中国科学技术大学软件学院
SCHOOL OF SOFTWARE ENGINEERING OF USTC

第七章　模板与标准模板库

ffh

目录

7.1模板的基本知识

- 类型的严格性与灵活性
  - 在强类型程序设计语言中，参与运算的所有对象的类型在编译时即可确定下来，并且编译程序将进行严格的类型检查。
  - 强类型语言提高了程序的可靠性，但也带来了一些负作用，如，假定x、y类型相同，现求x、y中的较大值。 可有以下两个函数：

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

和

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

强类型的程序设计迫使程序员为逻辑结构相同而具体数据类型不同的对象编写模式—致的代码，而无法抽取其中的共性，不利于程序的扩充和维护。

7.1模板的基本知识

再如：（书 p.257）下面程序创建了一个intArray 类，可以仿照 intArray 创建charArray、stringArray类等。它们的代码一致，差别只是类型不同。

7.1模板的基本知识

再如：（书 p.257）

```
class intArray
{ public:
    int& operator [ ] ( int );
    const int& operator[ ] ( int ) const;
    intArray( int s )  {  a = new int[ size = s ];  }
    ~intArray()  { delete [] a;  }
    int get_size() const { return size; }
  private:
    int* a;
    int size;
    intArray();              //*** for emphasis
    int dummy_val;     //arbitrary value
};
```

7.1模板的基本知识

```
int& intArray::operator [] ( int i )
{  if (i < 0 || i >= size)
        {  cerr << "index " << i << "Out of bounds: ";
                return dummy_val;        }
    return a[i];
}
const int& intArray::operator[] ( int i ) const
{   if (i < 0 || i >= size)
        {  cerr << "index " << i << "Out of bounds: ";
                return dummy_val;        }
    return a[i];
}
```

补充：函数重载之const

- 类中，同名的const函数和非const函数能够重载，它们被调用的时机为：如果定义的对象是常对象，则调用的是const成员函数，如果定义的对象是非常对象，则调用重载的非const成员函数。如下例：

补充：函数重载之const

```
class Test  {
public:    void test1() {cout<<"non const"<<endl;}
           void test1() const {cout<<"const"<<endl;}
           void test2() const {cout<<"const 2"<<endl;}
           void test3() {cout<<"non const 3"<<endl;}
   };
int main() {
   Test  t1;   t1.test1();   t1.test2();  t1.test3();
   const Test  t2;     t2.test1();  t2.test2();
   t2.test3(); //错:不能将this指针从“const Test”转换为“Test &”
   return 0;
}
```

运行结果：

```
non const
const 2
non const 3
const
const 2
```

7.1模板的基本知识

- 解决冲突的途径

  解决类型的严格性与灵活性冲突，以前有3种方法：

  – 用宏函数
  – 为各种类型都重载这一函数
  – 放松类型检查

  最理想的方法是：直接将数据类型作为类的参数，就好像函数可以将数据作为参数一样，这种机制称为类属（模板）。

  如，较大值若使用模板，则只定义一个函数：

```
Template<class  T>
T max( T   a,T   b)  { return(a>b)?a,b; }
```

  再如：（p.258）

7.1模板的基本知识

```
template< class T >
class Array
{ public:
    T& operator [] ( int );
    const T& operator[] ( int ) const;
    Array( int );
    ~Array();
    int get_size() const { return size; }
  private:
    T* a;        int size;
    Array();                  //*** for emphasis
    T dummy_val;      //arbitrary value
};
```

7.1模板的基本知识

```
template< class T >
T& Array< T >::operator [] ( int i )
{   if (i < 0 || i >= size)
        {       cerr << "index " << i << "Out of bounds: ";
                return dummy_val;       }
    return a[i];  }

template< class T >
const T& Array< T >::operator[] ( int i ) const
{   if (i < 0 || i >= size)
        {       cerr << "index " << i << "Out of bounds: ";
                return dummy_val;       }
    return a[i];    }
```

7.1模板的基本知识

```
template< class T >
Array< T >::Array(int s) {       a = new T[ size = s ]; }

template< class T >
Array< T >::~Array() {  delete [] a; }

template< class T >
ostream& operator<< ( ostream& os, const  Array<T>&  ar)
{   for ( int i = 0; i < ar.get_size(); i++ )
       os << ar[ i ] << endl;
    return os;
}
```

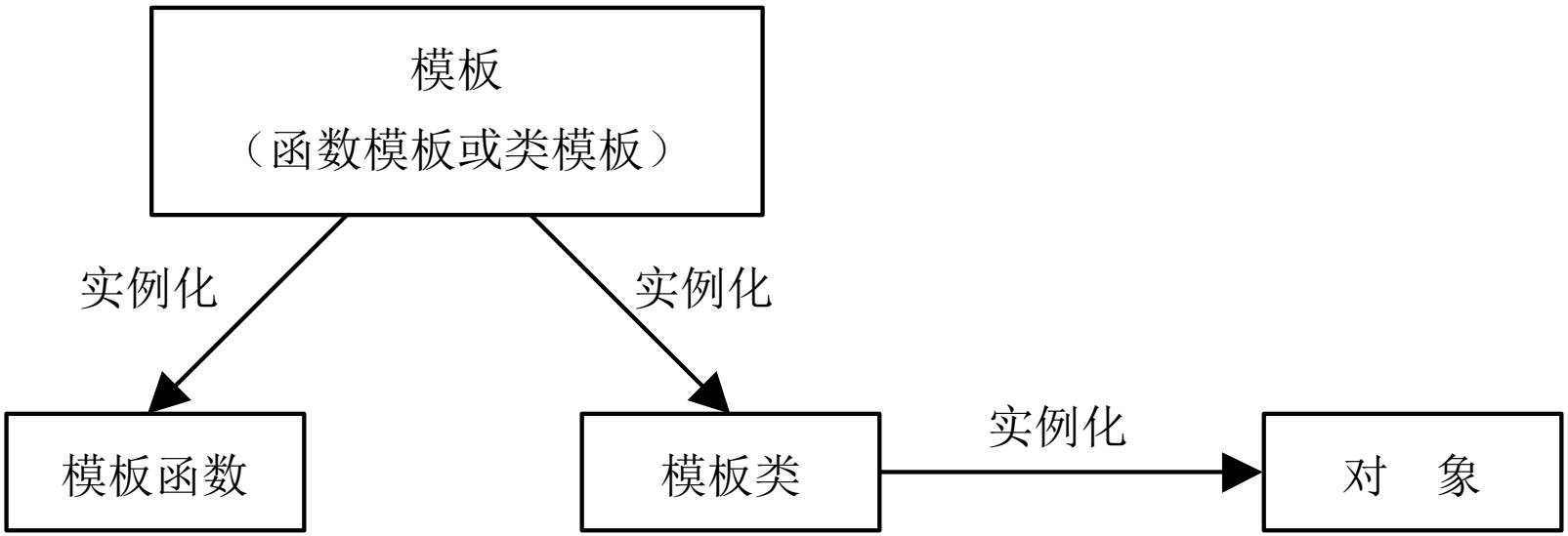
7.1模板的基本知识

```
int main()
{   Array< double >  a1(100);  //array of 100 doubles
    a1[ 6 ] = 3.14;
    cout<<a1[6]<<endl;
    //...
}
```

7.1模板的基本知识

- 模板的概念
  - 模板是一种参数化多态性的工具，可以为逻辑功能相同而类型不同的程序提供一种代码共享的机制。
  - C++中，模板分为函数模板和类模板。
  - 一个模板并非一个实实在在的函数或类，仅仅是一个函数或类的描述，这些模板运算对象的类型不是实际的数据类型，而是一种参数化的类型(又称为类属类型)。
  - 模板的类属参数由调用它的实际参数的具体数据类型替换，由编译器生成一段真正可以运行的代码。这个过程称为实例化。
  - 通过参数实例化可以再构造出具体的函数或类，称为模板函数和模板类，它们之间的关系如下图所示。

7.1模板的基本知识



7.*1 函数模板

函数模板是一种不指定某些参数的数据类型的函数，在函数模板被调用时根据实际参数的类型决定这些函数模板参数的类型。

7.*1.1 函数模板的定义

函数模板的定义格式如下：

```
template <模板参数表>
<返回值类型> <函数名>(<参数表>)
{   <函数体> }
```

其中:

- 关键字template是定义一个模板的关键字
- <模板参数表>中包含一个或多个用逗号分开的模板参数项，每一项由保留字class或typename开始，后跟一种数据类型。
- 函数模板中可利用模板参数定义函数返回值类型、参数类型和函数体中的变量类型。它同基本数据类型一样，可以在函数中的任何地方使用。

7.*1 函数模板

- 例：定义函数模板求两个数中的较大值。

```
template <class T>
T max(T a, T b)
{  return a>b? a:b;  }
```

