

第1章

C++回顾

本章内容:

■ C++特性:

- 参数传递方式（如值传递、引用传递和常量引用传递）。
- 函数返回方式（如值返回、引用返回和常量引用返回）。
- 模板函数。
- 递归函数。
- 常量函数。
- 内存分配和释放函数：new与delete。
- 异常处理结构：try, catch和throw。
- 类与模板类。
- 类的共享成员、保护成员和私有成员。
- 友元。
- 操作符重载。
- 标准模板库。

函数与参数

- 参数传递方式
 - 值传递 (Value Parameters)
 - 引用传递 (Reference Parameters)
 - 常量引用传递 (Const Reference Parameters)
- 函数返回方式
 - 值返回 (Return Value)
 - 引用返回 (Return Reference)
 - 常量引用返回 (Return Const Reference)

传值参数

程序1-1

```
int abc(int a, int b, int c)    ---形式参数
{
    return a+b*c;
}
```

函数执行前：实际参数的值→形式参数（由形式参数类型的复制构造函数(copy constructor)完成；

例：z=abc(2, x, y) ---实际参数
2 →a; x →b; y→c; ---复制构造函数

函数执行后：形式参数类型的析构函数(destructor)负责释放形式参数(实际参数的值不会修改)。

模板函数

程序1-1

```
int abc(int a, int b, int c)
{
return a+b*c;
}
```

计算一个浮点数表达式
重新编写一个相应的函数？

程序1-2

```
float abc(float a, float b, float c)
{
return a+b*c;
}
```

模板函数

程序1-3

```
template<class T>
T abc(T a, T b, T c)
{
    return a+b*c;
}
```

将参数的数据类型作为一个变量，它的值由编译器来确定。

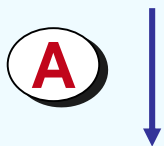
模板函数使用示例

```
main()
{
    int n=1;
    int k=1;
    int j=2;
    int m=abc(n,k,j);//
}
```

传值传递模式- 1

代码

```
m=20;  
x = 10;  
y = 15;  
z=Abc (m, x, y);
```



```
int Abc(int a, int b, int c)  
{  
    return a+b+b*c+4;  
}
```

在 **A** Abc方法前

m	20
x	10
y	15
z	?

存储器
状态

A. 在执行方法前，局部量的不存在。



传值传递模式- 2

代码

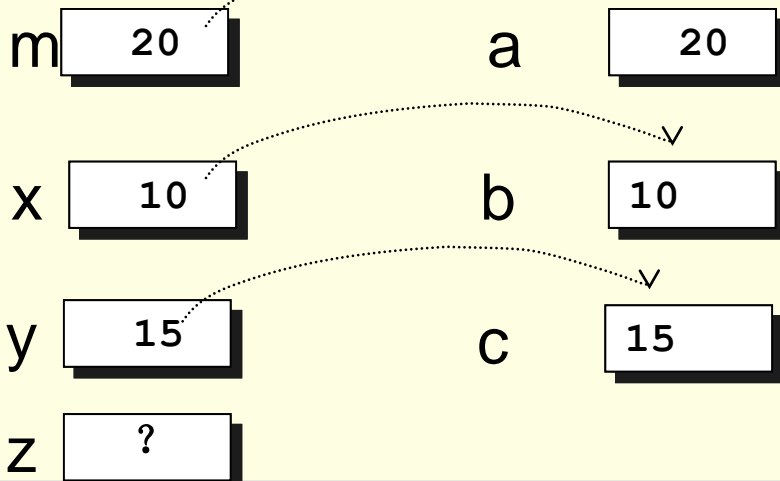
```
m=20;  
x = 10;  
y = 15;  
z=Abc (m, x, y) ;
```

```
int Abc(int a, int b, int d)  
{  
    return a+b+b*c+4;  
}
```

B

Values are copied at

B



存储器
状态

B. 调用方法时首先给形式参数分配空间，然后把实际参数的值赋给对应的形式参数。

传值传递模式- 3

代码

```
m=20;  
x = 10;  
y = 15;  
z=Abc (m, x, y);
```

```
int Abc(int a, int b, int  
c)  
{  
    return a+b+b*c+4;  
}
```

执行 **C** 后

m 20

a 20

x 10

b 10

y 15

c 15

z ?

