){
    n+=100;
    cout<<n<<endl;
    func();
    cout<<n<<endl;
    return 0;}
```

- 全局变量：在多个函数间共享数据



The screenshot shows a Windows command prompt window titled "D:\教学\C++汇总\c++李\程序\Debug\ch2_1.exe". The output of the program is displayed on a black background with white text. The first line is "100", the second line is "200", and the third line is "Press any key to continue_". Two red arrows point from the code on the left to the output: one from the first 'cout' statement to the first line of output, and another from the second 'cout' statement to the second line of output. A blue arrow points from the 'n' variable in the 'main' function to the first line of output.

```
"D:\教学\C++汇总\c++李\程序\Debug\ch2_1.exe"
100
200
Press any key to continue_
```

函数声明域

- 函数声明域中，形参作用域始于"("，结束于")"

例如：求两数最大值的函数可声明为以下3种形式：

```
int max ( int a , int b );
```

```
int max ( int x, int y );
```

```
int max ( int , int );
```

- x, y的作用域仅在于此，不能用于程序正文其它地方。因而**声明时可以使用和定义不同的形参名**

//返回两个整数中的较大值

```
int max ( int a, int b )  
{    return (a>=b?a:b); }
```

- 形参的作用域仅在于此,因而可有可无。因而**函数声明中，形参表可只给出形参类型，形参名可省略**

变量的生存期

- 指的是变量从获得空间到空间释放之间的期间
- 变量只有在生存期中、并且在其自己的作用域中才能被访问

存储类型不同，变量的生存期也不同

变量的存储类型

■ 变量的定义格式：**存储类型** 数据类型 变量名

• 存储类型的说明符

— auto

— register

— static

— extern

➤ 用auto和register修饰的称为自动存储类型

➤ 用static修饰的称为静态存储类型

➤ 用extern修饰的称为外部存储类型

auto变量

- 自动变量用关键字auto作存储类别的声明，但常常省略
- 生存期：**所在的块({})的执行时间**

- 函数中的局部变量（函数中的形参和在函数中定义的变量，包括在复合语句中定义的变量），若未声明为static存储类别都属于auto变量。
 - 在调用该函数时系统会给它们分配存储空间
 - 在函数调用结束时就自动释放这些存储空间

static 变量

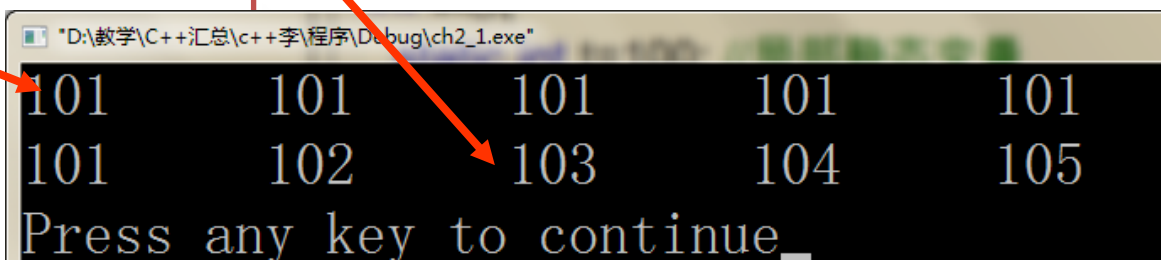
- 存储类型为static的变量称为静态变量，存放在静态存储区
- 如果程序未显式给出静态变量的初始化值，初始化为0
- 不论是局部静态变量还是全局静态变量，其生存期都等于整个程序执行期

- Ex6.28：在函数体内设置一个变量记录函数被调用的次数
 - 函数体内变量都是局部变量
 - 如果希望局部变量在函数调用结束仍然生存，就必须将该局部变量声明为static类型

自动变量与静态变量的区别