或：

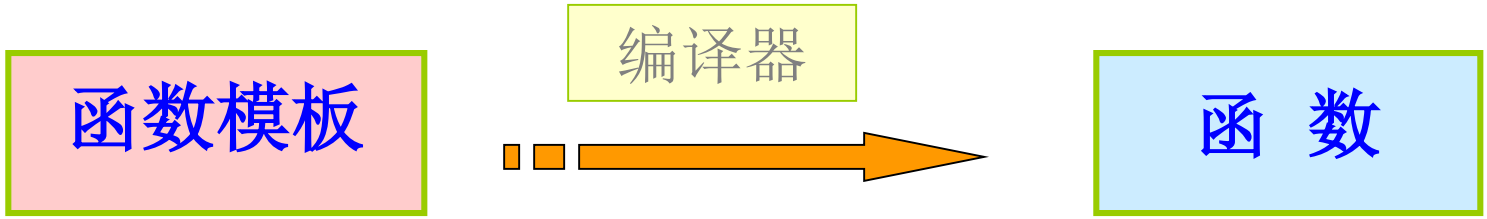
```
template  <class  T1，class  T2>
T1   max ( T1  x,  T2  y)
{   return  x>=y ? x : (T1)y; }
```

当程序中使用这个函数模板时，编译程序将根据函数调用时的实际数据类型产生相应的函数。如产生求两个整数中的较大值的函数，或求两个浮点数中的较大值函数等等。

7.*1 函数模板

7.*1.2 函数模板的实例化

- 函数模板是对一组函数的描述，它以类型作为参数及函数返回值类型。它不是一个实实在在的函数，编译时并不产生任何执行代码。
- 当编译系统在程序中发现有与函数模板中相匹配的函数调用时，便生成一个重载函数。该重载函数的函数体与函数模板的函数体相同，参数为具体的数据类型。我们称该重载函数为模板函数，它是函数模板的一个具体实例。



7.*1 函数模板

例：template <typename T> //函数模板

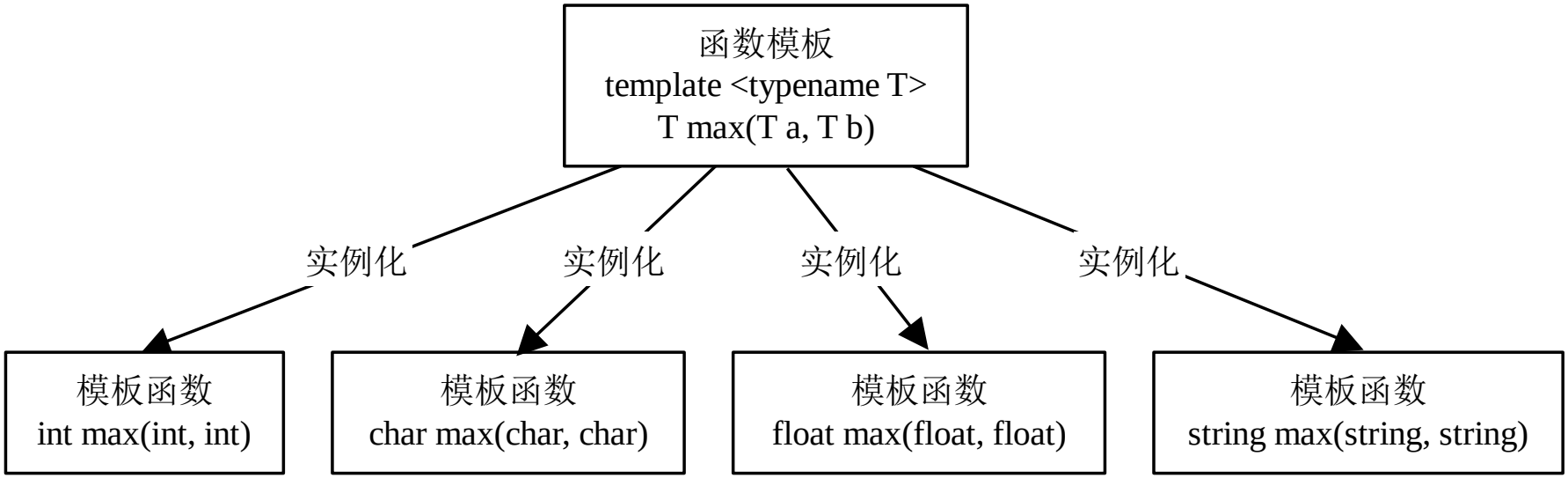
```
T max(T a, T b) { return a>b? a:b;}
int main( )
{   int a,b;
    cout << "Input two integers to a&b:"<<endl;
    cin >> a >> b;
    cout << "max(" << a << ","<< b << ") = "<< max(a,b) << endl;
    char c,d;
    cout << "Input two  chars to d&t: "<<endl;
    cin >> c >> d;
    cout << "max(" << "\'"<<c<< "\'" << "," << "\'"<< d << "\'"<< ") = ";
    cout<<max(c,d) << endl;
```

7.*1 函数模板

```
    float x,y;
    cout << "Input two floats to   x&y: "<<endl;
    cin >> x >> y;
    cout << "max(" << x << "," << y << ") = " << max(x,y) << endl;
    cout << "Input two strings to  p&h: "<<endl;
    string  p,h;
    cin >> p >> h;
    cout << "max("<<"\""<< p << "\""<< "," << "\""<<h << "\""<<") = " ;
    cout<<max(p,h) << endl;
    return 0;
}
```

7.*1 函数模板

函数模板和模板函数的关系



7.*1 函数模板

再如：一个对具有n个元素的数组a[ ]求最小值的程序。

```
template <typename  T>
T min(T a[ ],int n)
{   int i;
    T minv=a[0];
    for(i=1;i<n;i++)
      if(minv>a[i])  minv=a[i];
    return minv;
}
```

7.*1 函数模板

```
void main( )
{   int a[ ]={1,3,0,2,7,6,4,5,2};
    double b[ ]={1.2,-3.4,6.8,9.8};
    cout<<"a数组的最小值为："<<min(a,9)<< endl;
    cout<<"b数组的最小值为："<<min(b,4)<<endl;
}
```

类型参数T
替换为int

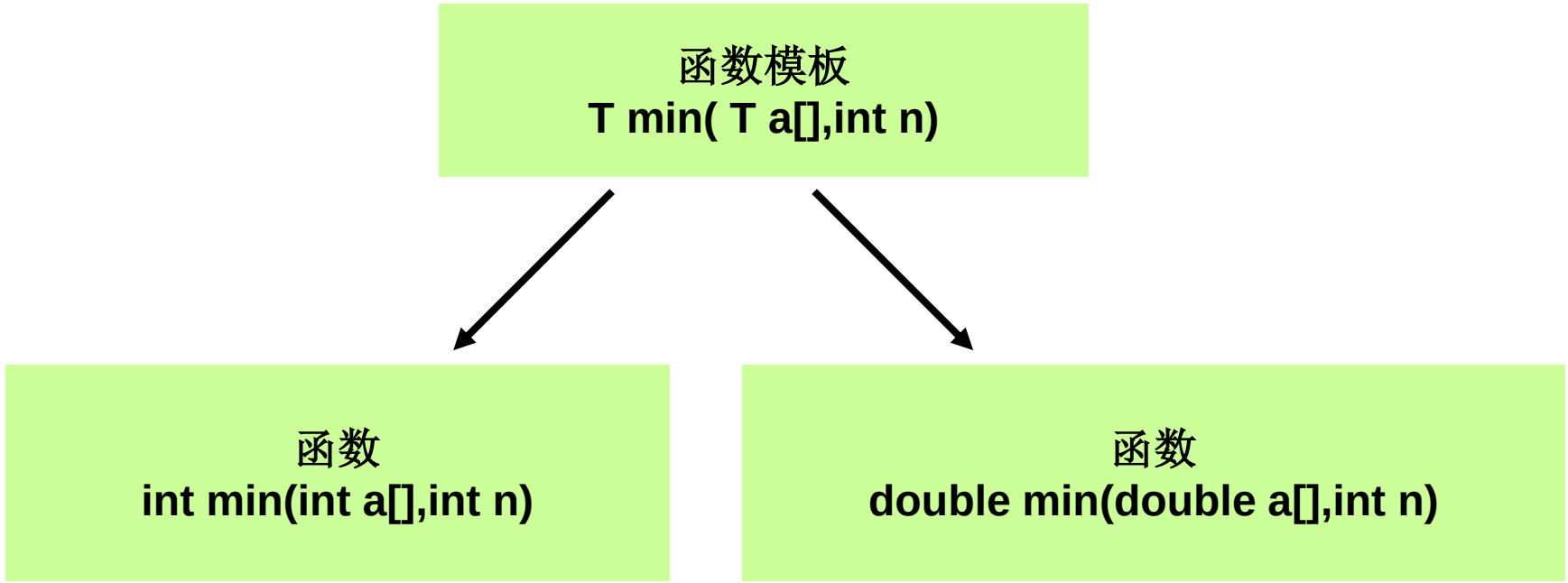
类型参数T
替换为double

运行结果

a数组的最小值为：0

b数组的最小值为：-3.4

7.*1 函数模板


函数模板
T min(T a[],int n)
函数
int min(int a[],int n)
函数
double min(double a[],int n)

