2023
面向对象程序设计

第七讲：模板

李际军
lijijun@cs.zju.edu.cn

学习目标 /GOALS

- 了解模板的概念；
- 掌握函数模板的定义和使用，理解函数模板与模板函数的关系；
- 掌握模板函数显式具体化；
- 掌握类模板的定义和使用，理解类模板与模板类的关系；
- 掌握类模板的派生；
- 掌握类模板的显式具体化。

前言 /PREFACE

- 模板是 C++ 支持参数化多态性的工具之一。
- C++ 的模板机制为泛化型程序设计提供了良好的支持。使用模板可以方便地建立起通用类型的函数库和类库，减少程序开发的重复及代码冗余。
- *模板* 是生成类或函数的框架。

  与类或函数显式指定数据类型不同，模板使用形参。

  当实际数据类型赋值给形参的时候，才由编译器生成类或函数。

目录 /Contents

01

模板的概念

[引例]

```
int  max( int x, int y)
{
   return (x>y) ? x: y;
}
double  max( double x, double y)
{
   return (x>y) ? x: y;
}
char  max( char x, char y)
{
   return (x>y) ? x:
}
```

可以看出，这些函数版本的功能都是相同的，只是参数类型和函数返回类型不同。

那么能否为这些函数只写出一套代码呢？

C++解决这个问题的一个方法就是使用模板。

7.1 模板的概念

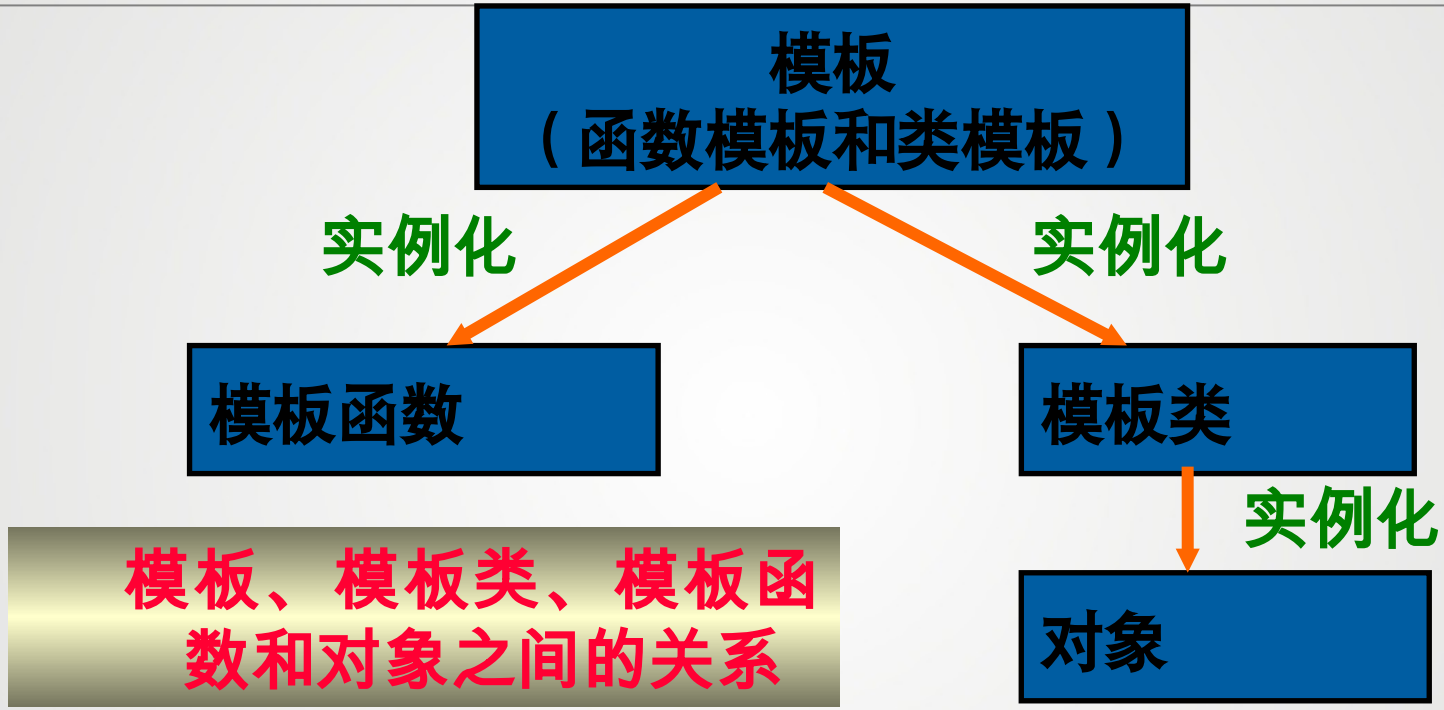
- 在 C++ 中，模板是实现代码重用机制的一种工具，它可以实现类型参数化，即把类型定义为参数，从而实现代码的可重用性。
- C++ 程序由类和函数组成， C++ 中的模板也分为类模板和函数模板。

[例]

```
template<class T>
T max( T x, T y)
{
        return (x>y) ? x
}
```

这个以参数化表示的函数称为函数模板。

模板、模板函数、模板类和对象之间的关系



［说明］ 由于模板的作用是使程序能够对不同类型的数据进行相同方式的处理，因此，在进行相同方式的处理时，只有当参加运行的数据类型不同时，才可以定义模板。

02 函数模板与模板函数

- 函数模板

所谓函数模板，实际上是建立一个通用函数，其函数类型和形参类型不具体指定，用一个虚拟的类型（如： T ）来代替，这个通用函数就称为函数模板。

- 模板函数

在定义了一个函数模板后，当编译系统发现有一个对应的函数调用时，将根据实参中的类型来确认是否匹配函数模板中对应的形参，然后生成一个重载函数，该函数的定义与函数模板的函数定义体相同，称之为模板函数。

[比较] 函数模板和模板函数的区别

- 函数模板是模板的定义，定义中用到通用类型参数。
- 模板函数是实实在在的函数定义，它由编译系统在遇到具体函数调用时所生成，具有程序代码。

同样，类模板是模板的定义，不是一个实实在在的类，其定义也用到通用类型参数。在定义了一个类模板后，可以创建类模板的实例，即生成模板类。

7.2.1 函数模板的定义和模板函数的生成

定义函数模板的一般形式:

```
template <class 类型参数名1, class 类型参数名2,
...>
函数返回值类型 函数名（形参表）
{
    //函数体
}
```

说明函数模板的关键字。

关键字class后面的类型参数名是模板形参，它可以代表基本数据类型，也可以代表类。

[例 7-1] 将求最大值的函数 max() 定义成函数模板。


```
template <class T>  // 模板声明
T max(T x, T y)             // 模板定义体
{
    return (x>y) ? x:y;
}
```

其中，T 是模板形参，它既可以取系统预定义的数据类型，也可以取用户自定义的类型。

[例 7-1] 将求最大值的函数 max() 定义成函数模板。

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

void main()
{
   cout<<max(5,6)<<endl;
}
```

定义

使用

［例 7-1］ 将求最大值的函数 max() 定义成函数模板。

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

```
int max(int x, int y)
{ return (x>y)? x : y; }
```

编译器自动生成！
隐式的！

```
void main()
{   cout<<max(5,6)<<endl;  }
```

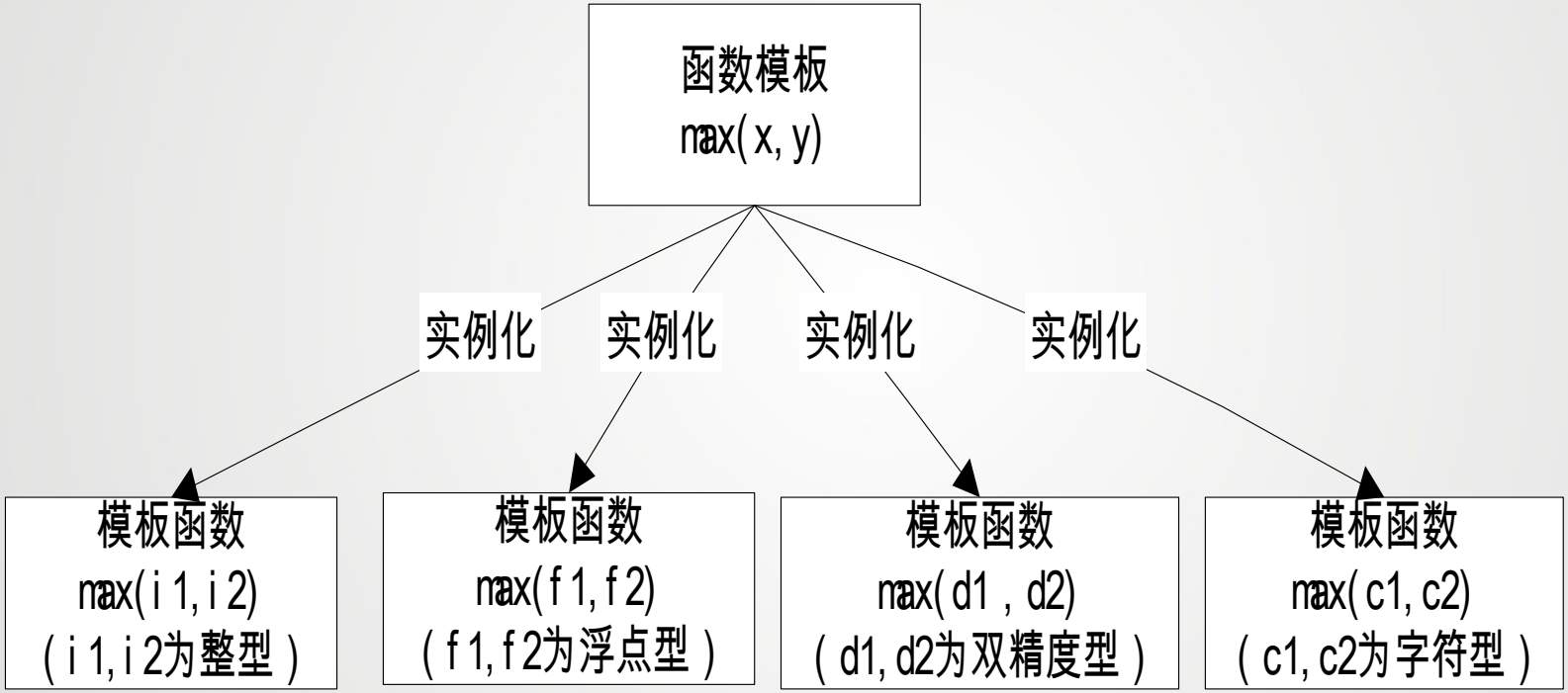
函数模板生成模板函数

看一段主程序的代码：