存储器
状态

C. 在方法体中，使用形式参数运算。



传值传递模式- 4

代码

```
m=20;  
x = 10;  
y = 15;  
z=Abc (m, x, y);
```

D

```
int Abc(int a, int b, int c)  
{  
    return a+b+b*c+4;  
}
```

D 执行方法后

m 20 z 184

x 10

y 15

存储器
状态

D. 方法体执行完，形式参数失去了意义，在方法体中没有改变实际参数的值，计算值被返回。.

引用参数

```
template<class T>
T abc(T a, T b, T c)
{
    return a+b*c;
}
```

T : matrix(10*100)

abc 被调用时: 3000 次操作 (copy constructors)

abc 返回时: 3000 次操作 (destructors)

引用参数

程序1-4

```
template<class T>
T abc(T& a, T& b, T& c)
{
return a+b*c;
}
```

abc(x,y,z):

abc 被调用时 :实际参数 (*x*、*y* 和*z*) 将被分别赋予名称*a*、*b* 和*c*

abc(x,y,z): 在函数*abc* 执行期间, *x*、*y* 和*z* 被用来替换对应的*a*、*b* 和*c*。

与传值参数的情况不同, 在函数被调用时, 本程序并没有复制实际参数的值, 在函数返回时也没有调用析构函数。

传引用传递模式- 2

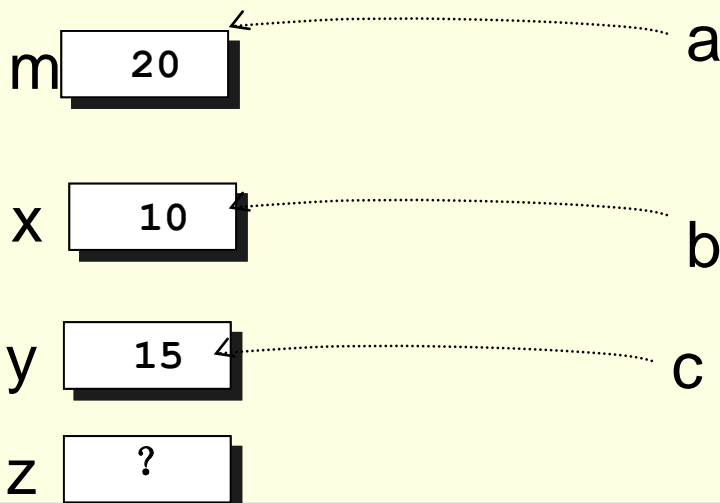
代码

```
m=3;  
x = 10;  
y = 20;  
z=Abc (m, x, y) ;
```

```
int Abc(int a, int b, int d)  
{  
    return a+b+b*c+(a+b-  
    c) / (a+b)+4;  
}
```

B

Values are copied at **B**



存储器
状态

B.调用参数的值，这里是一个地址，被复制给形式参数。

传引用传递模式- 4

代码

```
m=3;  
x = 10;  
y = 20;  
z=Abc (m, x, y);
```

D

```
int Abc(int a, int b, int  
c)  
{  
return a+b+b*c+(a+b-  
c)/(a+b)+4;  
}
```

D 执行方法后

m 20 z 184

x 10

y 15

存储器
状态

D. 方法体执行完，形式参数失去了意义，在方法体中没有改变实际参数的值，计算值被返回。

常量引用参数

程序1-6

```
template<class Ta, class Tb , class Tc >
Ta abc(const Ta& a, const Tb& b, const Tc& c)
{
    return a+b*c;
}
```

函数不得修改引用参数。

返回值

- 值返回

被返回的对象均被复制到调用（或返回）环境中。

- 引用返回

```
T& X(int i, T& z)
{.....
Return z;
}
```

Z 是对一个实际参数的引用，不会把**z**的值复制到返回环境中。

- 常量引用返回

```
Const T& X(int i,T& z)
```

返回的结果是一个不变化的对象。

编程建议：

对于诸如 `int`、`float` 和 `char` 的简单数据类型，
当函数不会修改实际参数值的时候我们可以采用
传值参数；

对于所有其他的数据类型（包括模板类型），当函数
不会修改实际参数值的时候可以采用常量引用
参数。

重载函数

- 函数重载：定义多个同名函数的机制。
- C++允许定义多个同名函数，但同名函数不能有同样的签名。
- 函数签名：
 - 函数的形参类型和形参个数确定。

程序1-1 函数abc的签名
(int,int,int) ;

程序1-2 函数abc的签名
(float, float, float) ;

程序1-1

```
int abc(int a, int b, int c)
{
    return a+b*c;
}
```

程序1-2

```
float abc(float a, float b, float c)
{
    return a+b*c;
}
```

异常

- 抛出异常
- 异常：程序出现错误的信息。

程序1-8 抛出一个类型为char*的异常

```
int abc(int a, int b, int c)
{
    if(a<=0 || b<=0 || c<=0 )
        throw "All parameters should be >0" ;
    return a+b*c;
}
```

异常

- 处理异常

- try-catch结构:

程序1-9 捕获一个类型为char*的异常

```
int main()
{
    try (cout<<abc(2,0,4)<<endl; )
        catch (char* e)
        {cout<< “The parameters to abc were 2,0,4 ”<<endl;
          cout<< “An exception has been thrown”<<endl;
          cout<< e )<<endl;
        return 1;
        }
    return 0;
}
```