7.*1 函数模板

- 注意：

虽然模板参数T可以实例化成各种类型，但是采用模板参数T的各参数之间必须保持完全一致的类型。

```
template <typename T>//函数模板
T max1(T a, T b)  { return a>b? a:b; }
…
int a=4,b=5;    float c=5;
cout << max1(a,b) << endl;        //ok
cout << max1(a,c) << endl;        //error
```

模板类型并不具有隐式的类型转换，例如在int与float之间、float与double之间等的隐式类型转换。而这种转换在C++中是非常普遍的。

7.*1 函数模板

- 7.*1.3 函数模板的重载

  函数模板可以用多种方式重载，可以定义同名的函数模板，提供不同的参数和实现；也可以用其它非模板函数重载。

- 函数模板的重载

  例：重载上例函数模板求数组数中的最大值。

7.*1 函数模板

```
template <typename T>//第1个函数模板
T max1(T a, T b)  {return a>b? a:b;}

template <typename T>//第2个函数模板
T max1(T a[], int n)
{   T temp;
    temp=a[0];
    for(int i=1;i<=n-1;i++)
      if (a[i]>temp)  temp= a[i];
    return temp;
}
```

7.*1 函数模板

```
int main( )
{   int a,b;
    cout << "Input two integers to a&b:"<<endl;
    cin >> a >> b;
    cout << "max(" << a << ","<< b << ") = "<< max1(a,b) << endl;
    int i,aa[10]={3,7,9,11,0,6,7,5,4,2};
    cout<<"The original array: "<<endl;
    for (i=0;i<10;i++)    cout<<aa[i]<<" “<<endl;
    cout<<"max of 10 integers is "<<max1(aa, 10)<<endl;
    return 0;
}
```

7.*1 函数模板

- 用普通函数重载函数模板 （例7.1普通函数重载函数模板.cpp）

```
#include <iostream>
using namespace std;
template <typename T>//函数模板
T max1(T a, T b)  { cout<<"call template:"; return a>b? a:b;}
int max1(int a,float b)  // 用普通函数重载函数模板
 { cout<<"call func:"; return a>b? a:b;}
int main( )
{   char a='4',b='5‘;    int c=5;
    cout << max1(a,b) << endl;
    cout << max1(a,c) << endl;
    return 0;
}
```

运行结果：
call template:5
call func:52

7.*1 函数模板

修改上例，增加比较字符数组的功能。

```
template <typename T>//函数模板
T max1(T a, T b)  {cout<<"call template:"; return a>b? a:b;}
char* max1(char* a,char* b)//重载函数，优先模板函数
{  cout<<"call func:"; return strcmp(a,b)>0? a:b;}
int main( )
{   char a='4',b='5‘;
    cout<< max1(a,b) << endl;  //模板函数
    char *p,*h;  p=“abc“; h=”ef“;
    cout<< max1(p,h) << endl; //重载函数
    return 0;
}
```

运行结果：

```
call template:5
call func:ef
```

7.*1 函数模板

- 函数模板方法克服克服了C++函数重载用相同函数名字重写几个函数的繁琐。因而，函数模板是C++中功能最强的特性之一，是提高软件代码重用率的重要手段。

7.*2 类模板

- 类是对问题的抽象，而类模板是对类的抽象，即更高层次上的抽象。
- 类模板称为带参数(或参数化)的类，也称为类工厂，它可用来生成多个功能相同而某些数据成员的类型不同或成员函数的参数及返回值的类型不同的类。

7.*2 类模板

7.*2.1 类模板定义

- 为了起到模板的作用，与函数模板一样，定义一个类模板时必须将某些数据类型作为类模板的类型参数。
- 模板类的实现代码与普通类没有本质上的区别，只是在定义其成员时要用到类模板的类型参数。
- 类模板的定义格式：

```
template <模板参数表>
class <类模板名>
{  <类成员声明>  }
```

其中，<模板参数表>中包含一个或多个用逗号分开的类型。

7.*2 类模板

- 例如，以下定义了含有一个类型参数的类模板：

```
template  < class  T >
class  MyTemClass
{  private:
     T  x;      // 类型参数T用于声明数据成员
   public:
     void  SetX( T  a ) { x=a; };
                // 类型参数T用于声明成员函数的参数
     T  GetX( ) { return  x; };
                // 类型参数T用于声明成员函数的返回值
};
```

7.*2 类模板

- 如果在模板类的外部定义模板类的成员函数，必须采用如下形式：

  template <模板参数表>

  <返回值类型><类模板名><类型名表>::<函数名>(<参数表>)

  { <函数体>

  }

  - 其中，<类模板名>即是类模板中定义的名称，<类型名表>即是类模板定义中的类型形式参数表中的参数名。如：

```
template   < class  T >     // 不能省略模板声明
void  MyTemClass <T> :: SetX( T   a )
{        x=a;  }
```

7.*2 类模板

- 模板可以拥有多个类参数，这些参数用逗号隔开。每个参数都必须有关键字class。含有多个参数类模板的定义：

```
template<class T1,class T2,class T3>
 class Sample{
 public:
   T2    m( T3  p ){          }
 private:
   T1   x;
};
```

7.*2 类模板

7.*2.2 类模板的实例化

与函数模板不同，类模板不是通过调用函数时实参的数据类型来确定类型参数具体所代表的类型，而是通过在使用模板类声明对象时所给出的实际数据类型确定类型参数。

类模板的实例化是指用某一数据类型替代类模板的类型参数，格式为：　　<类模板名> <类型实参表>

由类模板经实例化而生成的具体类称之为模板类。

声明对象的格式为：

<类模板名> <类型实参表>　<对象名>[（<实参表>）]

如，以下使用类模板声明了一个类型参数为int的模板类的对象：

MyTemClass < int >　intObject;

编译器首先用int替代类模板定义中的类型参数T，生成一个所有数据类型已确定的类；然后再利用这个类创建对象intObject。

7.*2 类模板

- 类模板不能直接定义对象，只能实例化后才能定义对象。

```
template <class T>
class Array{    };
Array   a0(50);   //error
Array< T >   a1(50);   //error
Array< int >   a3(50);  //ok
```

7.1.2 参数表中的模板类

- 模板类可以作为一种数据类型出现在参数表中。

```
template < class T>
ostream& operator<<(ostream & os, const  Array<T> &  ar)
{  for( int  i=0;i<ar.get_size( );i++)
      os<<ar[ i ]<<endl;
   return   os;
}
```

7.1.3 模板的函数式参数

- 类模板可以有非类参数的参数，称为函数类型参数。

在类模板的<模板参数表>中，必须至少有一个类参数。还可以有非类参数的参数，一般称之为函数类型参数，也称之为无类型模板参数。

```
template<class  T, int X,  float  Y>
class  C{
 };
```

通过函数类型参数可以做到：

```
C<double>  a1( 10 );
```

改为

```
C<double,10>  a1;
```

具体如下例：

7.1.3 模板的函数式参数

```
#include <iostream>
using namespace std;
template<class  T, int a>
class C {      T x;
       public:   C() { cout<<"1:";x=a; }
                 C(T xx) { cout<<"2:"<<a<<",";  x=xx; }
                 T getX()  { return x; }
 };
int main()
{      C<double,8>  a1( 10 );   cout<<a1.getX()<<endl;
       C<double,5>  a2;         cout<<a2.getX()<<endl;
       return 0;   }
```

运行结果：

```
2: 8,10
1: 5
```

补充1：

- 继承：

模板类可以与普通的类一样有基类，也同样可以有派生类。它的基类和派生类既可以是模板类，也可以不是模板类。所有与继承相关的特点，模板类也都具备。但仍有一些值得注意的地方。

设有如下类关系：

```
template<class T>  class A { ... };
A<int> aint;
A<double>  adouble;
```

则aint和adouble并非A的派生类，甚至可以说根本不存在A这个类，只有A<int>和A<doubl>这两个类。这两个类没有共同的基类，因此不能通过类A来实现多态。如果希望对这两个类实现多态，正确的类层次应该是：

补充1：

class Abase {…含虚函数};

template<class T> class A: public Abase {...};

A<int> aint;　　A<double> adouble;

也就是说，在模板类之上增加一个抽象基类，注意，这个抽象基类是一个普通类，而非模板。

– 有关类模板继承的几种情况

  • 普通类派生类模板

```
class Abase  { public:int x; };
template<class T>  class A: public Abase  { public: T y;};
int main()
{          A<int> aint;      A<double> adouble;
           aint.x=5;   adouble.y=10.5;
           cout<<aint.x<< adouble.y<<endl;  } //输出 5 10.5
```

