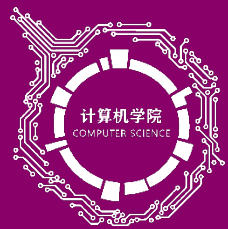




# 第九章 模板与STL程序设计

主讲： 刘晓光 张海威  
张 莹 殷爱茹  
沈 玮 宋春瑶  
李雨森





函数模板



类模板的基本概念



类模板的继承和派生



类模板综合示例



标准模板库程序设计



## 函数模板



## 类模板的基本概念



## 类模板的继承和派生



## 类模板综合示例



## 标准模板库程序设计

# 函数模板

通常设计的算法（处理语句）是可以处理多种数据类型的，但目前处理相同的问题，仍要分别定义多个类似的函数

- 【例如】

```
int max (int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
}
```

```
double max (double a, double b)
{
    if (a>b)
        return a;
    else
        return b;
}
```

```
char max (char a, char b)
{
    ...
}
```

# 函数模板

实际上，若“提取”出一个可变化的类型参数T，则可“综合”成为如下的同一个函数（即函数模板），它实际上代表着一组函数

```
T max (T a, T b) {  
    if (a>b)  
        return a;  
    else  
        return b;  
}
```

# 函数模板

在C++中定义完整的函数模板max时，格式如下

```
template <typename T>
T max (T a, T b) {
    if (a>b)
        return a;
    else
        return b;
}
```

# 函数模板

函数模板定义的一般格式为：

**template** <[类型参数和非类型参数表] >

**返回类型**    **函数模板名 (函数模板形参表)**    { **函数体** }

- 类型参数表

**typename** 类型形参名1 , ... , **typename** 类型形参名n

- 非类型参数表：普通形参表

- 注意：

- 应在函数模板的“返回类型”或“形参表”或“函数体”中使用上述的“类型形参名”。
- 调用处则类似于一般函数，用户只需给出具体的实参。
- 模板函数调用时，不进行实参到形参类型的自动转换。
- 从物理意义上，函数模板类似于重载



# 函数模板

【例9.1】 定义一个函数模板max，而后对它进行不同的调用

```
#include <iostream>
using namespace std;
template <typename T>
T max (T a, T b) {
    if (a>b)
        return a;
    else
        return b;
}
```

```
void main() {  
    int i1=-11, i2=0;  
    double d1, d2;  
    cout<<max(i1,i2)<<endl;  
    //由实参i1, i2, 系统可确定 “类型形参T”对应于int  
    cout<<max(23,-56)<<endl;  
    cout<<max('f', 'k')<<endl;  
    cin>>d1>>d2;  
    cout<<max(d1,d2)<<endl;  
    //cout<<"max(23,-5.6)  ="<<max(23,-5.6)<<endl;  
    //出错！ 不进行实参到形参类型的自动转换  
}
```

程序执行后的显示结果如下：

```
int i1=-11, i2=0; => max(i1,i2) = 0
```

```
max(23,-56) = 23
```

```
char c1='T', c2='F'; => max(c1,c2) = T
```

```
max('f', 'k') = k
```

```
input double d1, d2 : 123.45 99.67
```

```
d1=123.45, d2=99.67 => max(d1,d2) = 123.45
```

# 函数模板的特例

函数体对于参数的操作，无法支持全部数据类型，例如：

- 自定义类型数据的输出
  - cout
- 自定义数据类型的比较
  - 关系运算

【例9.2】定义一个函数模板和该函数模板的特例，函数模板名都是add，作为特例的函数模板，用来处理char\*类型的相加。特例版本定义的格式：

`template<>`带有实例化类型参数的函数原型

必须提前定义好相关的函数模板，才能定义其特例版本。另外，也可以定义一个普通函数，用来连接char\*类型的数据。C++允许函数模板与函数同名的所谓重载使用方法。但注意，在这种情况下，每当遇见函数调用时，C++编译器都将首先检查是否存在重载函数，若匹配成功则调用该函数，否则再去匹配函数模板

```
#include <iostream>
#include <string>
using namespace std;
template <typename type>
type add(type a, type b) {
    return a + b;
}
template <>
char* add <char*>(char* a, char* b) {
    cout << "call add template " << endl;
    return strcat(a,b);
}
```

```
char* add(char* a, char* b) {  
    cout << "call add function " << endl;  
    return strcat(a,b);  
}  
  
int main() {  
    cout << add(3, -10) << endl; //使用函数模板  
    cout << add(2.5, 99.5) << endl;  
    cout << add('m', 'c') << endl;  
    char str1[40] = "The C program ", str2[20] =  
        "The C++ program ";  
    cout << add(str1, str2) << endl;  
    //使用函数!如果去掉这个函数,则使用特例版本  
}
```