动态存储空间分配

- C++内存分配
- C/C++定义了4个内存区间：
 - 1. 代码区
 - 2. 全局变量与静态变量区
 - 3. 局部变量区(栈区)
 - 4. 动态存储区，即堆(heap)区或自由存储区(free store)

C++内存分配

- 通常定义变量(或对象)，编译器在**编译时**都可以根据该变量(或对象)的类型知道所需内存空间的大小，从而系统在适当的时候为它们分配确定的存储空间。这种内存分配称为**静态存储分配**；
- 有些操作对象只在程序运行时才能确定，这样编译时就无法为它们预定存储空间，只能在**程序运行时**，系统根据运行时的要求进行内存分配，这种方法称为**动态存储分配**。
- 所有动态存储分配都在堆区(动态存储区)中进行。

C++内存分配

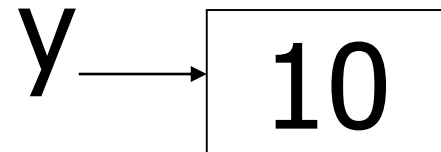
- 当程序运行到需要一个动态分配的变量或对象时，必须向系统**申请**取得堆中的一块所需大小的存贮空间，用于存贮该变量或对象。
- 当不再使用该变量或对象时，也就是它的生命结束时，要**显式释放**它所占用的存贮空间，这样系统就能对该堆空间进行再次分配，做到重复使用有限的资源。

操作符 new

- **new**—C++操作符new可用来进行动态存储分配，该操作符返回一个指向所分配空间的**指针**。

```
int *y ;  
y=new int;  
*y = 10;
```

```
int *y = new int;  
*y = 10;
```



```
int *y = new int (10);
```

```
int *y;  
y = new int (10);
```

一维数组

如果数组的大小在编译时是未知的，必须进行动态存储分配。

```
float *x=new float[n];
```



可以使用 **x[0], x[1], ..., x[n-1]** 来访问每个数组元素。

异常处理

```
float *x=new float[n]
```

如果计算机不能分配足够的空间怎么办？在这种情况下， **new** 不能够分配所需数量的存储空间，将会引发一个异常。

```
float *x;
```

```
try {x = new float [n];}
```

```
catch (bad_alloc) { //仅当new 失败时才会进入  
    cerr << "Out of Memory" << endl;  
    exit (1) ; }
```

操作符 delete

- Delete:

释放由操作符new所分配的空间

```
int *y = new int (10);  
Delete y;
```

```
float *x=new float[n];  
Delete []x;
```

二维数组

- 二维数组每一维的大小在编译时都是已知时：

```
char c[7][5];
```

- 在编译时数组的行数未知：

```
char (*c)[5];
```

```
try { c = new char [n][5]; }
```

```
catch (bad_alloc) { //仅当new 失败时才会进入
```

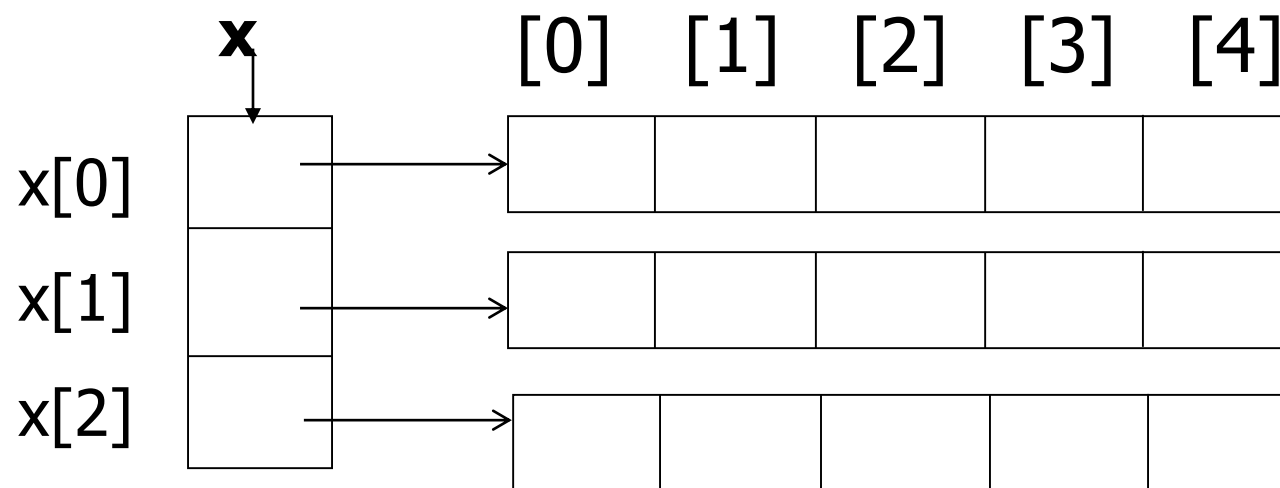
```
    cerr << "Out of Memory" << endl;
```

```
    exit (1); }
```