```
main()      // 主程序
{           int i1=10, i2=56;
            float f1=12.5, f2=24.5;
            double d1=50.344, d2=4656.346;
            char c1='k', c2='n';
            cout<<"the max of i1,i2 is:"<<max(i1,i2)<<endl;
                    // 根据传入的参数类型生成模板函数 max(int,int)
            cout<<"the max of f1,f2 is:"<<max(f1,f2)<<endl;
                    // 根据传入的参数类型生成模板函数 max(float,float)
            cout<<"the max of d1,d2 is:"<<max(d1,d2)<<endl;
                    // 根据传入的参数类型生成模板函数
max(double,double)
            cout<<"the max of c1,c2 is:"<<max(c1,c2)<<endl;
                    // 根据传入的参数类型生成模板函数 max(char,char)
            return 0;
}
```

函数模板生成模板函数

上面例子中，函数模板 max(x, y) 根据传入的参数类型不同，生成了四个模板函数：



函数模板和模板函数的区别：

- 函数模板是模板的定义，定义中用到的是通用的参数类型，它可以是任意类型 T 为参数和返回值。

- 模板函数是实实在在的函数定义，它是由编译系统碰见具体函数调用时生成的，具有函数代码。

- 函数模板实现了函数参数的通用性，作为代码的重用机制，可以大幅度的提高程序设计的效率。

注意

（1）在 template 语句和函数模板定义语句之间不允许有其他的语句，例如：

```
template<class T1,class T2>
int t;                              // 错误，不允许有其他的语句
T1 max(T1 x,T2 y)
{
    return (x>y)?x:y;
}
```

（2） 模板函数中的动作必须相同。例如，下面的函数只能用函数重载，而不能用模板函数。

```
void print(char *name)
{
   cout<<name<<endl;
}
void print(char *name,int no)
{
   cout<<name<<no<<endl;
}
```

（3）函数可以带有模板参数表中未给出的、已存在的数据类型的参数。

例如:

```
template <class T>
    T func2(T arg1 ,  int arg2)
    {
            …
    }
```

（4）虽然函数模板中的模板形参 T 可以实例化为各种类型，但实例化 T 的各模板实参之间必须保持完全一致的类型。模板类型并不具有隐式的类型转换，例如：在 int 与 char 之间、 float 与 int 之间、 float 与 double 之间等的隐式类型转换。

例如:

```
void func(int i,char c)
{
      max(i, i);    // 正确 , 调用 max(int,int)
      max(c, c);   // 正确 , 调用 max(char,char)
      max(i, c);    // 错误 , 类型不匹配
      max(c, i);    // 错误 , 类型不匹配
}
```

[例 7-2] 函数模板参数的问题，分析下面程序中的错误


```
#include <stdafx.h>
#include <iostream>
using namespace std;
template<class T >  // 模板声明
T max(T x,T y)     // 定义模板
{
      return(x>y)?x:y;
}
int main()
{     int i=4,j=8;
      char c='a',d='b';
      float f=23.5;
      double g=12222.222;
      cout<< “the max of i,j is:  ” <<max(i,j)<<endl;        // 正确。
      cout<<"the max of i,f is:  "<<max(f,i)<<endl;                    // 错误，类型不匹配
      cout<<"the max of i,c is:  "<<max(i,c)<<endl;                    // 错误，类型不匹配
      cout<<"the max of g,d is:  "<<max(g,d)<<endl;                    // 错误，类型不匹配
      return 0;
}
```

[解决方法]

◆ 1）采用强制类型转换。

[例] 将调用语句 max（f ,i） 改写为：

```
maximum(f, float(i))
```

◆ （2）显式给出模板实参，强制生成对特定实例的调用。具体地说，就是在调用格式中要插入一个模板实参表。

[例] 将调用语句 max（i,c） 和 max(g,d) 分别改写为：

```
max<int>(i,c)
max<double> (g,d)
```

◆ （3）将函数模板中< >的类型参数定义为两个类型参数分别为 T1 和 T2，分别接受不同的数据类型。函数模板的返回类型参数为 T1 或 T2 。

◆ （4）显式具体化模板函数。例如，我们通过重载模板函数实现比较字符串的大小。

模板函数显式具体化

① 声明一个非模板函数的原型：

```
template<class T>
T max(T x, T y)
{
        return (x>y)?x:y;
}
int max(int, int); // 只声明一个非模板函数的原型
void func(int i,char c)
{
    max(i, i);     // 正确 , 调用 max(int,int)
    max(c, c);  // 正确 , 调用 max(char,char)
    max(i, c);    // 正确 , 调用 max(int,int), 使用隐式类型
转换
    max(c, i);    // 正确 , 调用 max(int,int), 使用隐式类型转
换
}
```

模板函数显式具体化

② 定义一个完整的非模板函数重载模板函数

按照这种方式定义重载函数，所带的参数类型可以随意，就像一般的重载函数一样定义。比如：在上面程序的模板定义下面定义如下函数：

```
char *max(char *x, char *y)
{
      return (strcmp(x,y)>0)?x:y;
}
```

此函数重载了上述函数模板，当出现调用语句 max(“abcd”,“efgh”)；时，执行的是这个重载的非模板函数。

模板函数显式具体化

```
#include <iostream.h>
template <class T>
T Power(T a, int exp)
{
        T ans = a;
        while(--exp>0) ans*=a;
        return ans;
}
// 测试用主函数
int main()
{
        cout << "3^5= " <<Power(3, 5) << endl;
        cout << "1.1^2= " << Power(1.1, 2) << endl;
        return 0;
}
```

运算结果：

3^5=243

1.1^2=1.21

模板函数重载的调用顺序

在C++中函数模板与同名的非模板函数重载时，调用的顺序遵循下述约定：

1.寻找一个参数完全匹配的函数，如果找到就调用它。

2.寻找一个函数模板，将其实例化，产生一个匹配的模板函数。若找到了，就调用它。

3.若（1）和（2）都失败，再试一试低一级的对函数的重载方法。例如通过类型转换可产生参数匹配等，若找到了，就调用它。若（1），（2），（3）均未找到匹配的函数，则是一个错误的调用。如果在第（1）步有多于一个的选择，那么这个调用是意义不明确的，是一个错误调用。

03 类模板与模板类

类是对一组对象的公共性质的抽象，而类模板则是对一组类的公共性质的抽象。类模板是一系列相关类的模板，这些相关类的成员组成相同，成员函数的源代码形式也相同，不同的只是所针对的类型。类模板为类声明了一种模式，使得类中的某些数据成员、成员函数的参数和成员函数的返回值能取任意类型（包括系统预定的和用户自定义的）。

假如有两个或多个类的功能相同，只是数据类型不同。

```
class Compare_int
{
public :
            Compare(int a,int b)
            {
                x=a;y=b;            }
            int max( )
            {
            return (x>y)?x:y;}
            int min( )
            {
            return (x<y)?x:y;}
private :
            int x,y;
};
```

```
            class Compare_float
        {
            public :
                    Compare(float a,float b)
            {
               x=a;y=b;
            }
             float max( )
             {
                 return (x>y)?x:y;
             }
             float min( )
             {
                 return (x<y)?x:y;
             }
    private :
          float x,y;
};
```

类模板的定义格式如下:

```
template <class 类型参数名1， class 类型参数名2，…>
class  类名
{
      类声明体
};
```

例如，将上面两个类写成以下的类模板：

```
template <class T>    // 声明一个模板，虚拟类型名为 T
class Compare         // 类模板名为 Compare
{
public :
          Compare(T a, T b)
          {
                    x=a;y=b;
          }
          T max( )
          {
                    return (x>y)?x:y;
          }
          T min( )
          {
                    return (x<y)?x:y;
          }
private :
          T x,y;
};
```

类模板的使用：

类模板不是一个具体的、实际的类，而是代表一种类型的类，编译程序不会为类模板创建程序代码，但是通过对类模板的实例化生成一个具体的类（即模板类）和该具体类的对象。

其实例化的一般形式是：

类名＜实际的数据类型 1，实际的数据类型 2 ， …> 对象名

例如:

```
Compare <int> cmp(4,7);
```

在类模板名之后的尖括号中指定实际的类型为 int，编译系统就用 int 取代类模板中的类型参数 T，这样就把类模板实例化了，并生成了该整型类的一个对象 cmp。

[比较] 类模板与模板类的区别

▶ 类模板是模板的定义，不是一个实实在在的类，定义中用到通用类型参数；

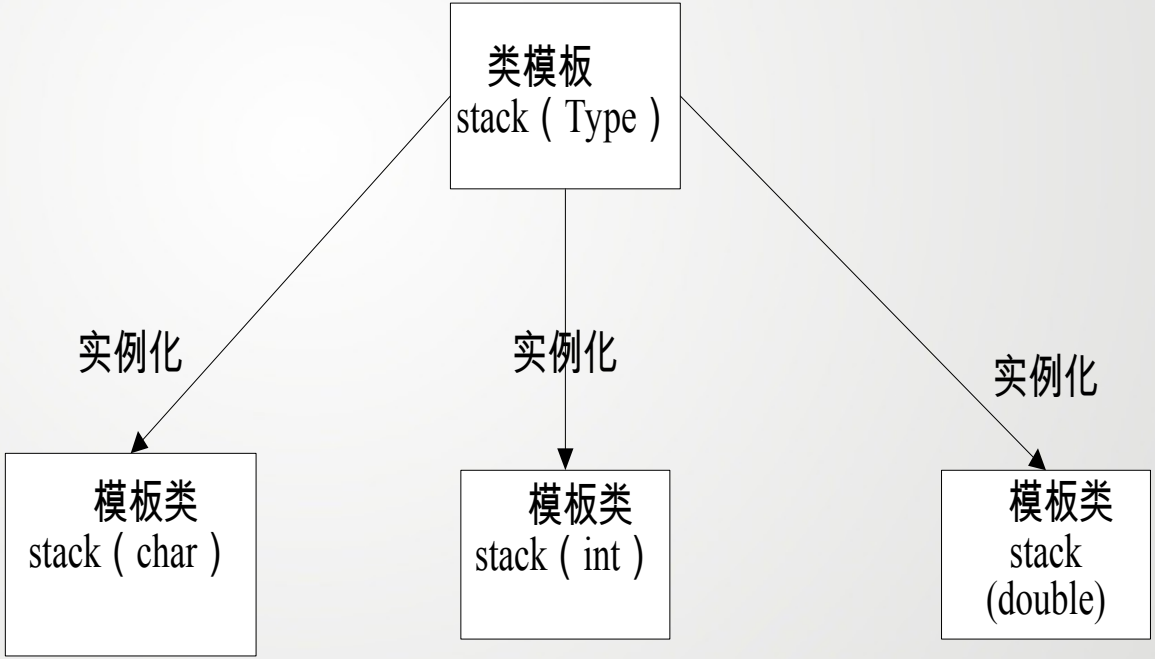
▶ 模板类是实实在在的类定义，是类模板的实例化。类定义中参数被实际类型所代替 .

例：

```
template<class T>
class Stack
 {
        数据成员;
        成员函数;
 };
```

直接使用:　声明对象