```
#include <iostream>
using namespace std;
int st(){
    static int t=100; //局部静态变量
    t++;    return t;
}
int at(){
    int t=100; //自动变量
    t++;    return t;
}
int main(){
    int i;
    for(i=0;i<5;i++) cout<<at()<<"\t";
    cout<<endl;
    for(i=0;i<5;i++) cout<<st()<<"\t";
    cout<<endl;
    return 0;}
```

- 局部静态变量如果显式给出初始化值，则在该块第一次执行时完成，且只进行一次



"D:\教学\C++汇总\c++李\程序\Debug\ch2_1.exe"

101	101	101	101	101
101	102	103	104	105

Press any key to continue

3 函数参数传递

函数的关键语法细节

函数调用

函数名 (实参1, 实参2, 实参3,)

2. 实参vs形参类型，匹配？

函数定义

返回值类型 函数名 (类型 1 形参1, 类型 2 形参2,)

{

.....

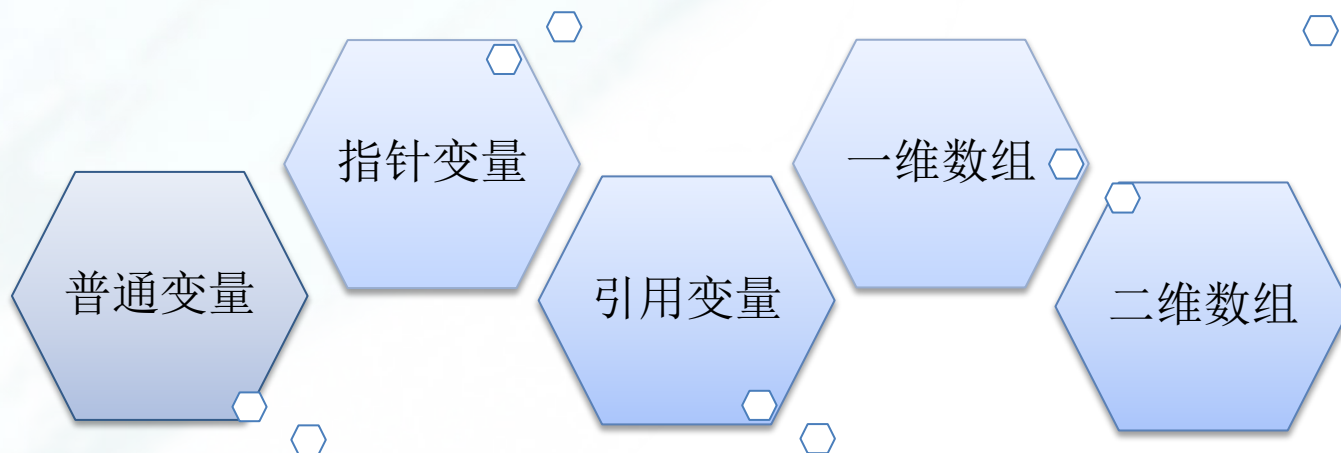
.....

}

1. 形参数量？

3. 数据的访问方法vs形参类型？

函数的关键语法细节



函数声明:

- ✓ 依据功能选择形参类型

问题1

问题2

函数调用:

- ✓ 实参要与形参类型匹配

函数定义:

- ✓ 数据的访问方法由形参类型决定

问题3

变量的值交换函数

```
void f 1( int x , int y ) //参数传递式 int x=a, int y=b;
```

```
{      int t;          t=x,    x=y,    y=t;    }
```

```
void f2 ( int & x , int & y ) //参数传递式 int &x=a,int &y=b;
```

```
{      int t;          t=x,    x=y,    y=t;    }
```

```
void f 3( int * x , int * y ) //参数传递式 int *x=&a,int *y=&b
```

```
{      int t;          t= *x,  *x=*y, *y=t;    }
```

```
int main()
```

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

```
    a=12,b=7;    f1( a , b );    cout<<a<<'\t'<<b<<endl;
```

```
    a=12,b=7;    f2( a , b );    cout<<a<<'\t'<<b<<endl;
```

```
    a=12,b=7;    f3( &a , &b );    cout<<a<<'\t'<<b<<endl;
```

```
    return 0;
```

```
}
```

值传递：普通变量作为函数形参

```
void f 1( int x , int y ) //参数传递式 int x=a, int y=b;
```

```
{  
    int t;  
    t=x,   x=y,   y=t;  
}
```

```
int main()  
{  
    int a=12,b=7;  
    f1( a , b );  
    cout<<a<<'\t'<<b<<endl;  
    return 0;  
}
```

- 调用时，实参与形参的数据传递表达式为：int x=a, int y=b;
 - 该表达式创建临时形参变量x、y，为x、y分配临时空间
 - 将实参a、b的值复制给x、y
- a/b与x/y是独立变量。f1函数只能访问x/y的空间，不能访问a/b的空间
 - 在函数中参加运算的是形参，而实参不会发生任何改变
 - 函数调用结束时，系统回收为形参分配的临时存储单元

引用传递：引用变量作为函数形参