二维数组

- 在编译时数组的列数未知：

`char **x;`



二维数组

为一个二维数组分配存储空间（程序 1-10）

```
template <class T>
bool make2dArray (T** &x, int numberOfRows, int numberOfcolumns)
{ // 创建一个二维数组
    try {
        // 创建行指针
        x = new T* [numberOfRows];

        // 为每一行分配空间
        for (int i = 0 ; i < numberOfRows; i++)
            x[i] = new T [numberOfcolumns];
        return true;
    }
    catch (bad_alloc) { return false; }
}
```

二维数组

程序 1-11 创建一个二维数组但不处理异常.

```
template <class T>
Void make2dArray ( T** &x, int numberOfRows, int
numberOfcolumns)
{//创建一个二维数组

    //创建行指针
    x = new T*[numberOfRows];

    //为每一行分配空间 .
    for (int i = 0 ; i < numberOfRows; i++)
        x[i] = new T [numberOfcolumns];
}
```


二维数组

程序 1-11——不捕获异常

- 简化了函数的代码设计
- 可以使异常捕获落在一个更适合报告错误或或修改错误的地方.

```
try { make2dArray (x, r, c);}  
catch (bad_alloc) {cerr<< "Could not create x" << endl;  
                  exit (1) ; }
```

二维数组

程序 1-12 释放程序1 - 10中为二维数组所分配的空间

```
template <class T>
void delete2dArray( T** &x, int numberOfRows)
{ // 删除二维数组x

    // 释放为每一行所分配的空间
    for (int i = 0 ; i < numberOfRows ; i++)
        delete [ ] x[i];

    // 删除行指针
    delete [] x;
    x = NULL;
}
```

自有数据类型

■ 类**currency**

类型**currency**的对象(实例):

\$2.35 -\$6.05 -\$10.00

成员: 符号 (+或-), 美元, 美分

对**currency**类型对象我们想要执行的操作:

- 设置成员值
- 确定各成员值
(即符号, 美元数目和美分数目)
- 两个对象相加
- 增加成员的值
- 输出

```
enum signType {plus, minus};
```

currency类声明

```
class currency {
public:
    //构造函数
    currency(signType theSign = plus, unsigned long theDollars = 0, unsigned int
theCents = 0);
    //析构函数
    ~currency() {}
    bool setValue(signType, unsigned long, unsigned int);
    bool setValue(double);
    signType getSign() const {return sign;}
    unsigned long getDollars() const {return dollars;}
    unsigned int getCents() const {return cents;}
    currency add(const currency&) const;
    currency& increment(const currency&);
    void output() const;
private:
    signType sign;           //对象的符号
    unsigned long dollars;    //美元的数量
    unsigned int cents;       //美分的数量
```

创建对象、同名函数的调用

- 创建currency类对象两种方式:
 - ✓ `currency f,g(plus,3,45),h(minus,10);`
`f= $0.00, g=$3.45, h= -$10.00`
 - ✓ `currency *m=new currency (plus,8,12)`

setValue:设置成员的值

`g.setValue(minus,33,0); //g=-$33.00`

`h.setValue(20.52); //h=$20.52`

常量函数、复制构造函数

- 常量函数: 函数不会修改调用对象的值

```
signType getSign() const {return sign;}
```

```
unsigned long getDollars() const {return dollars;}
```

```
unsigned int getCents() const {return cents;}
```

- 复制构造函数:

- 被用来执行返回值的复制及传值参数的复制。
- C++中缺省的复制构造函数: 仅进行数据成员的复制。

类**currency**的构造函数

程序1-14

```
currency::currency(signType theSign, unsigned long theDollars,  
unsigned int theCents)
```

```
{//创建一个currency类对象 .  
    if (theCents > 99)        //美分太多  
        throw illegalParameterValue("Cents should be < 100" );  
  
    sign = theSign;  
    dollars = theDollars;  
    cents = theCents;  
}
```

函数setValue

程序1-15

```
bool currency::setValue((signType theSign, unsigned long
theDollars, unsigned int theCents))
{ //给调用对象的数据成员赋值
    if (theCents > 99)        //美分太多
        throw illegalParameterValue("Cents should be < 100" );

    sign = theSign;
    dollars = theDollars;
    cents = theCents;
    return true;
}
```


函数setValue

程序1-15

```
bool currency::setValue(double theAmount)
{ //给调用对象的数据成员赋值
    if (theAmount < 0) { sign = minus; theAmount = -theAmount; }
    else sign = plus;
        dollars = (unsigned long) theAmount;    //提取整数部分
        cents = (unsigned int) ((theAmount+0.001-dollars)*100); //提
取两个小数位
    return true;
}
```