# 函数模板的重载

【例9.3】定义两个函数模板，它们都叫做sum，都使用了一个类型参数Type，但两者的形参个数不同，C++允许使用这种函数模板重载的方法。

- 注意，参数表中允许出现与类型形参Type无关的其它类型的参数，如“int size”。



```
#include <iostream>
using namespace std;

template <typename Type>
Type sum (Type * array, int size ) {
    //求array数组前size个元素之和
    Type total=0;
    for (int i=0;i<size;i++)
        total+=*(array+i);
    return total;
}
```

```
template <typename Type>
Type sum (Type * a1, Type * a2, int size ) {
    //求a1数组与a2数组前size个元素之和
    Type total=0;
    for (int i=0;i<size;i++)
        total+=a1[i]+a2[i];
    return total;
}
```

```
void main() {  
    int a1[10],a2[8];  
    float af[10];  
    ... //为数组分量定值  
    cout<<sum(a1,10)<<endl;  
    //求出a1数组前10个元素之和并输出  
    cout<<sum(af,10)<<endl;  
    cout<<sum(a1,a2,8)<<endl;  
    //求a1与a2数组前8个元素之和并输出  
}
```

```
#include <iostream>
#include <string>
using namespace std;
template<typename T> T larger(T a, T b); // Function template prototype
int main()
{
    cout << "Larger of 1.5 and 2.5 is " << larger(1.5, 2.5) << endl;
    cout << "Larger of 3.5 and 4.5 is " << larger(3.5, 4.5) << endl;

    int big_int{ 17011983 }, small_int{ 10 };
    cout << "Larger of " << big_int << " and " << small_int << " is "
         << larger(big_int, small_int) << endl;

    string a_string{ "A" }, z_string{ "Z" };
    cout << "Larger of \" << a_string << \" and \" << z_string << \"
is \" << '\"' << larger(a_string, z_string) << '\"' << endl;

    //add some codes here for further discussion
}
```

```
// Template for functions to return the larger of  
two values  
template <typename T>  
T larger(T a, T b)  
{  
    return a > b ? a : b;  
}
```

# 函数模板的实例化

## 主函数中增加代码

```
cout << "Larger of \"<<small_int<<\" and 2.5 is \" <<  
larger_int(small_int, 19.6) << endl;
```

## 会出现什么结果？

```
✗ C2672 "larger": 未找到匹配的重载函数  
✗ C2784 "T larger(T,T)": 未能从"double"为"T"推导 模板 参数  
✗ C2782 "T larger(T,T)": 模板 参数"T"不明确  
▶ abc E0304 没有与参数列表匹配的 函数模板 "larger" 实例
```

## 利用函数模板实例化，**显式**指定类型参数

```
cout << "Larger of \"<<small_int<<\" and 2.5 is \" <<  
larger<double>(small_int, 19.6) << endl;
```

# 带有多参数的函数模板

修改函数模板的定义：

```
template <typename T1, typename T2>
??? larger(T1 a, T2 b)
{
    return a > b ? a : b;
}
```

如何确定函数模板的返回值类型？

```
template <typename TReturn, typename T1, typename T2>
TReturn larger(T1 a, T2 b)
{
    return a > b ? a : b;
}

cout << "Larger of \"<<small_int<<\" and 2.5 is " <<
larger<int, double, int>(1.5, 2) << endl;
```

# 带有多个参数的函数模板

## 指定部分类型参数

```
template <typename TReturn, typename T1, typename T2>
TReturn larger(T1 a, T2 b)
{
    return a > b ? a : b;
}
```

```
cout << "Larger of \"<<small_int<<\"\\ and 2.5 is \" <<
larger<int, double, int>(1.5, 2) << endl;
```

```
cout << "Larger of \"<<small_int<<\"\\ and 2.5 is \" <<
larger<int, double>(1.5, 2) << endl;
```

```
cout << "Larger of \"<<small_int<<\"\\ and 2.5 is \" <<
larger<int>(1.5, 2) << endl;
```



# 函数模板的返回值类型推断

当无法确定函数模板的返回值类型时，使用auto关键字，自动推断函数模板的返回值类型

```
template <typename T1, typename T2>  
auto larger(T1 a, T2 b)  
{  
    return a > b ? a : b;  
}
```

# 函数模板参数的默认值

可以在定义函数模板时，给出类型参数的默认值，如果未通过实例化给出类型参数值，或者无法推断出类型参数值，则使用该默认的类型参数值

```
template <typename TReturn = double, typename T1,  
typename T2>  
TReturn larger(T1 a, T2 b)  
{  
    return a > b ? a : b;  
}
```

## 函数模板的非类型参数

非类型参数，是指函数模板的普通参数，类似于普通函数的参数，其类型是基本类型、复合类型或者自定义类型

```
template <typename T, int N>
T average(const T array[N])
{
    T sum {}; // Accumulate total in here
    for (size_t i {}; i < N; ++i)
        sum += array[i]; // Sum array elements
    return sum / N; // Return average
}
```

## 练习9.1

编写函数模板，实现将n个数据进行由小到大排序的功能

- 排序算法自行选择
- 能够处理的数据类型包括：
  - 整型
  - 浮点型
  - 字符型
  - 自定义类型，如复数类型



## 函数模板



## 类模板的基本概念



## 类模板的继承和派生



## 类模板综合示例



## 标准模板库程序设计

# 类模板

类模板（带**类型参数**或**普通参数**的类）用来定义具有共性的一组类

- “共性”通过类模板参数体现
- 通过类模板的定义，类中的某些数据成员、某些成员函数的参数、某些成员函数的返回值都可以是任意类型的
- 可将程序所处理的对象（数据）的类型参数化，从而使同一段程序可用于处理多种不同类型的对象（数据）