补充1：

- 类模板派生类模板

这时，派生类模板的参数表中应包含基类模板的参数

```
template<class T1> class Abase  { public:T1 x; };
template<class T2,class T3> class A: public Abase<T2> //T2 传给T1
{ public: T3 y;};
int main()
{A<int,float> a1;    A<double,float> a2;
a1.x=5;  a2.x=10.5;
cout<<a1.x<<'\t'<<a2.x<<endl;  }  //输出 5   10.5
```

补充1：

- 类模板派生非模板类

```
template<class T1> class Abase  { public:T1 x; };
class A: public Abase<int>  { public: float y;};
int main()
{ A a1;
a1.x=5;  a1.y=10.5;
cout<<a1.x<<'\t'<<a1.y<<endl; } //输出 5   10.5
```

在定义A类时，Abase已实例化成了int型的模板类。

- 模板类与普通类一样也具有多继承，即模板类之间允许有多继承。

补充2：

- 静态成员

类的静态成员，同一个模板类共享

```
template<class T> class A {  static char a_;  };
A<int>  aint1, aint2;
A<double>  adouble1, adouble2;
```

这里模板A中增加了一个静态成员，注意：对于aint1和adouble1，它们并没有一个共同的静态成员。而aint1与aint2有一个共同的静态成员(对adouble1和adouble2也一样)。

这个问题实际上与继承里面讲到的问题是一回事，关键要认识到aint与adouble分别是两个不同类的实例，而不是一个类的两个实例。

课堂练习

- 下列对模板的声明,正确的是__C__。

  A)template<T>　　B)template<class T1,T2>

  C)template<class T1,class T2>

  D)template<C1ass T1,class T2>

- 一个__C__允许用户为类定义一种模式,使得类中的某些数据成员及某些成员函数的返回值能取任意类型。

  A)函数模板　　B)模板甲数　　C)类模板　　D)模板类

- 类模板的模板参数__D__。

  A)只可作为数据成员的类型

  B)只可作为成员函数的返回类型

  C)只可作为成员函数的参数类型

  D)以上三者皆可

课堂练习

- 模板是实现类属机制的一种工具,其功能非常强大,它既允许用户构造类属函数,即（B），也允许用户构造类属类,即（D）。

  A)模板函数　　B)函数模板　　C)模板类　　D)类模板

- 类模板的使用实际上是将类模板实例化成一个具体的（D）。

  A)类　　B)对象　　C)函数　　D)模板类

- 关于函数模板，描述错误的是（D）。

  A) 函数模板实例化为可执行的模板函数

  B) 函数模板的实例化由编译器实现

  C) 一个类定义中，只要有一个函数模板，则这个类是类模板

  D) 类模板的~~成员函数都是函数模板~~，类模板实例化后，成员函数也随之实例化

B/D D D

课堂练习

- 关于类模板，描述错误的是（AD）

  A) 一个普通基类不能派生类模板

  B) 类模板从普通类派生，也可以从类模板派生

  C) 根据建立对象时的实际数据类型，编译器把类模板实例化为模板类

  D) 函数的类模板参数须通过构造函数（编译器）实例化

- 需要一种逻辑功能能一样的函数，而编制这些函数的程序文本完全一样，区别只是数据类型不同。对于这种函数，下面不能用来实现这一功能的选项是（D）

  A）宏函数　　B）为各种类型都重载这一函数

  C）模板　　D）友元函数

课堂练习

- 分析以下程序的执行结果

```
template <class T>
class Sample
{       T n;
  public:
        Sample(T i) { n=i; }
        void operator++();
        void disp() { cout<<“n=”<<n<<endl; }    };
template <class T>
void Sample<T>::operator++()无参数，前加
{   n+=1;     // 不能用n++; 为什么？ }n的类型是T，有可能是浮点类型
void main()
{  Sample<char>  s('a');
    s++;
   s.disp();   }
```

7.2 示例：模板堆栈类

```
#include <iostream>
#include <string>
//#define NDEBUG //**** enable/disable assertions
#include <cassert>
using namespace std;

template< class T >
class Stack {
public:
   enum { DefaultStack = 50, EmptyStack = -1 };
   Stack()  {  size = DefaultStack;   allocate(); }
   Stack( int );
```

```
    ~Stack() { delete[ ] elements;  }
    void push( const T& );
    T pop();
    T topNoPop() const;
    bool empty() const  {  return top <= EmptyStack;  }
    bool full() const  {   return top + 1 >= size; }
private:
    T* elements;
    int top;
    int size;
    void allocate()  { elements = new T[ size ];  top = EmptyStack;   }
    void msg( const char* m ) const  {
        cout << "*** " << m << " ***" << endl;   }
    friend ostream& operator<<( ostream&, const  Stack<T>& );
};
```

7.2 示例：模板堆栈类

```
template< class T >
Stack<T>::Stack( int s ) {
   if ( s < 0 )       // negative size?
     s *= -1;
   else if ( 0 == s ) // zero size?
     s = DefaultStack;
   size = s;
   allocate();
}
```

7.2 示例：模板堆栈类

```
template< class T >
void Stack<T>::push( const T& e ) {
      assert( !full() ); /* 断言 ,如果 !full() 值为假(即为0， full()为真),
                            那么它先向stderr打印一条出错信息,
                            然后终止程序运行。*/
   if ( !full() )
     elements[ ++top ] = e;
   else
     msg( "Stack full!" );
}
```

7.2 示例：模板堆栈类

```
template< class T >
T Stack< T >::pop() {
   assert( !empty() );
   if ( !empty() )
     return elements[ top-- ];
   else {
     msg( "Stack empty!" );
     T dummy_value=0;              //
     return dummy_value; // return arbitrary value
   }
}
```

7.2 示例：模板堆栈类

```
template< class T >
T Stack< T >::topNoPop() const {
   assert( top > EmptyStack );
   if ( !empty() )
     return elements[ top ];
   else {
     msg( "Stack empty!" );
     T dummy_value ;
     return dummy_value;  // Warning
   }
}
```

7.2 示例：模板堆栈类

```
template< class T >
ostream& operator<<( ostream& os, const Stack<T>& s ) {
   s.msg( "Stack contents:" );
   int t = s.top;
   while ( t > s.EmptyStack )  cout << s.elements[ t-- ] << endl;
   return os;
}
```

7.2 示例：模板堆栈类

```
int main() {  //将输入的代数表达式逆置输出
    const string prompt = "Enter an algebraic expression: ";
    const char lParen = '(';
    const char rParen = ')';
    Stack< char >  s;
    string buf;
    cout<< prompt << endl;
    getline( cin, buf );
    for ( int i = 0; i < static_cast<int>(buf.length()); i++)
    {  if ( !isspace( buf[i] )  )  s.push( buf[i] );   }
```

7.2 示例：模板堆栈类

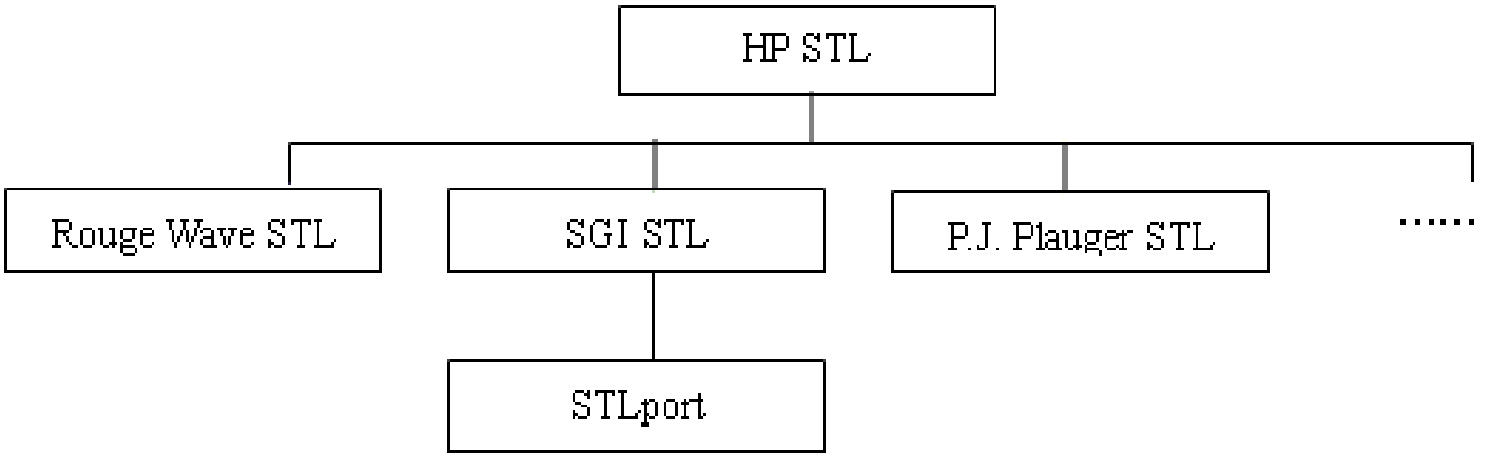
```
        cout<< "Original expression:  "  <<buf   <<endl;
        cout<< "Expression in reverse:  ";
        while  (!s.empty())
        {           char  t = s.pop();
                    if (t == lParen)  t = rParen;
                              else if (t == rParen)  t = lParen;
                    cout<<t;  }
        cout<<endl;
return 0;
```