函数add

程序1-16 (1)

```
currency currency::add(const currency& x) const
{ //把x和 *this相加.
    long a1, a2, a3;
    currency result;

    //把调用对象转换成符号整数
    a1 = dollars * 100 + cents;
    if (sign == minus) a1 = -a1;

    //把x转换成符号整数
    a2 = x.dollars * 100 + x.cents;
    if (x.sign == minus) a2 = -a2 ;

    a3 = a1 + a2;
```

函数add

程序1-16 (2)

//转换为currency 对象的表达形式

```
if (a3 < 0) {result.sign = minus; a3 = -a3 ; }
```

```
else result.sign = plus;
```

```
result.dollars = a3/100;
```

```
result.cents = a3 - result.dollars * 100;
```

```
return result;
```

```
}
```

函数increment和output

程序1-17

```
currency& currency::increment(const currency& x)
{
    //增加量 x .
    *this = add(x);
    return *this;
}
```

```
void currency::output ( ) const
{
    //输出调用对象的值
    if (sign == minus) cout << '-';
    cout << '$' << dollars << '.';
    if (cents < 10) cout << "0";
    cout << cents;
}
```

类**currency**的应用

- **currency**类声明和类实现都在文件**currency.h**中
程序1-18(1)

```
#include <iostream>
#include "currency.h "
using namespace std;
int main ()
{
    currency g, h(plus, 3, 50), i, j;

    //使用两种形式的setValue来赋值
    g.setValue(minus, 2, 25);
    i.setValue ( - 6. 45 );
```

类**currency**的应用

程序1-18(2)

//调用add和output

```
j = h.add(g);  
h.output( );  
cout << " + ";  
g.output( );  
cout << " = ";  
j.output( ); cout << endl;
```

//连续调用两次函数add

```
j = i.add(g).add(h);  
.....//省略输出语句
```

//调用函数increment和add

```
j = i.increment(g).add(h);  
.....//省略输出语句
```

类currency的应用

程序1-18(3)

//测试异常

```
cout<< " Attempting to initalize with cents =152" << endl;
try {i.setValue(plus, 3, 152); }
catch (illegalParameterValue e)
{
    cout<< "Caught thrown exception" << endl;
    e.outputMessage();
}
return 0;
}
```

一种不同的描述方法

- 已经有许多应用程序采用currency类;
- 修改currency类, 使其应用频率最高的两个函数add和increment可以运行得更快
- 不影响应用代码的正确性


```

class currency
{
public:
    //构造函数
    currency(signType theSign = plus, unsigned long theDollars = 0, unsigned int
theCents = 0);
    //析构函数
    ~currency() {}
    bool setValue(signType, unsigned long, unsigned int);
    bool setValue(double);
    signType getSign() const
    {if (amount < 0) return minus;
     else return plus;}
    unsigned long getDollars() const
    {if (amount < 0) return (-amount) / 100;
     else return amount / 100;}
    unsigned int getCents() const
    {if (amount < 0)
        return -amount - Dollars() * 100;
     else return amount - Dollars() * 100;;}
    currency add(const currency&) const;
    currency& increment(const currency&);
    {amount += x.amount; return *this;}

    void output() const;
private:
    long amount;
}

```

操作符重载

程序1-22 包含操作符重载的类声明

```
class currency {  
public :  
    //构造函数  
    currency(signType theSign = plus, unsigned long theDollars = 0,  
        unsigned int theCents = 0);  
    :  
    :  
    currency operator+(const currency& ) const;  
    currency& operator+=(const currency& x)  
        { amount += x.amount; return *this; }  
    void output(ostream& out) const;  
Private:  
    long amount;  
} ;
```

程序1-23

currency currency::operator+(const currency& x) const

```
{//把参数对象x 和调用对象 *this相加.  
    currency result;  
    result.amount = amount + x.amount;  
    return result;  
}  
void currency::output(ostream& out) const  
{//将货币值插入到输出流 .  
    long theAmount = amount;  
    if (theAmount < 0) {out << '-' ; theAmount = -theAmount ; }  
    long dollars = theAmount / 100;    // dollars  
    out << '$' << dollars << '.' ;  
    int cents = theAmount - dollars * 100;    // cents  
    if (cents < 10) out << "0";  
    out << cents;  
}  
//重载 <<  
ostream& operator<<(ostream& out, const currency& x)  
{x.output(out); return out;}
```

类currency的应用

- 使用操作符重载，程序1-18（类currency的应用）：

.....