```
Stack<char>  st;
```



还可以在类定义体外定义成员函数:

其一般格式为:

template <class 类型参数名 1， class 类型参数名 2，…>

函数返回类型 类名<类型参数名 1, 类型参数名 2,…>:: 成员函数名（参数表）

{

函数体

}

模板参数可以是一个，也可以是多个，可以是类型参数，也可以是非类型参数。参数类型由关键字 class 或 typename 及其后面的标识符构成。非类型参数由一个普通参数构成，代表模板定义中的一个常量。

类模板的派生有 2 种方式:

- 1. 从类模板派生类模板,
- 2. 从类模板派生非模板类（普通类）。

1. 从类模板派生类模板：

从一个已有的类模板派生出新的类模板，格式如下：

```
template <class T>
class Base
{
    ...
};
template <class T>
class Derived:public Base<T>
{
    ...
};
```

2. 从类模板派生非模板类（普通类）：

从一个已有的类模板派生出非模板类，格式如下：

```
template <class T>
class Base
{
        ...
};
class Derived:public Base<int>
{
        ...
};
```

类模板显式具体化与模板函数显式具体化类似，也是特定类型用于替换模板中类型参数的定义。通过显式具体化类模板，可以优化类模板基于某种特殊数据类型的实现，可以克服某种特定数据类型在实例化类模板所出现的不足。

- 模板有两种特化：全特化和偏特化（也称为局部特化）。
- 全特化就是模板中模板参数全被指定为确定的类型。
- 偏特化就是模板中的模板参数没有被全部指定为确定的类型，需要编译器在编译时进行确定。
- 模板函数只能全特化，没有偏特化。而模板类是可以全特化和偏特化的。

例如，类模板 MyClass：

```
template<class T1,class T2>
class MyClass {…};
```

◆ 全特化:

```
template<>
class MyClass<int,int>{…};        // 全特化
```

◆ 偏特化

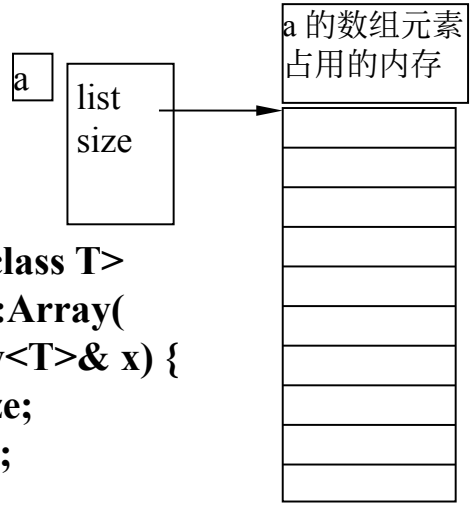
```
template<class T1>
class MyClass<T1,int>{…};         // 偏特化
```

```
#ifndef ARRAY_H
#define ARRAY_H
#include <cassert>
template <class T> // 数组类模板定义
class Array {
private:
        T* list;        // 用于存放动态分配的数组内存首地址
        int size;       // 数组大小（元素个数）
public:
        Array(int sz = 50);                     // 构造函数
        Array(const Array<T> &a);               // 拷贝构造函数
        ~Array();                               // 析构函数
        Array<T> & operator = (const Array<T> &rhs);            // 重载 "="
        T & operator [] (int i);  // 重载 "[]"
        const T & operator [] (int i) const;
        operator T * ();                        // 重载到 T* 类型的转换
        operator const T * () const;
        int getSize() const;        // 取数组的大小
        void resize(int sz);        // 修改数组的大小
};
```

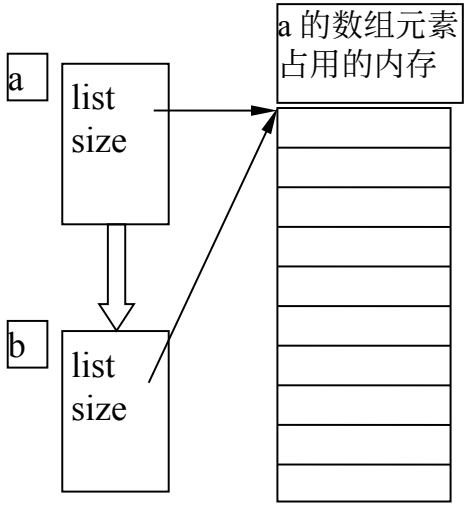
```
// 构造函数
template <class T>
Array<T>::Array(int sz) {
        //sz 为数组大小（元素个数），应当非负
        assert(sz >= 0);
        // 将元素个数赋值给变量 size
        size = sz;
        // 动态分配 size 个 T 类型的元素空间
        list = new T [size];
}
```

```
// 拷贝构造函数
template <class T>
Array<T>::Array(const Array<T> &a) {
    // 从对象 x 取得数组大小，并赋值给当前对象的成员
    size = a.size;
    // 为对象申请内存并进行出错检查
    list = new T[size];// 动态分配 n 个 T 类型的元素空间
    // 从对象 X 复制数组元素到本对象
    for (int i = 0; i < size; i++)
            list[i] = a.list[i];
}
```

浅拷贝



拷贝前

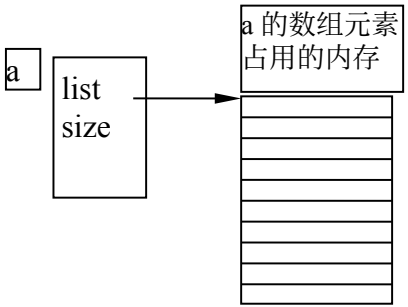


拷贝后

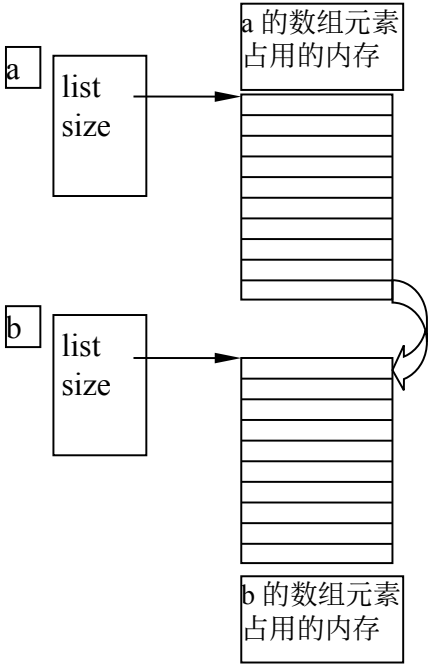
```
template <class T>
Array<T>::Array(
const Array<T>& x) {
  size = x.size;
  list = x.list;
}
```

```
int main() {
  Array<int> a(10);
  ......
  Array<int> b(a);
  ......
}
```

深拷贝


a
list
size
a 的数组元素
占用的内存

拷贝前


a
list
size
a 的数组元素
占用的内存
b
list
size
b 的数组元素
占用的内存

拷贝后

```
//重载“=”运算符
template <class T>
Array<T> &Array<T>::operator = (const Array<T>& rhs) {
        if (&rhs != this) {
                if (size != rhs.size) {
                        delete [] list;          //删除数组原有内存
                        size = rhs.size;         //设置本对象的数组大小
                        list = new T[size];      //重新分配 n 个元素的内存
                }
                //从对象 X 复制数组元素到本对象
                for (int i = 0; i < size; i++)
                        list[i] = rhs.list[i];
        }
        return *this;            //返回当前对象的引用
}
```

```
template <class T>
T &Array<T>::operator[] (int n) {
        assert(n >= 0 && n < size);// 越界检查
        return list[n];          // 返回下标为 n 的数组元素
}
template <class T>
const T &Array<T>::operator[] (int n) const {
        assert(n >= 0 && n < size); // 越界检查
        return list[n];          // 返回下标为 n 的数组元素
}
```

```
template <class T>
Array<T>::operator T * () {
    return list;              // 返回私有数组的首地址
}

template <class T>
Array<T>::operator const T * () const {
    return list;              // 返回私有数组的首地址
}
```

```
#include <iostream>
using namespace std;
void read(int *p, int n) {
 for (int i = 0; i < n; i++)
    cin >> p[i];
}
int main() {
 int a[10];
 read(a, 10);
 return 0;
}
```

```
#include "Array.h"
#include <iostream>
using namespace std;
void read(int *p, int n) {
 for (int i = 0; i < n; i++)
   cin >> p[i];
}
int main() {
 Array<int> a(10);
 read(a, 10);
 return 0;
}
```

Array 类的应用

- 例： 求范围 2~N 中的质数， N 在程序运行时由键盘输入。

```
#include <iostream>
#include <iomanip>
#include "Array.h"
using namespace std;
int main() {
    Array<int> a(10);    // 用来存放质数的数组，初始状态有 10 个元素。
    int n, count = 0;
    cout << "Enter a value >= 2 as upper limit for prime numbers: ";
    cin >> n;
    for (int i = 2; i <= n; i++) {
            bool isPrime = true;
            for (int j = 0; j < count; j++)
                    if (i % a[j] == 0) {    // 若 i 被 a[j] 整除，说明 i 不是质数
                            isPrime = false; break;                    }
            if (isPrime) {
                    if (count == a.getSize()) a.resize(count * 2);
                    a[count++] = i;}
    }
    for (int i = 0; i < count; i++)      cout << setw(8) << a[i];
    cout << endl;
    return 0;}
```

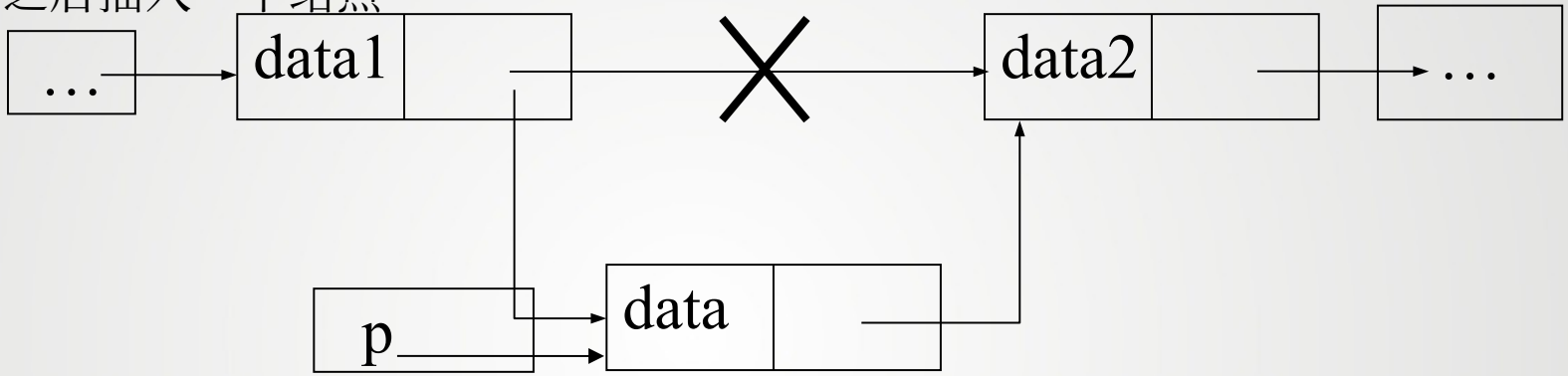
单链表的结点类模板