7.3 标准模板库STL (Standard Template Library)

- C++包含一个有许多组件的标准模板库(Standard Template Library，STL)，它是标准C++库的一部分。
- STL是由Alexander Stepanov(STL之父)、David R Musser和Meng Lee三位大师共同发展，于1994年被纳入C++标准程序库。
- 新的C++标准库中几乎每一样东西都是由模板(Template)构成的。

STL的不同实现版本

- HP版本
- P.J. Plauger 版本（被vc++采用）
- Rouge Wave版本（被c++Builder采用)
- STL port版本
- SGI STL 版本（被GCC采用）



7.3 标准模板库STL

- 7.3.1容器、算法和迭代器
  - 容器(containers)：各种对象（数据结构）的集合，如vector，stack，queue，deque，list，set，map等。
  - 算法(algorithms)：是对容器进行处理的函数，通常是各种常用算法，如sort,search,copy， merge，permute（序列改变）等。
  - 迭代器(iterators)：提供遍历容器的方法，可分别进行正向、反向、双向和随机遍历操作。

7.3.2 STL的优越性

- 容器的大小自动变化
- 有大量处理容器的算法，且大多算法提供内置迭代功能
- 使用了迭代器（替代循环结构），STL容器和算法灵活而高效。
- STL是可扩展的

如下例（7-11，p272）

7.3.2 STL的优越性

例，将某集合v中每个元素用其平方值代替输出，再排序输出。

```
#include  < vector >
#include  < algorithm >
using   namespace  std;
void   sq (int &a)  { a=a*a; }
void print(int a)  {cout << a << ' '; }
void main()
{     vector <int>  v;
      //fill up v with value;
      for (int   i = 10; i > 0; i--)  //容器大小自动变化
        v.push_back( i ); //Adds an element to the end of the vector.
      for (int i = 0; i < 10; i++)  { cout<<v[i]<<",";  }  //10,9,...,1,
```

7.3.2 STL的优越性

```
    cout<<endl;

    for_each(v.begin(),  v.end(),  sq) ;
    for_each(v.begin(),v.end(),print) ;  //输出v中内容:100  81  64…
    cout<<endl;

    sort( v.begin(),  v.end() );
    for_each(v.begin(),v.end(),print) ;  //输出v中内容:1 4 9 16…
}
```

7.3.3 容器基础知识

- STL基本容器可以分为两组：序列式容器和关联式容器。
  （见下页）
- 序列式容器如同数组，其中的元素可以通过下标来访问。
  （双向链表list 除外）例如：

```
vector<int>  v;
int  x= v[ 2 ];
```

- 关联式容器中的元素可以通过键值来访问，关联式容器可将任意类型的数据作为键值来使用。

  如m是一个map型集合，可有

```
float gdp=m[“china”];
```

7.3.3 容器基础知识

基本的STL容器

容器	类型	描述
list	序列式	双向链表
vector	序列式	按需要伸缩的数组
deque	序列式	两端进行有效插入/删除的数组
set	关联式	不含重复键的集合（元素集合）
multiset	关联式	允许重复键的set
map	关联式	用键访问的不含重复键的集合（元素对集合）
multimap	关联式	允许重复键的map

7.3.4 基本序列式容器vector、deque和list

- vector 容器的使用语法及功能：

```
vector<int>   s;   // Create an empty vector s
```

或

```
// Create a vector d with 1000  elements of default value 0
vector<double>  d ( 1000 );
```

或

```
// Create a vector v with 5 elements of value 2
vector <int>  v( 5, 2);
```

或

```
// Create a copy, vector v2, of vector v
vector <int>  v2( v );
```

7.3.4*1 基本序列式容器:vector

- vector支持在两端插入元素，并提供begin和end成员函数，分别用来访问头部和尾部元素。
- 成员函数begin返回的迭代器:
  - 如果容器不为空，指向容器的第一个元素。
  - 如果容器为空，指向容器尾部之后的位置。
- 成员函数end返回的迭代器：
  - 仅指向容器尾部之后的位置。

例7-12 （p274）

7.3.4*1 基本序列式容器:vector

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{     int i;
      vector< int >  nums;
      nums.insert( nums.begin(), -999);
      nums.insert( nums.begin(), 15);
      nums.insert( nums.end(), 60);
      for ( i = 0; i < nums.size(); i++)
                  cout << nums[i] << endl;     //15 -999 60
      cout<< endl;
```

7.3.4*1 基本序列式容器:vector

```
        nums.erase( nums.begin() );       //-999 60
        nums.erase( nums.begin() );       //60
        for ( i = 0; i < nums.size(); i++)
                cout << nums[i] << endl;            //60

        return 0;
}
```

7.3.4*2 基本序列式容器:deque

- deque和vector使用语法差别不大，vector适合插入和删除在尾部进行的情况。 deque适合在两端进行插入和删除操作的情况。
- 使用时把vector出现的地方改为deque即可。

7.3.4*3 基本序列式容器:list

- list 容器的语法和特性:

  list<string> names;

- 拥有的迭代器类型：

  list<数据类型>：：iterator

  list<数据类型>：：const_iterator

- list不能通过下标访问。

- 例7-13

```
#include < iostream >
#include < string >
#include < list >
```

7.3.4*3 基本序列式容器:list

```
#include  < algorithm >
using namespace std;
void dump( list<string> &);
void print(string a) {cout<<a<<" ";}
int main()
{       list<string> names;
        names.insert( names.begin(), "Kamiko" );
        names.insert( names.end(),  "Andre" );
        names.insert( names.begin(), "Chengwen" );
        names.insert( names.begin(), "Maria" );
        for_each(names.begin(),names.end(),print); //输出M C K A
```

7.3.4*3 基本序列式容器:list

```
    names.reverse();            cout<<endl;
    dump( names );              return 0;
}
void dump( list< string >& la)
{     //for_each(la.begin(),la.end(),print);
      list<string >::const_iterator  it; //定义迭代器。删除书上const
      it = la.begin();    //初始化迭代器
      while( it != la.end() ) { cout<< *it <<" ";  it++; }
}
```

输出：

Maria Chengwen Kamiko Andre

Andre Kamiko Chengwen Maria

7.3.5 vector、deque、list效率比较

操作	vector	deque	list
在头部插入和删除元素	线性	恒定	恒定
在尾部插入和删除元素	恒定	恒定	恒定
在中间插入和删除元素	线性	线性	恒定
访问头部的元素	恒定	恒定	恒定
访问尾部的元素	恒定	恒定	恒定
访问中间的元素	恒定	恒定	线性

7.3.6 基本的关联式容器

- STL的四种关联式容器可以分成两组:set和map。
- set是一种集合,其中可包含0个或多个不重复的和不排序的元素，这些元素被称为键值。

  例如： set集合

  { jobs, gates, ellison}

- map是一种集合,其中可包含0个或多个不排序的元素对，一个元素是不重复的键值，另一个是与键相关联的值。

  例如：map集合

  {(jobs,apple),(gates,microsoft),(ellison,oracle)}

例7-14 set容器的用法

7.3.6 基本的关联式容器

```
#include < iostream >
#include < set >
using namespace std;
int main() {
        set<int>  s;
        s.insert( - 999 );
        s.insert( 18 );
        s.insert( 321 );
        s.insert( -999 );  //not inserted
        set< int >::const_iterator  it;
        it = s.begin();
        while ( it != s.end())  cout << *it++ << endl;  //-999  18  321
```

7.3.6 基本的关联式容器

```
    int key;   cout<< "Enter an integer: ";
    cin >> key;
    it = s.find( key );

    if  (it == s.end())
            cout<< key << "is not in set." <<endl;
    else
            cout<< key << "is in set." << endl;

    return 0;
}
```

Set举例

明明想在学校中请一些同学一起做一项问卷调查，为了实验的客观性，他先用计算机生成了N个1到1000之间的随机整数(N≤100)，对于其中重复的数字，只保留一个，把其余相同的数去掉，不同的数对应着不同的学生的学号。然后再把这些数从小到大排序，按照排好的顺序去找同学做调查。请你协助明明完成“去重”与“排序”的工作。

输入格式：输入有两行，第1行为1个正整数，表示所生成的随机数的个数N。第2行有N个用空格隔开的正整数，为所产生的随机数。

输出格式：输出也是两行，第1行为1个正整数M，表示不相同的随机数的个数。第2行为M个用空格隔开的正整数，为从小到大排好序的不相同的随机数。

样例

输入 10

20 40 32 67 40 20 89 300 400 15

输出 8

15 20 32 40 67 89 300 400