```
void f2( int &x , int &y ) //参数传递式 int &x=a, int &y=b;
{
    int t;
    t=x,  x=y,  y=t;
}
```

```
int main()
{
    int a=12,b=7;
    f2( a , b );
    cout<<a<<'\t'<<b<<endl;
    return 0;
}
```

- 调用时，实参与形参的数据传递表达式为： **int &x=a, int &y=b;**
 - 该表达式创建临时形参变量 **x、y**
 - 不为x、y分配临时空间
 - **x/y 与 a/b 共享存储空间**
 - **x/y 即为 a/b，形参即为实参**
- f2函数只能访问x/y
- f2函数中交换x/y的值，即是交换了a/b的值

地址传递：指针变量作为函数形参

```
void f3( int *x, int *y ) //参数传递式 int *x=&a, int *y=&b;
{
    int t;
    t=*x, *x=*y, *y=t;
}
```

```
int main()
{
    int a=12,b=7;
    f3( &a, &b );
    cout<<a<<'\t'<<b<<endl;
    return 0;
}
```

- 调用时，实参与形参的数据传递表达式为： **int *x=&a, int *y=&b;**
 - 该表达式创建临时形参变量x、y，为x、y分配临时空间
 - 将实参&a、&b的值 (a/b的地址) 复制给x、y
 - x/y指向a/b
- a/b与x/y是独立变量
- f3函数以 “*指针名” 的形式 间接访问a/b的空间，从而交换a/b的值



小结1：函数定义时，形参与数据的匹配

数据规模	可选的形参的类型
一个形参接收一个数据	基本类型变量 引用变量 指针变量
一个形参接收“多个”数据	数组 指针变量

数据操作	可选的形参的类型
函数无需改变数据的值	基本类型变量，数组
函数需改变数据的值	数组，指针变量/引用变量



小结2：函数调用时，实参与形参的匹配

形参的数据类型		匹配的实参
基本类型 变量	int char double	同类型的 变量、常量、数值表达式
引用变量		要被引用的变量
指针变量		要访问的内存空间的地址
数组		数组首地址

排序函数（要排序的数据由调用者提供）

- 一维数组长度可缺省
- C++只传递数组首地址，而对数组边界不加检查

■ 实现方法1：形参定义为数组

```
void sort(int arr[ ], int len)  
{ 对arr数组进行升序排序;}
```

- 独立定义参数提供数组长度

■ 实现方法2：形参定义为指针

```
void sort(int *arr, int len)  
{ 对arr数组进行升序排序;}
```



一维数组作为函数形参

- 形参格式：数据类型 数组名[]
 - 实参格式：数组首地址
- 数组名作函数参数时所进行的传送是地址的传送，也就是把实参数组的首地址赋予形参数组名。
 - 形参数组名取得该首地址之后与实参数组为同一数组，共同拥有一段内存空间。
 - 形参数组的改变会直接影响到实参数组

排序函数的实现方法1：形参定义为数组

```
void in_data( int[ ], int );    //函数声明：为数组赋值
void sort( int[ ], int ); //函数声明：对数组进行排序
void out_data( int[ ], int );  //函数声明：打印数组
const int max=1000;
int main()
{
    int a[max],n,*p=a;
    cin>>n;
    in_data( a, n );
    sort( p, n );
    out_data( &a[0], n );
    return 0;
}
```

■ 形参：数组

■ 实参：数组首地址

■ 数组名

■ 存放数组首地址的指针

■ 数组首元素地址

* 排序函数的实现方法1: 形参定义为数组

```
void sort(int arr[], int len)
{
    int i,j,temp,noswap;
    for(i=0;i<len-1;i++)
    {
        noswap=1;
        for(j=len-1;j>i;j--)
            if (arr[j]<arr[j-1])
            {
                temp=arr[j]; arr[j]=arr[j-1]; arr[j-1]=temp;
                noswap=0;
            } //end for j
        if (noswap) break;
    } //end for i
} //自下而上的交换排序
```

排序函数的实现方法2：形参定义为指针