```
template <class T>
class Node {
private:
  Node<T> *next;
public:
  T data;
  Node(const T& item,Node<T>* next = 0);
  void insertAfter(Node<T> *p);
  Node<T> *deleteAfter();
  Node<T> *nextNode() const;
};
```

- 生成结点
- 插入结点
- 查找结点
- 删除结点
- 遍历链表
- 清空链表

单链表的结点类模板

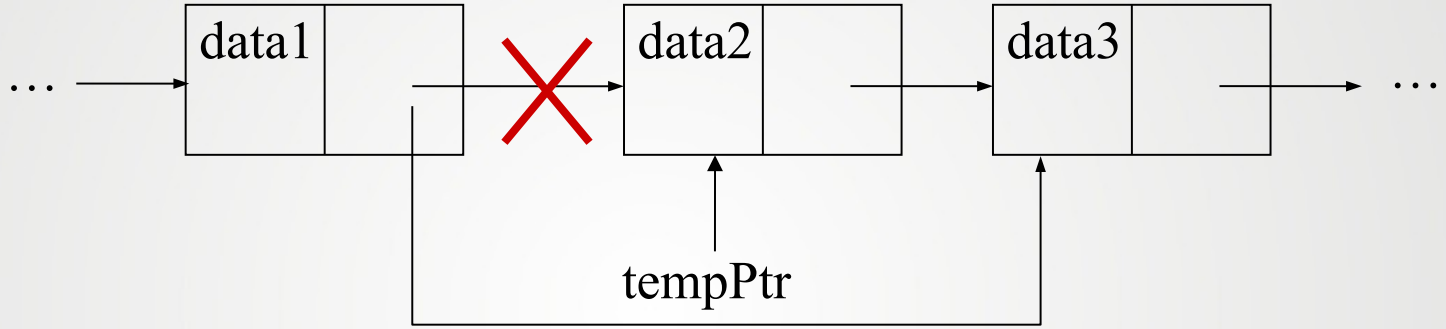
在结点之后插入一个结点




```
template <class T>
void Node<T>::insertAfter(Node<T> *p) {
  //p 节点指针域指向当前节点的后继节点
  p->next = next;
  next = p; // 当前节点的指针域指向 p
}
```

单链表的结点类模板

删除结点之后的结点




```
Node<T> *Node<T>::deleteAfter(void) {
  Node<T> *tempPtr = next;
  if (next == 0)
     return 0;
  next = tempPtr->next;
  return tempPtr;
}
```

链表类模板

```
#ifndef LINKEDLIST_H
#define LINKEDLIST_H
#include "Node.h"

template <class T>
class LinkedList {
private:
    // 数据成员:
    Node<T> *front, *rear
    Node<T> *prevPtr, *currPtr;
    int size;
    int position;

    Node<T> *newNode(const T &item,Node<T>
    *ptrNext=NULL);
    void freeNode(Node<T> *p);
    void copy(const LinkedList<T>& L);
public:
    LinkedList();
    LinkedList(const LinkedList<T> &L);
    ~LinkedList();
    LinkedList<T> & operator = (const
    LinkedList<T> &L);
    int getSize() const;
    bool isEmpty() const;
    void reset(int pos = 0
    void next();
    bool endOfList() const;
    int currentPosition(void) const;
    void insertFront(const T &item);
    void insertRear(const T &item);
    void insertAt(const T &item);
    void insertAfter(const T &item);
    T deleteFront();
    void deleteCurrent();
    T& data();
    const T& data() const
    void clear();
};

#endif  //LINKEDLIST_H
```

链表类模板

```
//9_7.cpp
#include <iostream>
#include "LinkedList.h"
using namespace std;
int main() {
        LinkedList<int> list;
        for (int i = 0; i < 10; i++) {
                int item;
                cin >> item;

list.insertFront(item);
        }
        cout << "List: ";
        list.reset();
        while (!list.endOfList()) {
                cout << list.data()
<< "  ";
                list.next();
        }
        cout << endl;
        int key;
        cout << "Please enter some integer needed to be deleted:
";
        cin >> key;
        list.reset();
        while (!list.endOfList()) {
                if (list.data() == key)
                        list.deleteCurrent();
                list.next();
        }
        cout << "List: ";
        list.reset();
        while (!list.endOfList()) {
                cout << list.data() << "  ";
                list.next
        }
        cout << endl;
        return 0;
}
```

第四部分

综合训练

综例 1

```
#include<iostream.h>
#include<stdlib.h>
template<class SType>
class Stack
{
public:
        Stack(int size);      // 构造函数
        ~Stack()                // 析构函数
        {delete [] s;}
        void push(SType i);         // 入栈操作
        SType pop();                  // 出栈操作
private:
        int tos,length;                  //length 为栈的空间长度，tos
为栈内已占空间的长度
        SType *s;
};
template <class Stype>
Stack<Stype>::Stack(int size)
{
    s=new Stype[size];
    if(!s)                                  // 如果内存分配不成功
```

综例 1

```
{
        cout<<"Can't Allocate stack."<<endl;
        exit(1);
    }
    length=size;
    tos=0;
}
template<class SType>
void Stack<SType>::push(SType i)
{
    if(tos==length)              //如果栈满
    {
        cout<<"Stack is full"<<endl;
    }
    s[tos]=i;
    tos++;
}
template <class Stype>
Stype Stack<Stype>::pop()
{
    if(tos==0)                   //如果栈空
    {
```

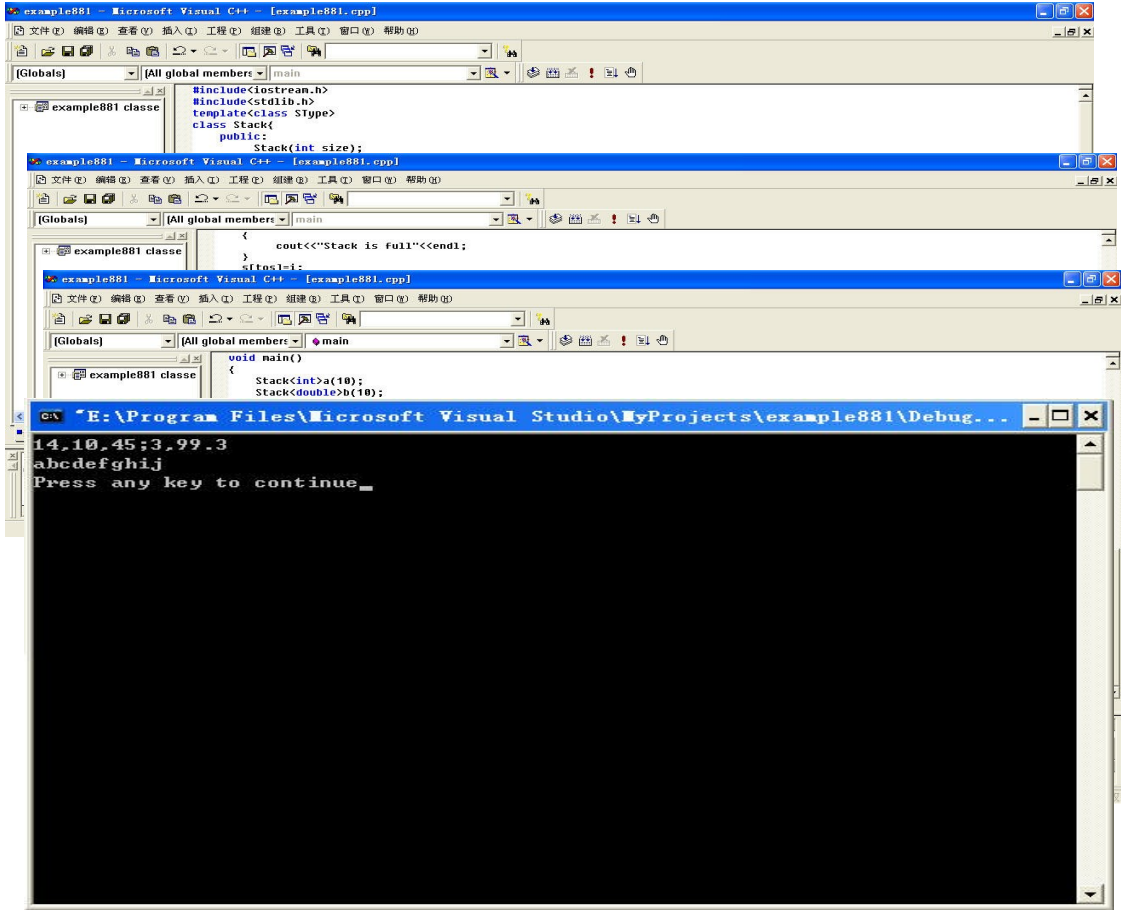
综例 1

```
 cout<<"Stack overflow."<<endl;
        return 0;
    }
    tos--;
    return s[tos];
}
void main()
{
    Stack<int>a(10);
    Stack<double>b(10);
    Stack<char>c(10);
    a.push(45);
    b.push(100-0.7);
    a.push(10);
    a.push(8+6);
    b.push(2*1.5);
    cout<<a.pop()<<",";
    cout<<a.pop()<<",";
    cout<<a.pop()<<";";
    cout<<b.pop()<<",";
    cout<<b.pop()<<endl;
    for(int i=0;i<10;i+
+)

c.push((char)'j'-i);
    for(i=0;i<10;i++)

cout<<c.pop();
    cout<<endl;
}
```

综例 1


example881 - Microsoft Visual C++ - [example881.cpp]
#include<iostream.h>
#include<stdlib.h>
template<class SType>
class Stack{
public:
Stack(int size);
cout<<"Stack is full"<<endl;
void main()
{
Stack<int>a(10);
Stack<double>b(10);
"E:\Program Files\Microsoft Visual Studio\MyProjects\example881\Debug...
14,10,45;3.99.3
abcdefghij
Press any key to continue

综例 2： 单向链表的类模板

要求：

设计一个单向链表的类模板，输出 26 个英文字母。

分析：

类模板不是一个具体的实际的类，使用时必须首先将其实例化为具体的模板类，然后再通过模板类定义对象。

一个完整的参考程序如下：

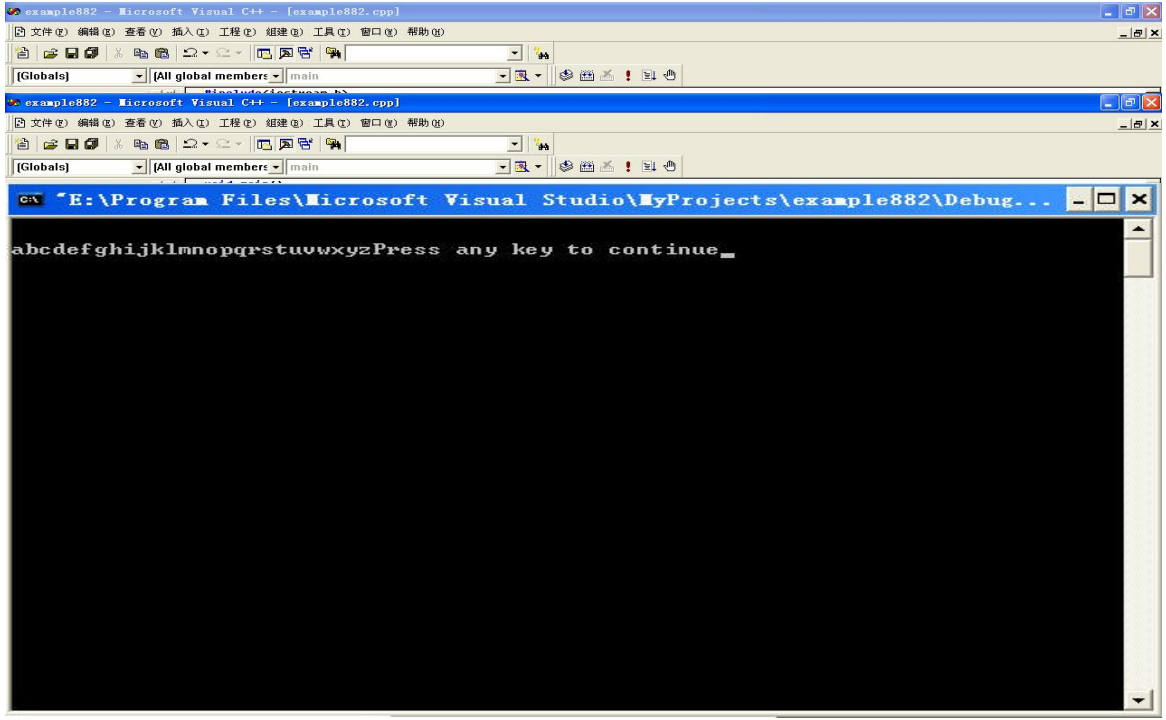
综例 2： 单向链表的类模板