```
#include<iostream>
#include<set>
using namespace std;
int main()
{  set<int>s;  //
   int a,n;
   cin>>n;
   for(int i=0;i<n;i++)
    { cin>>a;
      s.insert(a);    }
    cout<<s.size()<<endl;
    set< int >::const_iterator  it = s.begin();
    while(it!=s.end()) cout<<*it++<<" ";
    return 0;
 }
```

Set举例

一个n个元素的整数数组，如果数组两个连续元素之间差的绝对值包括了 [1,n−1]之间的所有整数，则称之符合“欢乐的跳”，如数组1 4 2 3符合“欢乐的跳”，因为差的绝对值分别为： 3,2,1。 给定一个数组，你的任务是判断该数组是否符合“欢乐的跳”。

```
#include<bits/stdc++.h>
using namespace std;
int main() {
  int n,a,b,c;
  set<int>s;
  cin>>n>>b;//这里的b是第一个数字
  for(int i=2;i<=n;i++) {
        cin>>a;
        c=abs(a-b);
        if(c>=1&&c<=n-1) s.insert(c);  //记录差
        b=a;  //b存储上一个的数字
    }
    if(s.size()==n-1) cout<<"Jolly"<<endl;//如果个数够，就输出Jolly
    else cout<<"Not jolly"<<endl;
    return 0;
}
```

Set举例

给你n个棍子，棍子可以接起来，问你用尽所有棍子能够拼成多少个完全不同的三角形。

输入第一行包含一个整数n，表示n个棍子。（1 <= n <= 15）

下一行包含n个整数li，指示每个棍子的长度。（1 <= li <= 10000）

输出为一个整数，指示不同三角形的数量。

样本输入　　5

1 3 2 3 4

样本输出　　5

```
#include<iostream>
#include<set>
using namespace std;
set<long long> s;
int n,v[16];
bool istriangle(int a, int b, int c)
{ if(a<=0 || b<=0 || c<=0) return false;
  if(a+b>c && a+c>b && b+c>a) return true;
  return false;
}
```

Set举例

```
void dfs(int d, int a, int b, int c)
{ if(d==n)
  {   if(a>b || a>c || b>c ) return; //只需要取a<=b<=c的值进行判断即可
      if(istriangle(a,b,c))
      {      long long tmp=1000000000000LL*a+1000000LL*b+c;
             s.insert(tmp);
             //cout<<a<<","<<b<<","<<c<<","<<tmp<<endl;
       }
     return ;
  }
  dfs(d+1,a+v[d+1],b,c);     dfs(d+1,a,b+v[d+1],c);
  dfs(d+1,a,b,c+v[d+1]);
}
```

Set举例

```
int main()
{ int t;
  cin>>n;
  for(int i=1; i<=n; i++ ) cin>>v[i];
  s.clear();
  dfs(0,0,0,0) ;
  cout<<sum<<endl;
  return 0 ;
}
```

7.3.6 基本的关联式容器

- 例7-15 map容器的用法

```
#include < iostream>
#include < string >
#include < map >
using namespace std;
int main()
{     map< string, int>  m;
      m[ "zero" ] = 0;
      m[ "one" ] = 1;
      m[ "two" ] = 2;
      m[ "three" ] = 3;
     m[ "four" ] = 4;
```

7.3.6 基本的关联式容器

```
    m[ "five" ] = 5;
    m[ "six" ] = 6;
    m[ "seven" ] = 7;
    m[ "eight" ] = 8;
    m[ "nine" ] = 9;
    cout<< m[ "three" ]<<endl
        << m[ "five" ] <<endl
        << m[ "seven" ]<<endl;
    return 0;
}
```

输出：3

5

7

7.3.6 基本的关联式容器:multiset、multimap

- multiset是容许有重复键值的set，而multimap是容许有重复键值 map。

7.3.7 容器适配器

- 容器适配器是基本容器的衍生物，并利用基本容器来实现其特定的功能。容器适配器有三种：

  stack、queue、priority_queue

- stack、queue模板类默认衍生自deque，因此定义

  ```
  stack< char >  s;
  ```

  等价于

  ```
  stack<char,deque<char>> s;
  ```

  如果要改变成vector的衍生，则用如下方式：

  ```
  stack< char, vector< char> >  s;
  ```

- priority_queue模板类默认衍生自vector。

7.3.7 容器适配器

例7-16： stack示例：读入一串字符，删除空白后，倒序输出

```
#include < iostream >
#include < stack >
#include < cctype >
#include <string>
using namespace std;
int main()
{     const string prompt = "Enter an algebraic expression: ";
      const char lParen = '(';
      const char rParen = ')';
      stack< char >  s;
      string buf;
```

7.3.7 容器适配器

```
cout<< prompt << endl;
getline( cin, buf );
for ( int i = 0; i<buf.length(); i++)
{  if ( !isspace( buf[i] )  )  s.push( buf[i] );   }
cout<< "Original expression: “  <<buf   <<endl;
cout<< "Expression in reverse:  ";
while  (!s.empty())
{     char  t = s.top();  s.pop(); //取栈顶；出栈
      if (t == lParen)  t = rParen;  else if (t == rParen)  t = lParen;
      cout<<t;  }
cout<<endl;
return 0;
}
```

7.3.7 容器适配器

例7-17： queue示例

```
#include < iostream >
#include < queue >
using namespace std;
int main()
{     queue< int >  q;          // == queque<int, deque< int > >
      for (int j=-1, i = 0; i < 6; i++)  q.push( j = j + 2 );
      while ( !q.empty() )
      {  cout << q.front() << endl;  //return integer 返回头元素
         q.pop();                    //return void
      }
      return 0;
}
```

7.3.7 容器适配器

例7-18： priority_queue示例

```
#include  <iostream>
#include  <functional>
#include  <queue>
#include  <cstdlib>
#include  <ctime>
using namespace std;
int main()
{     const int HOW_MANY = 8;
      int i;
      priority_queue<int>  nums;
      srand( time( NULL ) );
```

7.3.7 容器适配器

```
    for ( i = 0; i < HOW_MANY ; i++)
    {  int next = rand();  cout << next << " ";  nums.push( next );  }
    cout << "\n*** Priority by value: "<< endl;
    for (i = 0; i < HOW_MANY ; i++)
    {  cout<<nums.top()<<endl;  nums.pop();  }
    return 0;
}
```

输出：

11236 6291 23442 30119 17311 29887 312 26992

*** Priority by value:

30119 29887 26992 23442 17311 11236 6291 312

7.3.8 其他容器

例7-19 string类也可当STL容器来用

```
#include  <iostream>
#include  <string>
#include  <algorithm>
using namespace std;
void printChar(char  c) { cout << c; }
int main()
{      string  s =  "pele, the greatest ever";
       cout<<"s:                  "<< s <<endl;
       cout<<"s in reverse: ";
       for_each( s.rbegin(), s.rend(), printChar); //反转开始和结束
       cout<<endl;
}
```

7.3.8 其他容器

- bitset类

bitset是一个二进制序列。可以用来表示二进制的整数。

使用形式：

```
bitset<8>  bs1;  //8位0
bitset<128>  bs2;  //128位0
bitset<8> bs( 9 );   // 8位二进制表示的9
```

bitset对[ ]操作符进行了重载，可通过下标来访问bitset中的某一位。

7.3.8 其他容器

- bitset类的成员函数功能:
  - 将某个指定位设为0或1
  - 对某个位或所有的位求反
  - 测试某个位为0或1
  - 进行左移或右移操作
  - 进行与、或和异或等标准二进制操作
  - 将bitset转型为string或unsigned long整型

7.3.9 STL算法

- STL有大量用来处理容器的算法。
- 如同STL容器用模板类实现一样，STL算法是用模板函数实现。
- 算法部分主要由头文件<algorithm>，<numeric>和<functional>组成。

  <algorithm>由一大堆模版函数组成的，包括绝大多数常用到查找、遍历操作、复制等等。

  <numeric>体积很小，只包括几个在序列上面进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。

  (要使用 STL中的算法函数必须包含头文件<algorithm>，对于数值算法须包含<numeric>)

  <functional>中则定义了一些模板类，用以声明函数对象。

7.3.9 STL算法

- 从对容器的访问性质说，算法分为:

  只读形式(即不允许修改元素) 和 改写(即可修改元素)形式

- 从功能上说，可以分为:

  查找、比较、计算、排序、置值、合并、集合、管理等。

7.3.9 STL算法

例7-21 四种算法使用演示：generate，replace_if，sort和for_each。

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
using namespace std;
void dump(int i) { cout<<i<<endl; }
bool odd(int i) { return i % 2 != 0;}
//用于sort，如果sort的第一个参数比第二个参数大,comp返回true
//以这种形式实现降序排序
bool comp(const int& i1, const int& i2) { return i1>i2; }
```