```
void in_data( int*, int );    //函数声明：为数组赋值
void sort( int*, int );    //函数声明：对数组进行排序
void out_data( int*, int );    //函数声明：打印数组
const int max=1000;
int main()
{
    int a[max],n,*p=a;
    cin>>n;
    in_data( a, n );
    sort( p, n );
    out_data( &a[0], n );
    return 0;
}
```

- 形参：指针
- 实参：数组首地址
 - 数组名
 - 存放数组首地址的指针
 - 数组首元素地址

排序函数的实现方法2：形参定义为指针

```
void sort( int *arr, int len)
{
    int i,j,temp,noswap;
    for(i=0;i<len-1;i++)
    {
        noswap=1;
        for(j=len-1;j>i;j--)
            if ( *(arr+j)<*(arr+j-1) )
            {
                temp=*(arr+j); *(arr+j)=*(arr+j-1); *(arr+j-1)=temp;
                noswap=0;
            } //end for j
        if (noswap) break;
    } //end for i
} //自下而上的交换排序
```



排序函数的实现方法2：形参定义为指针

```
void sort(int *arr, int len)
{
    int i,j,temp,noswap;
    for(i=0;i<len-1;i++)
    {
        noswap=1;
        for(j=len-1;j>i;j--)
            if (arr[j]<arr[j-1])
            {
                temp=arr[j]; arr[j]=arr[j-1]; arr[j-1]=temp;
                noswap=0;
            } //end for j
        if (noswap) break;
    } //end for i
} //自下而上的交换排序
```


指针作为函数形参的优势

- 指针作为函数形参，为函数提供了通用、简洁的数据接口
 - 指针作为函数的形式参数，实参必须是某一内存区域的地址。这一内存区域存放的数据可能是一个变量、数组或其他用户自定义类型数据
 - 当形参指针获取数据区的地址后，就可以在函数内进行处理。



引用vs指针作为函数参数

- 函数的形参为引用时，**作为实参一个别名来使用**，对形参变量的操作就是直接对其相应的实参变量的操作。
- 使用**引用作为形参在内存中并没有产生实参的副本**，它是**直接对实参操作**，当参数传递的数据较大时，引用比用一般变量传递参数的效率和所占空间都好。
- 使用指针作为函数的形参也能达到与使用引用的效果，但是在被调函数中要给形参分配存储单元，且需要使用“*指针变量名”的形式进行运算。

综合应用：兔子的菜窖

- 有一只兔子，囤积了6种蔬菜，编号为0-5，准备过冬蔬菜的囤积数量各不相同，记录在数组t中，数组元素t[i]为编号i蔬菜的数量。为了能顺利度过冬天，兔子会检查蔬菜的囤积数量并向农夫购买蔬菜

- 请编写一个函数rep告诉兔子囤积数量最少的蔬菜编号和囤积数量最多的蔬菜编号

```
int main()
```

```
{
```

```
    int t[6]={20,180,67,96,12,40};
```

//存储蔬菜的数量

```
    int max; //数量最多的蔬菜编号
```

```
    int min; //数量最少的蔬菜编号
```

```
    .....
```

- 函数只能返回一个值
- rep函数不设返回值
- rep函数直接改变max, min的值

综合应用：兔子的菜窖

rep函数要从main函数得到的数据	形参
存储蔬菜数量的数组t	int[], 或int*
变量max, min	int&,int& 或 int*,int*

- 请编写函数rep告诉兔子囤积数量最少的蔬菜编号和囤积数量最多的蔬菜编号

```
int main()
```

```
{
```

```
    int t[6]={20,180,67,96,12,40};
```

//存储蔬菜的数量

```
    int max; //数量最多的蔬菜编号
```

```
    int min; //数量最少的蔬菜编号
```

```
    .....
```

- 函数只能返回一个值
- rep函数不设返回值
- rep函数直接改变max, min的值

综合应用：兔子的菜窖

- 主函数调用语句

```
int max,min;  
rep(t,6,max,min);
```

```
void rep(int x[], int n, int &t, int &b)  
{  
    t=0, b=0;  
    for(int i=0;i<n;i++)  
    {  
        if (x[i]>x[t]) t=i;  
        if (x[i]<x[b]) b=i;  
    }  
}
```

- 主函数调用语句