```
#include<iostream.h>
template<class DType>
class List
{
public:
        List(DType d){data=d;next=0;}       // 构造函数
        void Add(List *node){node->next=this;next=0;}
        List*GetNext(){return next;}
        DType GetData(){return data;}
private:
        DType data;
        List *next;
};
void main()
{
    List<char>start('a');      // 定义模板类对象 start
    List<char>*p,*last;      // 定义模板类对象指针 *p,*last
```

综例 2： 单向链表的类模板

```
        last=&start;
        for(int i=1;i<26;i++)
        {
            p=new List<char>('a'+i);
            p->Add(last);
            last=p;
        }
        cout<<endl;
        p=&start;
        while(p)
        {
            cout<<p->GetData();
            p=p->GetNext();
        }
}
```

综例 2： 单向链表的类模板



综例 3：带头结点的双向循环链表

要求：

构造带头结点的双向循环链表，用插入函数以及输出函数，初始化 26 个英文字母。

插入算法的思路：（insert）本算法为结点后插入

1、声明一结点 p 指向头结点，初始化 i 从 0 开始。

2、遍历链表，让指针 p 向后移动，直至移动到要插入元素的结点，即向后移动 k 次。

3、让指针 q 指向 p 的右边一个结点。

4、在系统中新建一个结点 s 。

删除算法的思路：（erase）本算法为删除第 k 个结点

1、声明一结点 p 指向链表第一个结点，初始化 i 从 0 开始。

2、遍历链表，让指针 p 向后移动，一直移动到要删除结点的前一个结点，即向后移动 k-1 次。

3、让指针 q 指向 p 的右边一个结点。

4、执行 q=p->right; p->right=q->right; q->right->left=q; 删除结点。

综例 3： 带头结点的双向循环链表

```
#include <iostream>
using namespace std;
template <typename T>
class List;
template <typename T>
class Node{
friend class List<T>;
private:
T data;
Node<T> *left,*right;
};
template <typename T>
class List{
public:
List();                          // 构造函数
~List();                         // 析构函数
bool empty()const{return header->right==header->left;}// 测试表是否为空
int size()const;                 // 返回表的长度
bool retrieve(int k,T& x)const;      // 返回表位置 k 处的元素 x
```

综例 3： 带头结点的双向循环链表

```
int locate(const T& x)const;          // 返回元素 x 在表中的位置
List<T>& insert(int k,const T& x);      // 在位置 k 处插入元素 x
List<T>& erase(int k);                  // 从位置 k 处删除元素
void print_list()const;                 // 打印表
private:
Node<T> *header;
};
template <typename T>
List<T>::List()
{
Node<T> *p=new Node<T>;
header=p->left=p->right=p;

}
```

综例 3：带头结点的双向循环链表

```
template <typename T>
List<T>::~List()
{
Node<T> *p=0,*q=0;
p=header->right;
while(header->right==header->left){
q=p->right;
header->right=q;
q->left=header;
delete p;
p=q;
}
delete header;
}
```

```
template <typename T>
int List<T>::size()const
{
Node<T> *p=header->right;
int len=0;
while(p!=header){
p=p->right;
++len;
}
return len;
}
```

综例 3： 带头结点的双向循环链表

```
template <typename T>
bool List<T>::retrieve(int k,T& x
)const
{
Node<T> *p=header->right;
int i=0;
while(i<k-1){
p=p->right;
++i;
}
x=p->data;
return true;
}
```

```
template <typename T>
int List<T>::locate(const T& x)c
onst
{
Node<T> *p=header->right;
int i=1;
while((p!=header)){
if (p->data==x) return i;
p=p->right;
++i;
}
return 0;
}
```

综例 3：带头结点的双向循环链表

```
template <typename T>
List<T>& List<T>::insert(int k,const T& x)
{
Node<T> *p=0,*q=0;
p=header;
int i=0;
while(i<k){
p=p->right;
++i;
}
q=p->right;
Node<T> *s=new Node<T>;
s->data=x;
s->right=p->right;
s->left=q->left;
p->right=s;
q->left=s;
return *this;
}
```

综例 3： 带头结点的双向循环链表

```
template <typename T>
List<T>& List<T>::erase(int k)
{
Node<T> *p=0,*q=0;
p=header;
int i=0;
while(i<k-1){
p=p->right;
++i;
}
q=p->right;
p->right=q->right;
q->right->left=q;
delete q;
return *this;
}
```

```
template <typename T>
void List<T>::print_list()const
{
Node<T> *p=header->right;
while(p!=header){
cout <<p->data <<" ";
p=p->right;
}
}
```

综例 3：带头结点的双向循环链表

```
int main()
{
        int s1,s2;
        s1='A';
        s2='Z';
        List<char> p;
        for(int i=s2; i>=s1; --i){
                p.insert(0,i);
        }
        p.print_list();
        return 0;
}
```

04 C++ 标准模板库

1 泛型程序设计

- 泛型：是指将类型参数化以达到代码复用提高软件开发工作效率的一种数据类型。
- 泛型程式设计是一种「将类型参数化」的思维（编程）模式。

- 目的：将程序写得尽可能通用 ，即将适用性较强的算法从特定的数据结构中抽象出来，成为通用的 .

1 泛型程序设计

- 在面向对象的程序设计范型之下，程序员的队伍可能要分为两种，它们都以类作为工作对象；
  - 一个队伍主要是设计类和类库；
  - 另一个队伍主要是使用类来设计应用程序；

- 结构化程序设计时代的标准：每天编写的源代码行数，程序结构清晰。
- 面向对象程序设计时代的标准：
  - 衡量一个应用程序员的生产力，要看他是否知道如何来最好地发挥已有类库的功能，
  - 要看他有没有能力将已有的类库与新问题紧密的匹配起来，
  - 还要看他不得不另外编写的代码是不是最少。

1 泛型程序设计（续）

- 将程序写得尽可能通用；
- 将算法从特定的数据结构中抽象出来，成为通用的；
- C++ 的模板为泛型程序设计奠定了关键的基础 ；
- STL 是泛型程序设计的一个范例
  - 容器 (container)
  - 迭代器 (iterator)
  - 算法（ algorithms ）
  - 函数对象（ function object ）

2. 什么是类库?

- 类库是类的集合，并且给出了多种类之间的关系描述。
- 为了便于程序员的开发工作，系统提供了一批可供重用的代码。 （源程序代码）
- 具体表现为一组类，通过建立彼此间的继承关系形成类库，以类的形式提供给用户重用。
- 在设计和实现面向对象的程序的时候，要用类和类库，才能得到所需的对象，即类的实例。

- 所以，类库是一种预定义的面向对象的程序库。

2. 什么是类库？： 类库的例子

- C++ Standard STL library；
- Microsoft Visual C++ 系统中提供的 MFC 类库；

  MFC: Microsoft Foundation Class
- Borland C++ 系统中提供的 OWL 类库；
- C++ Builder 系统中提供的 VCL 类库；
- 近年很流行的 QT 库；

2. 什么是类库？类库有什么特点

- **通用性**
  - 基于重用的目的，选择具有广泛适用性的东西作为类库的内容，并经过全面的考虑，使之适用于较多的情况；
- 可扩充性
  - 在软件开发过程中，可以添加新的类供以后使用；
  - 对已有的类库进行改进时，只要保持接口不变，修改不会引起外部（即应用系统）软件的变化；
- 概念性与层次性
  - 设计类要有一个明确的目标，类的概念要明确易于理解，基类具有共同性质；
- 灵活性

3. 标准模板库 STL

- STL （Standard Template Library），即标准模板库，是一个具有工业强度的，高效的 C++ 程序库。它被容纳于 C++ 标准程序库（C++ Standard Library）中，是 ANSI/ISO C++ 标准中最新的也是极具革命性的一部分。
- 该库包含了诸多在计算机科学领域里所常用的基本数据结构和基本算法。为广大 C++ 程序员们提供了一个可扩展的应用框架，高度体现了软件的可复用性。
- 类似于 Microsoft Visual C++ 中的 MFC （Microsoft Foundation Class Library），或者是 Borland C++ Builder 中的 VCL(Visual Component Library)；

3. 标准模板库 STL

- 从逻辑层次来看，在 STL 中体现了泛型化程序设计的思想（ generic programming ），引入了诸多新的名词，比如像需求（ requirements ），概念（ concept ），模型（ model ），容器（ container ），算法（ algorithmn ），迭代器（ iterator ）等。

- 从实现层次看，整个 STL 是以一种类型参数化（ type parameterized ）的方式基于模板（ template ）实现的。

3. 标准模板库 STL

- Standard Template Library (STL)
- 包含常用算法和数据结构的通用库
- STL 的核心内容是 3 个基本组件：
  - 容器
  - 算法
  - 迭代器

3. 标准模板库 STL ：容器类

- 容器 (container) 类是用来容纳、包含一组元素或元素集合的对象的， STL 中定义了多种不同类型的容器，例如：
  - 向量 vector
  - 线性表 list
  - 队列 queue
  - 映射 map
  - 集合 set
  - 字符串 string
  - stack: stack
  - associative array: map
- （还有其他容器类，可以参考其他书籍或者查阅联机手册）

3. 标准模板库 STL ：常用算法 <algorithm>

- C++标准模板库中包括 70 多个算法
  - 排序 sort( )
  - 查找 find( )
  - 替换 replace( )
  - 合并 merge( )
  - 反序 reverse( )
  - 统计 count( )
  - 其他等等算法

3. 标准模板库 STL ：迭代器（iterator）

- 迭代器是一种类似于指针的对象。可以使用迭代器来访问容器中的元素，就像我们使用指针来访问数组一样；

- STL 中定义了五种迭代器：
  - 随机访问迭代器　RandIter
  - 双向迭代器　BiIter
  - 前向迭代器　ForIter
  - 输入迭代器　InIter
  - 输出迭代器　OutIter

4. STL 的简单使用方法

- 在实际的 C++ 面向对象程序设计中， STL 库将起着举足轻重的作用。
- STL 是一个非常庞大、复杂的类库
- 目前已经有不少专著介绍 STL

- 我们通过简单的实例介绍最基本的应用方法

05 程序实例

标准模板库 STL 应用举例

- 向量 vector
- 线性表 list
- 队列 queue
- 集合 set
- 映射 map
- 字符串 string

1. 向量 vector

- 向量 vector 类可用来支持动态数组，动态数组是指可以根据需要改变大小的数组。

- 可以很容易地声明一个 vector 类对象，例如：