7.3.9 STL算法

```
int main()
{   srand(time(NULL));
    vector< int >  v(10);  //定义大小为10的int型向量
    generate(v.begin(), v.end(), rand);  //fill with random ints
    replace_if(v.begin(), v.end(), odd, 0);  //replace odds with 0
    sort(v.begin(), v.end(), comp);  //sort in descending ord
    for_each(v.begin(), v.end(), dump);  //Output

    return 0;
}
```

7.3.9 STL算法

下面给出所有STL算法的功能描述

1）查找算法(13个)：判断容器中是否包含某个值

- adjacent_find: 在iterator对标识元素范围内，查找一对相邻重复元素，找到则返回指向这对元素的第一个元素的ForwardIterator。否则返回last。
- binary_search: 在有序序列中查找value，找到返回true。
- count: 利用等于操作符，把标志范围内的元素与输入值比较，返回相等元素个数。
- count_if: 利用输入的操作符，对标志范围内的元素进行操作，返回结果为true的个数。

7.3.9 STL算法

- equal_range: 功能类似equal，返回一对iterator，第一个表示lower_bound，第二个表示upper_bound。
- find: 利用底层元素的等于操作符，对指定范围内的元素与输入值进行比较。当匹配时，结束搜索，返回该元素的一个InputIterator。
- find_end: 在指定范围内查找"由输入的另外一对iterator标志的第二个序列"的最后一次出现。找到则返回最后一对的第一个ForwardIterator，否则返回输入的"另外一对"的第一个ForwardIterator。
- find_first_of: 在指定范围内查找"由输入的另外一对iterator标志的第二个序列"中任意一个元素的第一次出现。
- find_if: 使用输入的函数代替等于操作符执行find。

7.3.9 STL算法

- lower_bound: 返回一个ForwardIterator，指向在有序序列范围内的可以插入指定值而不破坏容器顺序的第一个位置。
- upper_bound: 返回一个ForwardIterator，指向在有序序列范围内插入value而不破坏容器顺序的最后一个位置，该位置标志一个大于value的值。
- search: 给出两个范围，返回一个ForwardIterator，查找成功指向第一个范围内第一次出现子序列(第二个范围)的位置，查找失败指向last1。
- search_n: 在指定范围内查找val出现n次的子序列。

7.3.9 STL算法

2）排序和通用算法(14个)：提供元素排序策略

- inplace_merge: 合并两个有序序列，结果序列覆盖两端范围。重载版本使用输入的操作进行排序。
- merge: 合并两个有序序列，存放到另一个序列。重载版本使用自定义的比较。
- nth_element: 将范围内的序列重新排序，使所有小于第n个元素的元素都出现在它前面，而大于它的都出现在后面。
- artial_sort:对序列做部分排序，被排序元素个数正好可以被放到范围内。
- partial_sort_copy: 与partial_sort类似，不过将经过排序的序列复制到另一个容器。

7.3.9 STL算法

- partition: 对指定范围内元素重新排序，使用输入的函数，把结果为true的元素放在结果为false的元素之前。
- random_shuffle: 对指定范围内的元素随机调整次序。
- reverse: 将指定范围内元素重新反序排序。
- reverse_copy: 与reverse类似，不过将结果写入另一个容器。
- rotate: 将指定范围内元素移到容器末尾，由middle指向的元素成为容器第一个元素。
- rotate_copy: 与rotate类似，不过将结果写入另一个容器。
- sort: 以升序重新排列指定范围内的元素。
- stable_sort: 与sort类似，不过保留相等元素之间的顺序关系。
- stable_partition: 与partition类似，不过不保证保留容器中的相对顺序。

7.3.9 STL算法

3) 删除和替换算法(15个)

- copy: 复制序列。
- copy_backward: 与copy相同，不过元素是以相反顺序被拷贝。
- iter_swap: 交换两个ForwardIterator的值。
- remove: 删除指定范围内所有等于指定元素的元素。注意，该函数不是真正删除函数。内置函数不适合使用remove和remove_if函数。
- remove_copy: 将所有不匹配元素复制到一个制定容器，返回OutputIterator指向被拷贝的末元素的下一个位置。
- remove_if: 删除指定范围内输入操作结果为true的所有元素。
- remove_copy_if: 将所有不匹配元素拷贝到一个指定容器。
- replace: 将指定范围内所有等于vold的元素都用vnew代替。

7.3.9 STL算法

- replace_copy: 与replace类似，不过将结果写入另一个容器。
- replace_if: 将指定范围内所有操作结果为true的元素用新值代替。
- replace_copy_if: 与replace_if，不过将结果写入另一个容器。
- swap: 交换存储在两个对象中的值。
- swap_range: 将指定范围内的元素与另一个序列元素值进行交换。
- unique: 清除序列中重复元素，和remove类似，它也不能真正删除元素。
- unique_copy:与unique类似，不过把结果输出到另一个容器。

7.3.9 STL算法

4）排列组合算法**(2个)**：提供计算给定集合按一定顺序的所有可能排列组合。

- next_permutation: 取出当前范围内的排列，并重新排序为下一个排列。
- prev_permutation: 取出指定范围内的序列并将它重新排序为上一个序列。如果不存在上一个序列则返回false。

7.3.9 STL算法

5) 算术算法(4个)

- accumulate: iterator对标识的序列段元素之和，加到一个由val指定的初始值上。
- partial_sum: 创建一个新序列，其中每个元素值代表指定范围内该位置前所有元素之和。
- inner_product: 对两个序列做内积(对应元素相乘，再求和)并将内积加到一个输入的初始值上。
- adjacent_difference: 创建一个新序列，新序列中每个新值代表当前元素与上一个元素的差。

7.3.9 STL算法

6) 生成和异变算法(6个)

- fill: 将输入值赋给标志范围内的所有元素。
- fill_n: 将输入值赋给first到first+n范围内的所有元素。
- for_each: 用指定函数依次对指定范围内所有元素进行迭代访问，返回所指定的函数类型。该函数不得修改序列中的元素。
- generate: 连续调用输入的函数来填充指定的范围。
- generate_n: 与generate函数类似，填充从指定iterator开始的n个元素。
- transform: 将输入的操作作用与指定范围内的每个元素，并产生一个新的序列。重载版本将操作作用在一对元素上，另外一个元素来自输入的另外一个序列。结果输出到指定容器。

7.3.9 STL算法

7) 关系算法(8个)

- equal: 如果两个序列在标志范围内元素都相等，返回true。
- includes: 判断第一个指定范围内的所有元素是否都被第二个范围包含，使用底层元素的<操作符，成功返回true。
- lexicographical_compare: 比较两个序列。
- max: 返回两个元素中较大一个。
- max_element: 返回一个ForwardIterator，指出序列中最大的元素。
- min: 返回两个元素中较小一个。重载版本使用自定义比较操作。
- min_element: 返回一个ForwardIterator，指出序列中最小的元素。
- mismatch: 并行比较两个序列，指出第一个不匹配的位置，返回一对iterator，标志第一个不匹配元素位置。如果都匹配，返回每个容器的last。

7.3.9 STL算法

8) 集合算法(4个)

- set_union: 构造一个有序序列，包含两个序列中所有的不重复元素。
- set_intersection:构造一个有序序列，其中元素在两个序列中都存在。
- set_difference: 构造一个有序序列，该序列仅保留第一个序列中存在的而第二个中不存在的元素。重载版本使用自定义的比较操作。
- set_symmetric_difference: 构造一个有序序列，该序列取两个序列的对称差集(并集-交集)。

7.3.9 STL算法

9) 堆算法(4个)

- make_heap: 把指定范围内的元素生成一个堆。
- pop_heap: 并不真正把最大元素从堆中弹出，而是重新排序堆。它把first和last-1交换，然后重新生成一个堆。可使用容器的back来访问被"弹出"的元素或者使用pop_back进行真正的删除。
- push_heap: 假设first到last-1是一个有效堆，要被加入到堆的元素存放在位置last-1，重新生成堆。在指向该函数前，必须先把元素插入容器后。
- sort_heap: 对指定范围内的序列重新排序，它假设该序列是个有序堆。

例7-22

```
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;
void print(const char* msg, char  a[], int len) {
      cout<<msg;
      copy(a, a+len, ostream_iterator<char>(cout, “ ”);
}
int main() {
    const int LEN = 27;
    const int MED = LEN / 2;
    char  alph[] = “abcdefghijklmnopqrstuvwxyz{”;
```

```
    print("Original array:\n", alph, LEN );
    //用STL算法random_shuffle将alph数组字符打乱
    random_shuffle(alph, alph + LEN);
    print("\nAfter random_shuffle:\n", alph, LEN );
    /*nth_element(start, start+n, end) 方法可以使第n大元素处于第n位置，并且比这个元素小的元素都排在这个元素之前，比这个元素大的元素都排在这个元素之后，但不能保证他们是有序的*/
    nth_element(alph, alph + MED, alph + LEN);
    print("\n\nAfter nth_element:\n", alph, LEN );
    print("\n\t < median: ", alph, MED );
    print("\n\t    median: ", alph+MED ,1);
    print("\n\t > median: ", alph+MED+1 ,LEN / 2);
    cout<<endl;
    return 0;
}
```