```
int max,min;  
rep(t,6,&max,&min);
```

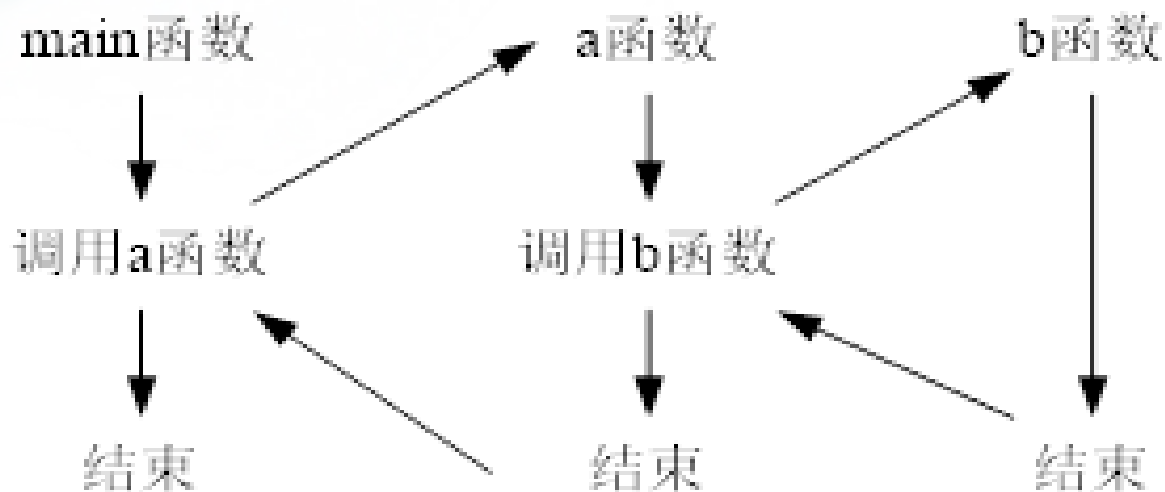
```
void rep(int *x, int n, int *t, int *b)  
{  
    *t=0, *b=0;  
    for(int i=0;i<n;i++)  
    {  
        if (x[i]>x[*t]) *t=i;  
        if (x[i]<x[*b]) *b=i;  
    }  
}
```

4 函数嵌套与递归调用



嵌套调用

- 函数不能嵌套定义，但可以在一个函数的定义中出现对另一个函数的调用。
- 函数的嵌套调用：在被调函数中又调用其它函数



递归

- 递归是一种分析和解决问题的方法和思想

原始问题可以转化为解决方法相同的新问题

新问题的规模比原始问题小

新问题又可转化为解决方法相同、规模更小的新问题

直至终结条件为止

递归设计要点

- 递归的实现：函数“自己调用自己”
 - ▣ 给出正确的递归算法
 - ✓ 算法关键1：确定递归的终止条件
 - ✓ 算法关键2：找出递推关系

- 递归定义的阶乘函数：

```
int fac(int n)
{
```

```
    if (n==0) return 1; //终止条件n=0/1
```

```
    else return( n*fac(n-1) ); //递推关系fac(n)=n*fac(n-1)
```

```
}
```

$$n! = \begin{cases} 1 & n = 0 \\ n * (n - 1)! & n > 0 \end{cases}$$

递归过程分析

- 递归函数的执行分为“**递推**”和“**回归**”两个过程
- 这两个过程由递归**终止条件**控制，即**逐层递推**，直至递归终止条件，然后**逐层回归**。
- 每次调用发生时都**首先判断递归终止条件**

递归算法例1

- 编写函数，求数组的最小值

■ 定义函数 `double f(double a[], int i)`

■ 函数功能：返回长度为*i*的数组*a*中的最小值

$$f(a, i) = \begin{cases} a[0] & i = 1 \\ \min(f(a, i - 1), a[i - 1]) & i > 1 \end{cases}$$

■ 终止条件：

□ 数组*a*长度*i=1*，函数返回值为[a\[0\]](#)

■ 递推关系：

□ 数组*a*长度*i>1*，函数返回值为函数[f\(a, i-1\)](#)和数组元素[a\[i-1\]](#)中的[较小值](#)

递归算法例1

■ 编写函数，求数组的最小值