```
vector <int> iv;
vector <int> cv(5);
vector <int> cv(5,'x');
vector <int> iv2(iv);
```

例 1. 向量 vector 应用实例

```
// Access a vector using an iterator.
#include <iostream>
#include <vector>
using namespace std;
int main( )
{
       vector<char> v; // create zero-length vector
       int i;

  // put values into a vector
      for(i=0; i<10; i++)
           v.push_back('A' + i);
```

例 1. 向量 vector 应用实例

```
// can access vector contents using subscripting
      for(i=0; i<10; i++) cout << v[i] << " ";
            cout << endl;

  // access via iterator
     vector<char>::iterator p = v.begin( );
      while(p != v.end())
     {
              cout << *p << " ";
              p++;
      }
      return 0;
}
```

例 2. 排序算法 sort 实例

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void load(vector<string>&);
void print(vector<string>);
const int SIZE=8;
int main()
{
    vector<string> v(SIZE);
    load(v);
    sort(v.begin(),v.end()); // 排序（利用标准库中的模板 sort( ) 。）
    print(v);
    return 0;
}
```

例 2. 排序算法 sort 实例

```
void load(vector<string>& v)
{
    v[0] = "Japan";
    v[1] = "Italy";
    v[2] = "Spain";
    v[3] = "Egypt";
    v[4] = "Chile";
    v[5] = "Zaire";
    v[6] = "Nepal";
    v[7] = "Kenya";
}
void print(vector<string> v)
{
    for (int i=0; i<SIZE; i++)
        cout << v[i] << endl;
    cout << endl;
}
```

输出结果如下：

```
Chile
Egypt
Italy
Japan
Kenya
Nepal
Spain
Zaire
```

2. 线性表 list

- 线性表 list 类定义了双向的线性表， 又可称为双向链表。 List 类只支持顺序访问。

- 下面的 C++ 程序通过实例化链表 list 类模板建立了一个保存字符的链表，接着使用类模板的排序方法 sort( ) 进行排序，然后输出经过排序后的字符。

例 3. 线性表 list 应用实例

```
// Sort a list.
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
   int i;
   list<char> lst;
  // create a list of random characters
   for(i=0; i<10; i++)
        lst.push_back('A'+ (rand()%26));
```

例 3. 线性表 list 应用实例

```
cout << "Original contents: ";
list<char>::iterator p = lst.begin();
while(p != lst.end())
{
    cout << *p;
    p++;
}
cout << endl << endl;

// sort the list
lst.sort( );
```

例 3. 线性表 list 应用实例

```
    cout << "Sorted contents: ";
    p = lst.begin();
    while(p != lst.end())
   {
        cout << *p;
        p++;
    }
    return 0;
}
```

例 4. 反向迭代器 reverse iterator 实例

```
#include <list>
#include <iostream>
int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;
   list <int>::reverse_iterator c1_rIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
```

例 4. 反向迭代器 reverse iterator 实例

```
c1_rIter = c1.rbegin( );
cout << "The last element in the list is " << *c1_rIter << "." << endl;

cout << "The list is:";
for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
   cout << " " << *c1_Iter;
cout << endl;
```

例 4. 反向迭代器 reverse iterator 实例

```
// rbegin can be used to start an iteration through a list
// in reverse order
     cout << "The reversed list is:";
     for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
          cout << " " << *c1_rIter;
     cout << endl;
     c1_rIter = c1.rbegin( );
     *c1_rIter = 40;
      cout << "The last element in the list is now " << *c1_rIter << "." << endl;
}
```

例 4. 反向迭代器 reverse iterator 实例

运行输出结果

```
The last element in the list is 30.
The list is: 10 20 30
The reversed list is: 30 20 10
The last element in the list is now 40.
```

3. 集合 set

- set 是关联容器的一种，是排序好的集合（元素已经进行了排序）。set 和 multiset 类似，它和 multiset 的差别在于 set 中不能有重复的元素。multiset 的成员函数 set 中也都有。
- 不能直接修改 set 容器中元素的值。因为元素被修改后，容器并不会自动重新调整顺序，于是容器的有序性就会被破坏，再在其上进行查找等操作就会得到错误的结果。因此，如果要修改 set 容器中某个元素的值，正确的做法是先删除该元素，再插入新元素。
- 使用 set 必须包含头文件 <set>。set 的定义如下：

```
template < class Key, class Pred = less<Key>, class A = allocator<Key> >
class set {...}
```

- set 中插入单个元素的 insert 成员函数与 multiset 中的有所不同，其原型如下：

```
pair<iterator, bool> insert(const T & val);
```

- 关联容器的 equal_range 成员函数的返回值也是 pair 模板类对象，其原型如下：

```
pair<iterator, iterator> equal_range(const T & val);
```

例 5. 集合 set 用法

```
#include <iostream>
#include <set> // 使用 set 须包含此文件
using namespace std;
int main()
{
typedef set<int>::iterator IT;
int a[5] = { 3,4,6,1,2 };
set<int> st(a,a+5); // st 里是  1 2 3 4 6
pair< IT,bool> result;
result = st.insert(5); // st 变成  1 2 3 4 5 6
if(result.second) // 插入成功则输出被插入元素
cout << * result.first << " inserted" << endl; // 输出 : 5 inserted
if(st.insert(5).second)
cout << * result.first << endl;
else
cout << * result.first << " already exists" << endl;
// 输出  5 already exists
pair<IT,IT> bounds = st.equal_range(4);
cout << * bounds.first << "," << * bounds.second ; // 输出：  4,5
return 0;
}
```

程序的输出结果是：

```
5 inserted
5 already exists
4,5
```

例 6. 集合 set 实例

```
#include <set>
#include <iostream>
#include <string>
int main()
{
    std::set<std::string> source;
    std::string input;
    for (int i=0;i<6;i++)
    {
            std::cin >> input;
            source.insert(input);
    }
    std::set<std::string>::iterator at = source.begin();
    while(at != source.end())
            std::cout << *at++ << std::endl;
}
```

例 7. 集合 set 实例

```
#include <cassert>
#include <iostream>
#include <set>
using namespace std;
int main (int argc, char* argv[])
{
    set< int > iset; // set of unique integer numbers
    iset.insert( 11 ); // populate set with some values
    iset.insert( -11 );
    iset.insert( 55 );
    iset.insert( 22 );
    iset.insert( 22 );
   if ( iset.find( 55 ) != iset.end() )
    { // is value already stored?
          iset.insert( 55 );
     }
      assert( iset.size() == 4 ); // sanity check :-)
       set< int >::iterator it;
        for ( it = iset.begin(); it != iset.end(); it++ )
             cout << " " << *it;
         return 0;
}                              // Output: -11 11 22 55
```

4. 集合 multiset

- multiset 是关联容器的一种，是排序好的集合（元素已经进行了排序），并且允许有相同的元素。
- 不能直接修改 multiset 容器中元素的值。因为元素被修改后，容器并不会自动重新调整顺序，于是容器的有序性就会被破坏，再在其上进行查找等操作就会得到错误的结果。因此，如果要修改 multiset 容器中某个元素的值，正确的做法是先删除该元素，再插入新元素。
- 使用 multiset 必须包含头文件 <set> 。 multiset 类模板的定义如下：

```
template <class Key, class Pred = less<Key>, class B = allocator<Key> >
    class multiset {
        ...
    };
```

- 第一个类型参数说明 multiset 容器中的每个元素都是 Key 类型的。第二个类型参数 Pred 用于指明容器中元素的排序规则，在被实例化后， Pred 可以是函数对象类，也可以是函数指针类型。

例 8. 集合 multiset 实例

```
#include <set>
#include <iostream>
#include <string>
int main()
{
	std::multiset<std::string> source;
	std::string input;
	for (int i=0;i<6;i++)
	{
			std::cin >> input;
			source.insert(input);
	}

	std::multiset<std::string>::iterator at = source.begin( );
	while(at != source.end())
			std::cout << *at++ << std::endl;
}
```

例 9. 集合 multiset 实例

```
#include <cassert>
#include <iostream>
#include <set>
using namespace std;
int main (int argc, char* argv[])
{
    set< int > iset; // set of unique integer numbers
    iset.insert( 11 ); // populate set with some values
    iset.insert( -11 );
    iset.insert( 55 );
    iset.insert( 22 );
    iset.insert( 22 );
   if ( iset.find( 55 ) != iset.end() )
    { // is value already stored?
          iset.insert( 55 );
     }
      assert( iset.size() == 4 ); // sanity check :-)
       set< int >::iterator it;
        for ( it = iset.begin(); it != iset.end(); it++ )
             cout << " " << *it;
         return 0;
}                              // Output: -11 11 22 55
```

例 10. 集合 multiset 实例

```
#include <iostream>
#include <set> // 使用 multiset 须包含此头文件
using namespace std;
template <class T>
void Print(T first, T last)
{
for (; first != last; ++first)
cout << *first << " ";
cout << endl;
}
class A
{
private:
int n;
public:
A(int n_) { n = n_; }
friend bool operator < (const A & a1, const A & a2)
{ return a1.n < a2.n; }
friend ostream & operator << (ostream & o, const A & a2)
{ o << a2.n; return o; }
friend class MyLess;
};
class MyLess
{
public:
bool operator() (const A & a1, const A & a2) // 按个位数比较大小
{ return (a1.n % 10) < (a2.n % 10); }
};
typedef multiset <A> MSET1; //MSET1 用“ <” 运算符比较大小
typedef multiset <A, MyLess> MSET2; //MSET2 用 MyLess::operator() 比较大小
int main()
{
const int SIZE = 6;
A a[SIZE] = { 4, 22, 19, 8, 33, 40 };
MSET1 m1;
m1.insert(a, a + SIZE);
m1.insert(22);
cout << "1)" << m1.count(22) << endl; // 输出 1)2
cout << "2)"; Print(m1.begin(), m1.end()); // 输出 2)4 8 19 22 22 33 40
MSET1::iterator pp = m1.find(19);
if (pp != m1.end()) // 条件为真说明找到
cout << "found" << endl; // 本行会被执行，输出 found
cout << "3)"; cout << *m1.lower_bound(22)
<< "," << *m1.upper_bound(22) << endl; // 输出 3)22,33
pp = m1.erase(m1.lower_bound(22), m1.upper_bound(22));
//pp 指向被删元素的下一个元素
cout << "4)"; Print(m1.begin(), m1.end()); // 输出 4)4 8 19 33 40
cout << "5)"; cout << *pp << endl; // 输出 5)33
MSET2 m2; //m2 中的元素按 n 的个位数从小到大排序
m2.insert(a, a + SIZE);
cout << "6)"; Print(m2.begin(), m2.end()); // 输出 6)40 22 33 4 8 19
return 0;
}
```