//调用add和output

j = h.add(g);

h.output();

cout << " + ";

g.output();

cout << " = ";

j.output(); cout << endl;



//调用add和output

j = h+g;

cout << h << " + " << g << " = " << j << endl;

//连续调用两次函数add

j = i.add(g).add(h);

.....//省略输出语句



//连续调用两次函数add

j = i+g+h;

cout << i << " + " << g << " + " << h << " = "
<< j << endl;

//调用函数increment和add

j = i.increment(g).add(h);

.....//省略输出语句



//调用函数increment和add

j = (i+=g)+h;

.....

友元和保护性类成员

- 把ostream& operator<<描述为currency类的友元
- ```
class currency {
 friend ostream& operator <<(ostream&, const currency&);
Public:
```

.....

## 程序1-25

```
//重载 <<
ostream& operator<<(ostream& out, const currency& x)
{//把货币值插入到输出流
 long theAmount = x.amount;
 if (theAmount < 0) {out << '-'; theAmount = -theAmount;}
 long dollars = theAmount / 100; //dollars
 out << '$' << dollars << '.';
 int cents = theAmount - dollars * 100; // cents
 if (cents < 10) out << "0";
 out << cents;
 return out;
}
```

# 保护性类成员

```
class currency
{
friend ostream& operator <<(ostream&, const currency&);
Public:
 ...
Protected:
 ...
Private:
 ...
}
```

- 派生类可以访问保护性类成员。

## 增加 `#ifndef`, `#define`, and `#endif` 语句

- 在文件头，必须放上语句：

```
#ifndef currency_
#define currency_
```

- 在文件尾需要放上语句：

```
#endif
```

- 确保 `currency` 的代码仅被程序包含和编译一次

# 异常类illegalParameterValue

```
class illegalParameterValue
{
public:
 illegalParameterValue():message("Illegal Parameter Value"){ }
 illegalParameterValue(char* theMessage)
 { message= theMessage; }
 void outputmessage() { cout<<message<<endl; }
private:
 string message;
};
```



# 递归函数（Recursive Functions）

- 递归函数：递归函数是一个自己调用自己的函数。
- 直接递归：  $f \longrightarrow f$   
函数 $f$ 的代码中直接包含了调用 $f$ 的语句。
- 间接递归：  $f \longrightarrow g \longrightarrow h \longrightarrow f$   
函数 $f$ 调用了函数 $g$ ， $g$ 又调用了 $h$ ，如此进行下去，直到 $f$ 又被调用。

# 数学函数的递归定义

阶乘函数  $f(n)=n!$

$$f(n) = \begin{cases} 1 & n \leq 1 \\ nf(n-1) & n > 1 \end{cases}$$

➤完整的递归定义，必须包含：

1. 基本部分（一般包含了  $n \leq k$  的情况）

$$f(n)=1 \quad n \leq 1$$

$f(n)$  是直接定义的（即非递归）

2. 递归部分：（右侧所出现的所有  $f$  的参数都必须有一个比  $n$  小）

$$f(n)=nf(n-1) \quad n > 1$$

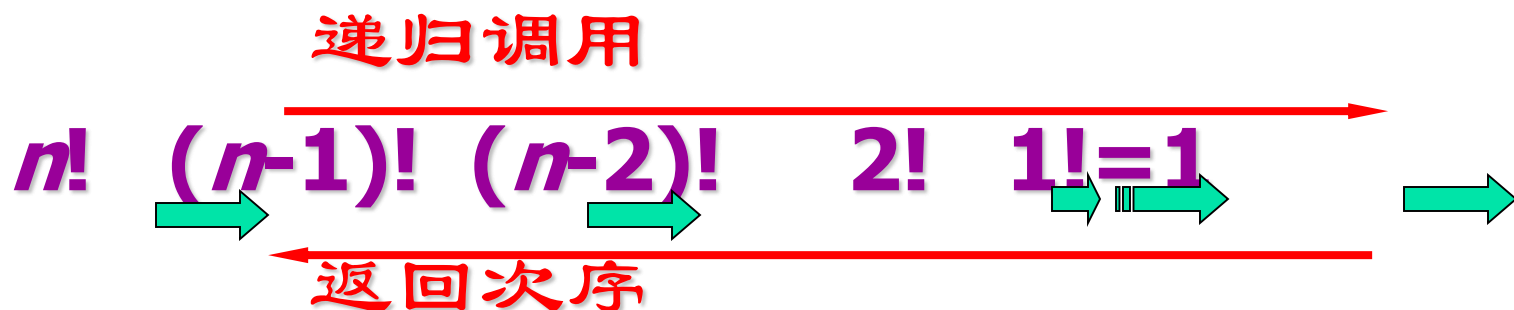
➤重复运用递归部分来改变右侧出现的  $f$ ，直至出现  $f$  的基本部分。

# 数学函数的递归定义

- 例如：
- $f(5)=5f(4)=20f(3)=60f(2)=120f(1)$
- 每次应用递归部分的结果是更趋近于基本部分，最后，根据基本部分的定义可以得到  $f(5)=120$ 。