程序输出：

Original array:

a b c d e f g h i j k l m n o p q r s t u v w x y z {

After random_shuffle:

m b j y a l z v e p s f o n x q { d u r i t w k c h g

After nth_element:

a b c d e f g h i j k l m n o p q r s t u v w x y z {

< median: a b c d e f g h i j k l m

median: n

> median: o p q r s t u v w x y z {

几个常用的模板函数

- Unique函数

  unique(首地址,尾地址)

  该函数的作用是“去除”容器或者数组中相邻元素的重复出现的元素

  (1) 这里的去除并非真正意义的erase，而是将重复的元素放到容器的末尾，返回值是去重之后的尾地址。

  (2) unique针对的是相邻元素，所以对于顺序错乱的数组成员，或者容器成员，需要先进行排序，可以调用sort()函数

♦ Unique函数举例

现有n个正整数，n≤10000，要求出这n个正整数中的第k个最小整数（相同大小的整数只计算一次），k≤1000，正整数均小于30000。

```
#include<bits/stdc++.h>
using namespace std;
int main()
{  int n,k,a[10000];              cin>>n>>k;
   for(int i=0;i<n;i++) cin>>a[i];
   sort(a,a+n);//快排数组a
   int ans=unique(a,a+n)-a;//数组a去重，并保留去重后元素长度=> ans
   if(k<ans) cout<<a[k-1]; //如果去重以后k<=ans，则输出对应的数
   else cout<<"NO RESULT";//否则输出 NO RESULT
   return 0;
}
```

几个常用的模板函数

- partition() 和 stable_partition()

  ForwardIt partition(ForwardIt first, ForwardIt last,

  UnaryPredicate p);

  对[first, last)元素进行处理，使得满足p的元素移到[first, last)前部，不满足的移到后部，返回第一个不满足p元素所在的迭代器，如果都满足的话返回last。

  两个方法的区别在于， partition()对于两个子序列中的元素不稳定，而stable_partition()则对两个子序列的元素稳定。

举例：

从键盘输入n(n<100) 个整数（以0 结束），存放在一个一维数组中，将它们按奇数在前、偶数在后， 奇数或偶数按从小到大的顺序排序， 并输出排序后的结果。

输入： 10 2 7 9 11 5 4 3 6 8 20 0

输出： 3 5 7 9 11 2 4 6 8 10 20

```
#include<iostream>
#include <algorithm>
using namespace std;
bool f(int n) { return n%2;}
int main()
{ int n,a[100],*p=a;
  while(cin>>*p && *p!=0) p++;
  n=p-a;    sort(a,p);    stable_partition(a,a+n,f);
  for(int *p=a;p<a+n;p++) cout<<*p<<" ";
  return 0;
}
```

一道华为面试题

- int A[nSize]，其中隐藏着若干0，其余非0整数，写一个函数int Func(int* A, int nSize)，使A把0移至后面，非0整数移至数组前面并保持有序，返回值为原数据中第一个元素为0的下标。

一道华为面试题

```
#include<iostream>
#include<algorithm>
using namespace std;
bool f(int n) { return n; }
int Func(int* A, int nSize)
{ sort(A,A+nSize);
  return stable_partition(A,A+nSize,f)-A;
}
int main()
{ int a[]={1,0,2,0,-1,0,5,-2,0,6};
  cout<<Func(a,10);
  return 0;
}
```

*迭代器

- 简单地说，迭代器是面向对象版本的指针，STL算法利用迭代器对存储在容器中的元素序列进行遍历，迭代器提供了访问容器和序列中每个元素的方法。
- 虽然指针也是一种迭代器，但迭代器却不仅仅是指针。指针可以指向内存中的一个地址，通过这个地址就可以访问相应的内存单元。而迭代器更为抽象，它可以指向容器中的一个位置。
- 所有的标准库容器都定义了相应的迭代器类型。迭代器对所有的容器都适用，C++ 程序更倾向于使用迭代器而不是下标操作访问容器元素。

*迭代器

- STL容器类定义中用typedef预定义迭代器：

预定义迭代器	++操作的方向	功能
iterator	向前	读/写
const_iterator	向前	读
reverse_iterator	向后	读/写
const_reverse_iterator	向后	读

*迭代器

- 作用：

  遍历容器内的元素，并访问这些元素的值。

- 区别

  iterator可以改元素值,但const_iterator不可改， const_iterator自身的值可以改(可以指向其他元素)，但不能改写其指向的元素值。

- const iterator与const_iterator

  声明一个 const iterator时，必须初始化它。一旦被初始化后，就不能改变它的值,它一旦被初始化后,只能用它来改它指的元素,不能使它指向其他元素。(因此const iterator几乎没什么用途)

7.3.10 其他STL构件

- 函数对象、函数适配器、STL allocator

  函数对象是一个类对象，它对 函数调用操作符() 进行重载，并且该重载函数是公有的。可以用STL函数对象来代替普通函数。

  例7-24：

  前例

```
void dump(int i)  {  cout<<i<<endl;  }
for_each(v.begin(), v.end(), dump);
```

  作为另一种实现方式，可设计

```
template<class T>
struct dumpIt {
void operator()(T arg)  { cout<<arg<<endl;}
};
```

7.3.10 其他STL构件

然后调用这个模板的int型实例的重载调用操作符：

```
for_each(v.begin(),v.end(),dumpIt<int>() );
```

- 函数适配器是用来以现有函数对象创建新函数对象的构件。
- STL allocator 是一个模板类，用于内存管理。

例：双向链表基本操作

- 编写一个对双向链表进行基本操作的程序。要求能从两端开始插入、删除和输出结点。

```
#include <iostream>
#include <list>//包含双向链表容器头文件
#include <iterator>//迭代器头文件，可以省略
#include<algorithm> //STL算法
using namespace std;
int main()
{      list< int >List;
       int Value=0;       //插入的节点值
       int Option=0;      //操作选择
```

```
do{
    cout<<endl
            <<"        双向链表菜单"<<endl
            <<" 1. 在链表首部插入一个结点 "<<endl
            <<" 2. 在链表尾部插入一个结点 "<<endl
            <<" 3. 从链表首部删除一个结点 "<<endl
            <<" 4. 从链表尾部删除一个结点 "<<endl
            <<" 5. 从链表首部开始输出结点内容 "<<endl
            <<" 6. 从链表尾部开始输出结点内容 "<<endl
            <<" 0. 退出 "<<endl
            <<"输出选择：";
    cin>>Option;
```

```
switch(Option)
{ case 1:          //在链表首部插入一个结点
        {          cout<<"输入结点数据：";  cin>>Value;
                   List.insert(List.begin(),Value);
                   cout<<"结点"<<Value<<"成功插入。"<<endl;
                   break;
        }
 case 2:           //在链表尾部插入一个结点
        {          cout<<"输入结点数据：";          cin>>Value;
                   List.insert(List.end(),Value);
                  cout<<"结点"<<Value<<"成功插入。"<<endl;
                   break;
        }
```

```
case 3:  //从链表首部删除一个结点
{        if(List.begin()==List.end())
                cout<<endl<<"没有链表，不能进行删除。";
         else
         {       List.erase(List.begin()) ;
                  cout<<endl<<"结点删除成功。"<<endl;    }
          break;
   }
```

```
case 4:        //从链表尾部删除一个结点
    {   if (List.begin()==List.end())
                        cout<<endl<<"没有链表，不能进行删除。";
        else
        {       List.erase(List.end());
                cout<<endl<<"结点删除成功。"<<endl;        }
        break;
    }
```

```
case 5:  //从链表首部开始输出结点内容
    {   list<int>::const_iterator p1;//p1是整型双向链表的迭代子
        if(List.begin()==List.end())
            cout<<endl<<"没有链表，没有结点输出。";
        else  {
            cout<<"从首部开始输出链表:"<<endl;
            for(p1=List.begin();p1!=List.end();p1++)
                cout<<*p1<<" ";   // 依次输出链表中所有元素
            cout<<endl;            }
        break;
    }
```

```
case 6: //从链表尾部开始输出结点内容
{     list<int>::reverse_iterator p2;//p2是整型双向链表的迭代子
      if (List.rbegin()==List.rend())
         cout<<endl<<"没有链表，没有结点输出。";
      else
      {  p2=List.rbegin(); //反向迭代指向最后一个元素
         cout<<"从尾部开始输出链表:："<<endl;
         while(p2!=List.rend())  //当反向迭代不指向第一个元素时
          {    cout<<*p2<<" ";   // 逆向输出链表中所有元素：
                 p2++;   }          }
      cout<<endl;       break;
}
```

```
        }
    } while(Option!=0);
    return 0;
    }
```