5. 映射 map, multimap

- 映射 map 类定义了一个关联容器，并且在容器中使用唯一的关键字来映射相应的值。
- map 类对象是一系列关键字 / 值的匹配对。
- map 的功能在于：只要知道了一个值的关键字，就可以找到这个值。容器中的元素是按关键字排序的，并且不允许有多个元素的关键字相同。
- 要使用 map，必须包含头文件 <map>。map 的定义如下：

```
template < class Key, class T, class Pred = less<Key>, class A = allocator<T> >
class map{
   ...
   typedef pair< const Key, T > value_type;
   ...
};
```

- 下面的实例程序通过实例化标准库中的 map 类模板映射建立了一些英文单词与其反义词的对应关系，利用这种对应系可以迅速查找到一个词的反义词。

例 11. 映射 map 类实例

```
// A map of word opposites, using strings.
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main( )
{
   int i;
   map<string, string> m;

   m.insert(pair<string, string>("yes", "no"));
   m.insert(pair<string, string>("up", "down"));
   m.insert(pair<string, string>("left", "right"));
   m.insert(pair<string, string>("good", "bad"));
   string s;
   cout << "Enter word: ";
   cin >> s;

   map<string, string>::iterator p;

   p = m.find(s);
   if(p != m.end())
        cout << "Opposite: " << p->second;
   else
        cout << "Word not in map.\n";

   return 0;
}
```

例 12. 映射 map 类实例

```
#include <iostream>
#include <map> // 使用 map 需要包含此头文件
using namespace std;
template <class T1,class T2>
ostream & operator <<(ostream & o,const pair<T1,T2> & p)
{ // 将 pair 对象输出为 (first,second) 形式
o << "(" << p.first << "," << p.second << ")";
return o;
}
template<class T>
void Print(T first,T last)
{// 打印区间 [first,last)
for( ; first != last; ++ first)
cout << * first << " ";
cout << endl;
}
typedef map<int,double,greater<int> > MYMAP;
// 此容器关键字是整型，元素按关键字从大到小排序
int main()
{
MYMAP mp;
mp.insert(MYMAP::value_type(15,2.7));
pair<MYMAP::iterator,bool> p = mp.insert(make_pair(15,99.3));
if(!p.second)
cout << * (p.first) << " already exists" << endl; // 会输出
cout << "1) " << mp.count(15) << endl; // 输出 1) 1
mp.insert(make_pair(20,9.3));
cout << "2) " << mp[40] << endl;// 如果没有关键字为 40 的元素，则插入一个
cout << "3) ";Print(mp.begin(),mp.end());// 输出： 3) (40,0)(20,9.3)(15,2.7)
mp[15] = 6.28; // 把关键字为 15 的元素值改成 6.28
mp[17] = 3.14; // 插入关键字为 17 的元素，并将其值设为 3.14
cout << "4) ";Print(mp.begin(),mp.end());
return 0;
}
```

程序的输出结果如下：

```
(15,2.7) already exists
1) 1
2) 0
3) (40,0) (20,9.3) (15,2.7)
4) (40,0) (20,9.3) (17,3.14) (15,6.28)
```

5. 映射 map, multimap

- multimap 是关联容器的一种， multimap 的每个元素都分为关键字和值两部分，容器中的元素是按关键字排序的，并且允许有多个元素的关键字相同。
- 注意，不能直接修改 multimap 容器中的关键字。因为 multimap 中的元素是按照关键字排序的，当关键字被修改后，容器并不会自动重新调整顺序，于是容器的有序性就会被破坏，再在其上进行查找等操作就会得到错误的结果。
- 使用 multimap 必须包含头文件 map 。 multimap 的定义如下：

```
template < class Key, class T, class Pred = less<Key>, class A = allocator<T> >
class multimap
{
   ...
   typedef pair <const Key, T> value_type;
   ...
};
```

multimap 中的元素都是 pair 模板类的对象。元素的 first 成员变量也叫“关键字”， second 成员变量也叫“值”。 multimap 容器中的元素是按关键字从小到大排序的。默认情况下，元素的关键之间用 less <Key> 比较大小 . multimap 允许多个元素的关键字相同。

例 13.multimap 类实例

```
#include <iostream>
#include <map> // 使用 multimap 需要包含此头文件
#include <string>
using namespace std;
class CStudent
{
public:
struct CInfo // 类的内部还可以定义类
{
int id;
string name;
};
int score;
CInfo info; // 学生的其他信息
};
typedef multimap <int, CStudent::CInfo> MAP_STD;
int main()
{
MAP_STD mp;
CStudent st;
string cmd;
while (cin >> cmd) {
if (cmd == "Add") {
cin >> st.info.name >> st.info.id >> st.score;
mp.insert(MAP_STD::value_type(st.score, st.info));
}
else if (cmd == "Query") {
int score;
cin >> score;
MAP_STD::iterator p = mp.lower_bound(score);
if (p != mp.begin()) {
--p;
score = p->first; // 比要查询分数低的最高分
MAP_STD::iterator maxp = p;
int maxId = p->second.id;
for (; p != mp.begin() && p->first == score; --p) {
// 遍历所有成绩和 score 相等的学生
if (p->second.id > maxId) {
maxp = p;
maxId = p->second.id;
}
}
if (p->first == score) { // 如果上面的循环因为 p == mp.begin()
// 而终止，则 p 指向的元素还要处理
if (p->second.id > maxId) {
maxp = p;
maxId = p->second.id;
}
}
cout << maxp->second.name << " " << maxp->second.id << " "
<< maxp->first << endl;
}
else //lower_bound 的结果就是 begin，说明没有分数比查询分数低
cout << "Nobody" << endl;
}
}
return 0;
}
```

6. 队列 queue 和 priority_queue

- **队列 (queue) 是一个先进先出 (FIFO: First In First Out) 的数据结构，在程序设计中经常使用。** queue 可以用 list 和 deque 实现，默认情况下用 deque 实现。
- queue 的定义如下：

```
template < class T, class Cont = deque<T> >
class queue{
   ...
};
```

queue 同样也有和 stack 类似的 push 、 pop 、 top 函数。区别在于， queue 的 push 发生在队尾， pop 和 top 发生在队头。

- 对一个队列常用的操作有，在队列尾增加一个元素、在队列头取一个元素以及测试队列是否为空、是否为满等操作。

6. 队列 queue 和 priority_queue

- priority_queue 是“优先队列”。它和普通队列的区别在于，优先队列的队头元素总是最大的——即执行 pop 操作时，删除的总是最大的元素；执行 top 操作时，返回的是最大元素的引用。
- priority_queue 的定义如下：

```
template < class T, class Container = vector <T>, class Compare = less<T> >
class priority_queue{
   …
};
```

priority_queue 的第三个类型参数可以用来指定排序规则。

- priority_queue 是使用“堆排序”技术实现的，其内部并非完全有序，但却能确保最大元素总在队头。因此， priority_queue 特别适用于“不停地在一堆元素中取走最大的元素”这种情况。 priority_queue 插入和删除元素的复杂度都是 O(log(n)) 。虽然用 set/multiset 也能完成此项工作，但是 priority_queue 比它们略快一些。

6. 队列　queue 和 priority_queue

Using queue class in the Standard C++ Library, Instantiate a queue for strings and demonstrate the following functions in main( ) to show that you know how to use this class:

queue::push( )

queue::pop( )

queue::empty( )

queue::front( )

queue::back( )

queue::size( )

例 14. 队列 queue 的应用实例

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;
void main( )
{
    queue<string> str_queue;
    str_queue.push("string1");
    str_queue.push("string2");
    str_queue.push("string3");
     cout<<"the size of the queue is "<<str_queue.size()<<endl;
     cout<<"the front one "<<str_queue.front()<<endl;
     cout<<"the back one "<<str_queue.back()<<endl;
     str_queue.pop( );
     str_queue.pop( );
     str_queue.pop( );
     if (str_queue.empty( ))
          cout<<" the queue is empty!"<<endl;
}
```

例 15. 队列 priority_queue 的应用实例

```
#include <queue>
#include <iostream>
using namespace std;
int main()
{
    priority_queue<double> pq1;
    pq1.push(3.2);
    pq1.push(9.8);
    pq1.push(9.8);
    pq1.push(5.4);
    while(!pq1.empty()) {
        cout << pq1.top() << " ";
        pq1.pop();
    } // 上面输出  9.8 9.8 5.4 3.2
    cout << endl;
     priority_queue<double,vector<double>,greater<double> > pq2;
     pq2.push(3.2);
     pq2.push(9.8);
     pq2.push(9.8);
     pq2.push(5.4);
     while(!pq2.empty()) {
          cout << pq2.top() << " ";
          pq2.pop();
     }
 // 上面输出  3.2 5.4 9.8 9.8
     return 0;
}
```

程序的输出结果是：

9.8 9.8 5.4 3.2

3.2 5.4 9.8 9.8

7. stack

- stack 是容器适配器的一种。要使用 stack ，必须包含头文件 <stack> 。
- stack 就是“栈”。栈是一种后进先出的元素序列，访问和删除都只能对栈顶的元素（即最后一个被加入栈的元素）进行，并且元素也只能被添加到栈顶。栈内的元素不能访问。如果一定要访问栈内的元素，只能将其上方的元素全部从栈中删除，使之变成栈顶元素才可以。
- stack 的定义如下：

```
template < class T, class Cont == deque <T> >
class stack{
    ...
};
```

- 虽然 stack 使用顺序容器实现，但它不提供顺序容器具有的成员函数。除了 size 、empty 这两个所有容器都有的成员函数外， stack 还有以下三个成员函数

void pop();	弹出（即删除）栈顶元素
T & top();	返回栈顶元素的引用。通过此函数可以读取栈顶元素的值，也可以修改栈顶元素
void push (const T & x);	将 x 压入栈顶

例 16. stack 的应用实例

```
//A stack is a container that permits to insert and extract its elements only from the top of the container:
#include <cassert>
 #include <stack>
using namespace std;
 int main (int argc, char* argv[])
{
      stack< int > st;
      st.push( 100 );                  // push number on the stack
      assert(   st.size() == 1 );      // verify one element is on the stack
      assert( st.top() == 100 );       // verify element value
      st.top() = 456;                  // assign new value
      assert( st.top() == 456 );
      st.pop();                                   //  remove element
      assert( st.empty() == true );
      return 0;
}
```

例 17. stack 的应用实例