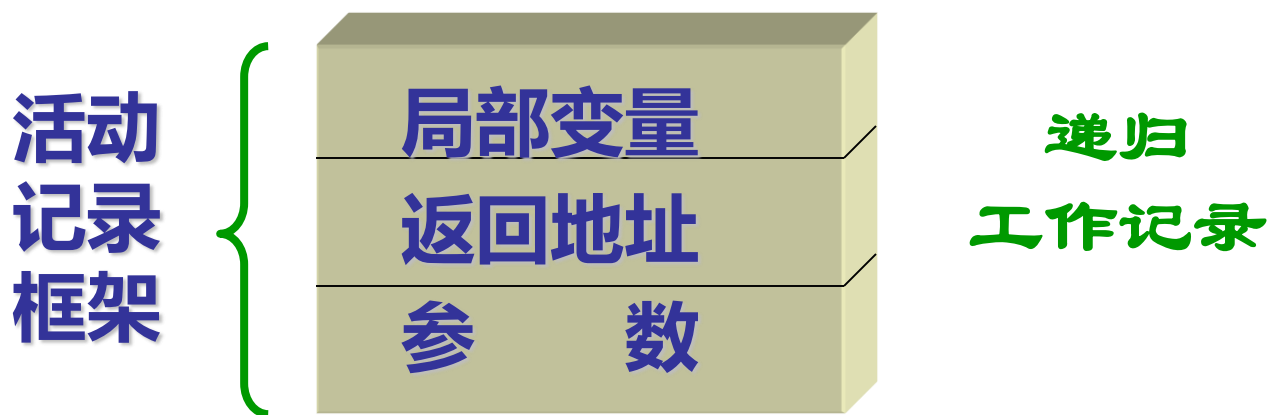
# 递归过程

递归过程在实现时，需要自己调用自己。  
层层向下递归，退出时的次序正好相反：



# 递归工作栈

每一次递归调用，需要分配存储空间，来保留程序的状态（如**局部变量**、**参数值**、**代码的执行位置**等）。每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。



# C++递归函数

■ 一个正确的递归函数必须包含：

➤ 一个基本部分

➤ 递归调用部分

所使用的参数值应比函数的参数值要小

例 1-1 (程序 1-29)

```
int factorial(int n)
{ //计算 n!
 if (n<=1) return 1;
 else return n*factorial(n-1);
}
```

# 求n个元素之和的非递归实现

## 例1-2 模板函数Sum

计算元素a[0]至a[n-1] 的和

程序 1-30

```
template<class T>
T Sum(T a[], int n)
{ //返回数组元素a[0]至a[n-1] 的和
 T thesum=0;
 for (int i = 0; i < n; i++)
 thesum += a[i];
 return thesum;
}
```

# 求n个元素之和的递归实现

例1-2 模板函数Sum

计算元素a [0]至a[n-1] 的和

当n=0时，和为0；当n>0时，n个元素的和是前面n-1个元素的和加上最后一个元素。

程序 1-31

```
template<class T>
T rSum(T a[], int n)
{
 if (n > 0)
 return rSum(a, n-1) + a[n-1];
 return 0;
}
```



# 例1-3 排列

## 例1-3 排列

$n$ 个不同元素的所有排列方式。

- 例：a，b和c的排列方式有：

abc,acb,bac,bca,cab,cba

## 例1-3 排列

$$E = \{e_1, e_2, \dots, e_{n-1}, e_n\}$$

$$E_i = \{e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_{n-1}, e_n\}$$

$\text{perm}(X)$  = 集合  $X$  中元素的排列方式。

$e_i \bullet \text{perm}(X)$  = 在 *perm* ( $X$ ) 中的每个排列方式的前面均加上  $e_i$  以后所得到的排列方式。

## 例1-3 排列

$$E = \{a, b, c\}$$

$$E_1 = \{b, c\}, \quad E_2 = \{a, c\}, \quad E_3 = \{a, b\}$$

$$\text{perm}(E_1) = (bc, cb)$$

$$\text{perm}(E_2) = (ac, ca)$$

$$\text{perm}(E_3) = (ab, ba)$$

$$a \cdot \text{perm}(E_1) = (abc, acb)$$

$$b \cdot \text{perm}(E_2) = (bac, bca)$$

$$c \cdot \text{perm}(E_3) = (cab, cba)$$

$$\text{perm}(E) = a \cdot \text{perm}(E_1) + b \cdot \text{perm}(E_2) + c \cdot \text{perm}(E_3)$$

## 例1-3 排列

$$\text{perm}(E) = \begin{cases} (e) & n=1 \\ e_1 \cdot \text{perm}(E_1) + e_2 \cdot \text{perm}(E_2) + \dots + e_n \cdot \text{perm}(E_n) & n>1 \end{cases}$$