```
double f( double a[ ], int i )  
{  
    if (i==1) return a[0];  
    else  
        if (f(a,i-1)<a[i-1]) return f(a,i-1);  
        else return a[i-1];  
}
```

$$f(a, i) = \begin{cases} a[0] & i = 1 \\ \min(f(a, i - 1), a[i - 1]) & i > 1 \end{cases}$$

5 函数深入应用

函数重载

- 本节主要讲解函数重载和带有默认形参值的函数定义的应用
 - 函数重载主要解决功能相近函数的命名问题
 - 有默认参数的函数时，实参的个数可以与形参的个数不同，实参未给定的，从形参的默认值得到值
 - 利用这一特性，可以使函数的使用更加灵活。

函数重载

- 有些函数实现的是同一类的功能，只是形参类型或者数量不同。

- 例如求数据之和，对于求2个整数、3个整数、3个双精度数的情况。定义函数时会分别设计出3个不同名的函数，其函数声明为：

```
int sum1(int a, int b); // 求2个整数之和
```

```
int sum2(int a, int b, int c); // 求3个整数之和
```

```
double sum3(double a, double b, double c); // 求3个双精度数之和
```

函数重载

- C++中，如果需要定义几个功能相似，而参数不同的函数，那么这样的几个函数可以使用相同的函数名，这就是函数重载
- 函数重载减少函数名的数量，提高程序的可读性

■ 例如求数据之和，对于求2个整数、3个整数、3个双精度数的情况。定义函数时使用相同的函数名sum。

```
int sum (int a, int b); // 求2个整数之和
```

```
int sum (int a, int b, int c); // 求3个整数之和
```

```
double sum (double a, double b, double c); // 求3个双精度数之和
```


函数重载

- 重载函数的**函数名必须相同**
- 重载函数的**形参必须不同（个数不同/类型不同）**

◆ 形参类型不同

⇒ `int add(int x,int y);`

⇒ `double add(double x,double y);`

◆ 形参个数不同

⇒ `int add(int x,int y,int z);`

⇒ `int add(int x,int y);`

函数重载

- 重载函数的**函数名必须相同**
- 重载函数的**形参必须不同（个数不同/类型不同）**

◆ 编译器不以形参名来区分

➔ `int add(int x,int y); int add(int a,int b);`

◆ 编译器不以返回值来区分

➔ `int add(int x,int y); void add(int x,int y);`

- **重载匹配规则**

- (1) 如果有严格匹配的函数，就调用该函数；
- (2) 参数内部转换后如果匹配，调用该函数；
- (3) 通过用户定义的转换寻求匹配。

带默认形参值的函数

- 有时多次调用同一函数时会使用同样的实参，C++提供简单的处理办法，**设置形参的默认值**

- 函数在调用时，对于默认参数，可以给出实参值，也可以不给出参数值
- 如果给出实参，将实参传递给形参进行调用
- 如果不给出实参，则按形参的默认值进行调用

默认参数

//延时函数,默认延时5个时间单位

```
void delay( int loops=5 )
```

```
{
```

```
    for (; loops>0; loops--);
```

```
}
```

```
int main(){
```

```
    delay(3);
```

延时3个时间单位

```
    delay( );
```

```
    return 0;
```

等同于delay(5),延时5个时间单位

```
}
```

默认参数

- 因为调用时实参取代形参是从左向右的顺序，所以默认形参值必须从右向左顺序声明
- 在默认形参值的右面不能有非默认形参值的参数



➔ `int add(int x,int y=5,int z=6);`//正确

➔ `int add(int x=1,int y=5,int z);`//错误

➔ `int add(int x=1,int y,int z=6);`//错误

默认参数

- 如果函数只有定义没有声明
 - 应在函数定义中给出默认值。
- 如果函数既有定义，又有声明
 - 应在函数声明中给出默认值
 - 不可以在声明和定义中同时指定默认值，即使默认值一样也不行

```
//函数声明中给出缺省值
int fun2 (int a, int b=10, int c=20);
void fun1()
{... }
int main()
{..... }
//定义中不再给出缺省值
int fun2(int a, int b, int c)
{... }
```



Thank You !