```
#include <iostream>
#include <stack> // 使用 stack 需要包含此头文件
using namespace std;
int main()
{
	int n, k;
	stack <int> stk;
	cin >> n >> k; // 将 n 转换为 k 进制数
	if (n == 0) {
		cout << 0;
		return 0;
	}
	while (n) {
		stk.push(n%k);
		n /= k;
	 }
	 while (!stk.empty()) {
		cout << stk.top();
		stk.pop();
	}
	 return 0;
}
```

8. std::pair 类

因为关联容器的一些成员函数的返回值是 pair 对象，而且 map 和 multimap 容器中的元素都是 pair 对象。 pair 的定义如下：

```
template <class_Tl, class_T2>
struct pair
{
        _T1 first;
        _T2 second;
        pair(): first(), second() {} // 用无参构造函数初始化 first 和 second
        pair(const _T1 &__a, const _T2 &__b): first(__a), second(__b) {}
        template <class_U1, class_U2>
        pair(const pair <_U1, _U2> &__p): first(__p.first), second(__p.second) {}
};
```

pair 实例化出来的类都有两个成员变量，一个是 first，一个是 second 。

8. std::pair 类

STL 中还有一个函数模板 make_pair ，其功能是生成一个 pair 模板类对象。 make_pair 的源代码如下：

```
template <class T1, class T2>
pair<T1, T2 > make_pair(T1 x, T2 y)
{
return ( pair<T1, T2> (x, y) );
}
```

例 18. pair 的应用实例

```
#include <cassert>
#include <string>
#include <utility>
using namespace std;
int main (int argc, char* argv[])
{
        pair< string, string > strstr;
        strstr.first = "Hello";
        strstr.second = "World";
        pair< int, string > intstr;
        intstr.first = 1;
        intstr.second = "one";
        pair< string, int > strint( "two", 2 );
        assert( strint.first == "two" );
        assert( strint.second == 2 );
        return 0;
}
```

例 19. pair 和 make_pair 的用法

```
#include <iostream>
using namespace std;
int main()
{
        pair<int,double> p1;
        cout << p1.first << "," << p1.second << endl; // 输出  0,0
        pair<string,int> p2("this",20);
        cout << p2.first << "," << p2.second << endl; // 输出  this,20
        pair<int,int> p3(pair<char,char>('a','b'));
        cout << p3.first << "," << p3.second << endl; // 输出  97,98
        pair<int,string> p4 = make_pair(200,"hello");
        cout << p4.first << "," << p4.second << endl; // 输出  200,hello
return 0;
}
```

9. bitset 类

- bitset 模板类由若干个位（bit）组成，它提供一些成员函数，使程序员不必通过位运算就能很方便地访问、修改其中的任意一位。bitset 模板类在头文件 <bitset> 中定义如下：

```
template <size_t N>
class bitset
{
   …
};
```

- size_t 可看作 unsigned int 。将 bitset 实例化时，N 必须是一个整型常数。

9. bitset 类

- bitset 有许多成员函数，有些成员函数执行的就是类似于位运算的操作。 bitset 成员函数列表如下：

- bitset <N> & operator &= (const bitset <N> & rhs); // 和另一个 bitset 对象进行与操作
- bitset <N> & operator |= (const bitset <N> & rhs); // 和另一个 bitset 对象进行或操作
- bitset <N> & operator ^= (const bitset <N> & rhs); // 和另一个 bitset 对象进行异或操作
- bitset <N> & operator <<= (size_t num); // 左移 num 位
- bitset <N> & operator >>= (size_t num); // 右移 num 位
- bitset <N> & set(); // 将所有位全部设成 1
- bitset <N> & set(size_t pos, bool val = true); // 将第 pos 位设为 val
- bitset <N> & reset(); // 将所有位全部设成 0
- bitset <N> & reset (size_t pos); // 将第 pos 位设成 0
- bitset <N> & flip(); // 将所有位翻转（0 变成 1， 1 变成 0）
- bitset <N> & flip(size_t pos); // 翻转第 pos 位
- reference operator[] (size_t pos); // 返回对第 pos 位的引用
- bool operator[] (size_t pos) const; // 返回第 pos 位的值
- reference at(size_t pos); // 返回对第 pos 位的引用
- bool at (size_t pos) const; // 返回第 pos 位的值
- unsigned long to_ulong() const; // 将对象中的 0 、 1 串转换成整数
- string to_string () const; // 将对象中的 0 、 1 串转换成字符串（Visual Studio 支持， Dev C++ 不支持）
- size_t count() const; // 计算 1 的个数
- size_t size () const; // 返回总位数
- bool operator == (const bitset <N> & rhs) const;
- bool operator != (const bitset <N> & rhs) const;
- bool test(size_t pos) const; // 测试第 pos 位是否为 1
- bool any() const; // 判断是否有某位为 1
- bool none() const; // 判断是否全部为 0
- bitset <N> operator << (size_t pos) const; // 返回左移 pos 位后的结果
- bitset <N> operator >> (size_t pos) const; // 返回右移 pos 位后的结果
- bitset <N> operator ~ (); // 返回取反后的结果
- bitset <N> operator & (const bitset <N> & rhs) const; // 返回和另一个 bitset 对象 rhs 进行与运算的结果
- bitset <N> operator | (const bitset <N> & rhs) const; // 返回和另一个 bitset 对象 rhs 进行或运算的结果
- bitset <N> operator ^ (const bitset <N> & rhs) const; // 返回和另一个 bitset 对象 rhs 进行异或运算的结果

例 20 bitset 的用法

```
#include <iostream>
#include <bitset>
#include <string>
using namespace std;
int main()
{
bitset<7> bst1;
bitset<7> bst2;
cout << "1) " << bst1 << endl; // 输出 1) 0000000
bst1.set(0,1);// 将第 0 位变成 1， bst1 变为 0000001
cout << "2) " << bst1 << endl; // 输出 2) 0000001
bst1 <<= 4; // 左移 4 位，变为 0010000
cout << "3) " << bst1 << endl; // 输出 3) 0010000
bst2.set(2);// 第二位设置为 1， bst2 变成 0000100
bst2 |=bst1; // bst2 变成 0010100
cout << "4) " << bst2 << endl; // 输出 4) 0010100
cout << "5) " << bst2.to_ulong () << endl; // 输出 5) 20
bst2.flip(); // 每一位都取反， bst2 变成 1101011
bst1.set(3); //bst1 变成 0011000
bst2.flip(6); //bst2 变成 0101011
bitset<7> bst3 = bst2^ bst1;//bst3 变成 0110011
cout << "6) " << bst3 << endl; // 输出 6) 0110011
cout << "7) " << bst3[3] << "," << bst3[4] << endl; // 输出 7) 0,1
return 0;
}
```

10. 字符串类 string

- C++ 提供了两种处理字符串的方法：
  - 以空字符‘ \0’ 结尾的字符数组
  - 容器类 string 类的对象（标准库中的 string 类）

- 使用标准库中的 string 类的三个理由：
  - 一致性（字符串定义为一种数据类型）
  - 方便性（可以使用标准的运算符）
  - 安全性（不会出现数组越界错误）

例 21. 字符串类应用实例

```
// Demonstrate insert(), erase(), and replace().
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1("This is a test");
  string str2("ABCDEFG");

  cout << "Initial strings:\n";
  cout << "str1: " << str1 << endl;
  cout << "str2: " << str2 << "\n\n";
```

例 21. 字符串类应用实例

```
// demonstrate insert()
  cout << "Insert str2 into str1:\n";
  str1.insert(5, str2);
  cout << str1 << "\n\n";

  // demonstrate erase()
  cout << "Remove 7 characters from str1:\n";
  str1.erase(5, 7);
  cout << str1 <<"\n\n";
```

例 21. 字符串类应用实例

```
// demonstrate replace
    cout << "Replace 2 characters in str1 with str2:\n";
    str1.replace(5, 2, str2);
    cout << str1 << endl;

    return 0;
}
```

例 21. 字符串类应用实例

程序运行后的输出信息

```
Initial strings:
str1: This is a test
str2: ABCDEFG

Insert str2 into str1:
This ABCDEFGis a test

Remove 7 characters from str1:
This is a test

Replace 2 characters in str1 with str2:
This ABCDEFG a test
```

例 22. String 的反向迭代器应用实例

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
	cout << "Please input the string : ";
	string str;
	cin>>str;
	cout << "Reverse browse : ";
	for (string::reverse_iterator rit = str.rbegin(); rit != str.rend(); ++ rit)
	{
		cout << *rit;
	}
	cout << endl;
	return 0;
}
```

例 23. 查找成员函数的示例程序

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
string s1("Source Code");
int n;
if ((n = s1.find('u')) != string::npos) // 查找  u  出现的位置
cout << "1) " << n << "," << s1.substr(n) << endl;
// 输出  l)2,urce Code
if ((n = s1.find("Source", 3)) == string::npos)
// 从下标 3 开始查找 "Source" ，找不到
cout << "2) " << "Not Found" << endl; // 输出  2) Not Found
if ((n = s1.find("Co")) != string::npos)
// 查找子串 "Co" 。能找到，返回 "Co" 的位置
cout << "3) " << n << ", " << s1.substr(n) << endl;
// 输出  3) 7, Code
if ((n = s1.find_first_of("ceo")) != string::npos)
// 查找第一次出现或  'c' 、 'e' 或 'o' 的位置
cout << "4) " << n << ", " << s1.substr(n) << endl;
// 输出  4) l, ource Code
if ((n = s1.find_last_of('e')) != string::npos)
// 查找最后一个  'e'  的位置
cout << "5) " << n << ", " << s1.substr(n) << endl; // 输出  5) 10, e
if ((n = s1.find_first_not_of("eou", 1)) != string::npos)
// 从下标 1 开始查找第一次出现非  'e' 、 'o'  或  'u'  字符的位置
cout << "6) " << n << ", " << s1.substr(n) << endl;
// 输出  6) 3, rce Code
return 0;
}
```

例 24. string 对象作为流处理

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main()
{
        string src("Avatar 123 5.2 Titanic K");
        istringstream istrStream(src); // 建立 src 到 istrStream 的联系
        string s1, s2;
        int n;
        double d;
        char c;
        istrStream >> s1 >> n >> d >> s2 >> c; // 把 src 的内容当做输入流进行读取
        ostringstream ostrStream;
        ostrStream << s1 << endl << s2 << endl << n << endl << d << endl << c <<endl;
        cout << ostrStream.str();
        return 0;
}
```

程序的输出结果是：

Avatar
Titanic
123
5.2
K

例 25. 用 STL 算法操作 string 对象

```
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
int main()
{
string s("afgcbed");
string::iterator p = find(s.begin(), s.end(), 'c');
if (p!= s.end())
cout << p - s.begin() << endl; // 输出  3
sort(s.begin(), s.end());
cout << s << endl; // 输出  abcdefg
next_permutation(s.begin(), s.end());
cout << s << endl; // 输出  abcdegf
return 0;
}
```

THANKS !