基本部分，采用 $n=1$ 。当只有一个元素时，只可能产生一种排列方式，所以 $\text{perm}(E)=(e)$ ，其中 $e$ 是 $E$ 中的唯一元素。

递归部分：当 $n>1$ 时，采用 $n$ 个 $\text{perm}(X)$ 来定义 $\text{perm}(E)$ ，其中每个 $X$ 包含 $n-1$ 个元素。

## 程序 1-32

```
template<class T>
```

```
void permutations(T list[], int k, int m)
```

```
{ //生成list[k:m]的所有排列方式，输出前缀是 list[0:k-1] 后缀是
list[k:m]的所有排列方式
```

```
 int i;
```

```
 if (k == m) { // list[k:m]只有一个排列
```

```
 copy(list, list+m+1, ostream_iterator<T>(cout, " ")) ;
```

```
 cout << endl;
```

```
 }
```

```
 else // list[k:m]有多个排列方式，递归地产生这些排列方式
```

```
 for (i=k; i <= m; i++) {
```

```
 swap (list[k], list[i]);
```

```
 permutations (list, k+1, m);
```

```
 swap (list[k], list[i]);
```

```
 }
```

```
 }
```

# 递归函数优缺点

- 递归优点：
  - 递归简洁、易编写、易懂
  - 易证明正确性
- 递归缺点：
  - 递归效率低
  - 重复计算多

# 递归函数

- 递归函数改为非递归
- 改为非递归的目的是提高效率
- 单向递归可直接用迭代实现非递归
- 其他情形必须借助栈实现非递归

# 递归函数

阶乘非递归实现

计算 $n!$  的非递归函数

```
int Factorial(int n)
{ //计算 $n!$
 if (n<=1) return 1;
 int f=2;
 for (int i=3; i<=n; i++)
 f*=i;
 return f;
}
```



# 递归函数

## 斐波那契数列非递归实现

```
long Fib (long n) {
 if (n <= 1) return n;
 long twoback = 0, oneback = 1, Current;
 for (int i = 2; i <= n; i++) {
 Current = twoback + oneback;
 twoback = oneback; oneback =
Current;
 }
 return Current;}

```

# 递归函数

程序1-31 sum 非递归实现

```
template<class T>
T Sum(T a[], int n)
{//计算a[0:n-1]的和
 T tsum = 0;
 for (int i=0; i<n; i++)
 tsum += a[i];
 return tsum;
}
```

当 $n=0$ 时，和为0；当 $n>0$ 时， $n$ 个元素的和是前面 $n-1$ 个元素的和加上最后一个元素。

# 作业

- a. 体会，学习计划。
- b. 试编写一个递归函数，用来输出 $n$ 个元素的所有子集。例如，三个元素  $\{a, b, c\}$  的所有子集是： $\{\}$ （空集）， $\{a\}$ ， $\{b\}$ ， $\{c\}$ ， $\{a, b\}$ ， $\{a, c\}$ ， $\{b, c\}$  和  $\{a, b, c\}$ 。

# C++标准模板库(STL)

- C++标准模板库(STL)是一个容器、适配器、迭代器、函数对象和算法的集合，使用STL，应用程序的设计会简单很多。
- `copy(start,end,to)`
- 对范围在`[start,end)`内的元素复制到`to`开始的另一个位置。
- 即：元素从位置`start, start+1,...,end-1`依次复制到`to, to+1,...,to+end-start`
- 例：`copy(list,list+m+1,ostream_iterator<T>(cout, " ")) ;`
- 等价于：`for (i = 0; i <= m; i++) cout << list [i];`

# 测试与调试

- 正确性是一个程序的重要属性.
- 采用严格的数学证明方法来证明一个程序的正确性是非常困难的。
- 程序测试
  - 是指在目标计算机上利用输入数据，也称之为测试数据（**test data**）来实际运行该程序，把程序的实际行为与所期望的行为进行比较。如果两种行为不同，就可判定程序中有问题存在。
  - 即使两种行为相同，也不能够断定程序就是正确的，因为对于其他的测试数据，两种行为又可能不一样。

# 程序测试

- 通过使用所用可能的测试数据，可以验证一个程序是否正确。
- 不可能进行穷尽测试；
- 实际用来测试的输入数据空间的子集称之为测试集（**test set**）。
- 测试的目的
  - 不是去建立正确性认证，而是为了发现尽可能多的缺陷(功能错误/性能低下/易用性差)
  - 一个成功的测试示例在于发现了至今尚未发现的缺陷。

# 程序测试

- 测试数据选择条件：
  - 这个数据能够发现错误的潜力如何？
  - 能否验证采用这个数据时程序的正确性？
- 设计测试数据的方法
  - 黑盒法
    - 考察程序的功能
  - 白盒法
    - 考察程序的结构