# 类模板的定义方式

**template** <类型形参或普通形参的说明列表>

**class** 类模板名

{ 带上述**类型形参**或**普通形参名**的类定义体 };

- 类型形参
  - **typename** 类型形参名（或：**class** 类型形参名）
- 普通形参
  - 数据类型 普通形参名
- 类模板名
  - 标识符

# 类模板的说明

类定义体中应使用上述的“类型形参名”及“普通形参名”。

利用类模板说明类对象时，要随类模板名同时给出对应于类型形参或普通形参的具体实参（从而**实例化**为一个具体的类）。说明格式为：

- 类模板名 < 形参1的相应实参, ... , 形参n的相应实参 >

注意：类型形参的相应实参为**类型名**，而普通形参的相应实参必须为一个**常量**。



# 类模板的成员函数

类模板的成员函数既可以在类体内进行说明（自动按内联函数处理），也可以在类体外进行说明

- 在类体外说明（定义）时使用如下格式：

**template** < 形参1的说明, ... , 形参n的说明 >

返回类型 类模板名 < 形参1的名字, ... , 形参n的名

字 >::函数名( 形参表 ) {

... // 函数体

};

# 类模板的成员函数

上述的“形参1的名字”来自于“形参1的说明”，由“甩掉”说明部分的“类型”而得，是对类型形参或普通形参的使用。而“类模板名 < 形参1的名字, ..., 形参n的名字 >::”所起的作用正是在类体外定义成员函数时在函数名前所加的类型限定符！

# 类模板的成员函数

**【例如】** 对具有一个类型参数T的类模板TestClass，在类体外定义其成员函数getData时的大致样式如下：

```
template <typename T>
T    TestClass<T> ::getData( 形参表 ) {
    ...           //函数体
};
```

- 其中的“TestClass<T>::”所起的作用正是在类体外定义成员函数时在函数名前所加类限定符！

# 类模板的实例化

不能使用类模板来直接生成对象

- 类型参数是不确定的

必须先为模板参数指定“实参”

- 即为模板“实例化”

实例化格式

类模板名 <具体的实参表>

利用类模板生成对象

类模板名 <具体的实参表> 对象名称

## 【例9.4】仅使用类型参数的类模板示例

```
#include <iostream>
using namespace std;
template <typename T>
class TestClass {
public:
    T buffer[10];
    //T类型的数据成员buffer数组大小固定为10
    T getData(int j);
    //获取T类型buffer(数组)的第j个分量
};

template <typename T>
T TestClass<T>::getData(int j) {
    return *(buffer+j);
};
```

```
void main() {  
    TestClass<char> ClassInstA;  
    //char取代T，从而实例化为一个具体的类  
    char cArr[6]="abcde";  
  
    for(int i=0; i<5; i++)  
        ClassInstA.buffer[i]=cArr[i];  
  
    for(int i=0; i<5; i++) {  
        char res=ClassInstA.getData(i);  
        cout<<res<<"  ";  
    }  
    cout<<endl;
```

```
TestClass<double> ClassInstF;  
//实例化为另外一个具体的类  
double fArr[6]={12.1, 23.2, 34.3, 45.4,  
56.5, 67.6};  
for(i=0; i<6; i++)  
    ClassInstF.buffer[i]=fArr[i]-10;  
for(i=0; i<6; i++) {  
    double res=ClassInstF.getData(i);  
    cout<<res<<"    ";  
}  
cout<<endl;  
}
```

程序执行后的显示结果如下:

```
a    b    c    d    e  
2.1   13.2   24.3   35.4   46.5   57.6
```

## 【例9.5】 仅使用普通参数(非类型参数)的类模板示例

```
#include <iostream>

using namespace std;

template <int i> class TestClass {
public:
    int buffer[i];
    //使buffer的大小可变化，但其类型则固定为int
    int getData(int j);
};

template <int i>
int TestClass<i>::getData(int j) {
    return *(buffer+j);
};
```



```
void main() {  
    TestClass<6> ClassInstF;  
    double fArr[6]={12.1, 23.2, 34.3, 45.4,  
56.5, 67.6};  
    for(int i=0; i<6; i++)  
        ClassInstF.buffer[i]=fArr[i]-10;  
    for(int i=0; i<6; i++) {  
        double res=ClassInstF.getData(i);  
        cout<<res<<"    ";  
    }  
    cout<<endl;  
}
```

程序执行后的显示结果如下：

2 13 24 35 46 57

## 【例9.6】既使用类型参数又使用普通参数的类模板示例

```
#include <iostream.h>
#include <string.h>
using namespace std;
template <typename T, int i> class TestClass {
    public:
        T buffer[i];
        //T类型的buffer, 其大小随普通形参i的值变化
        T getData(int j);
};
template <typename T, int i>
T TestClass<T,i>::getData(int j) {
    return *(buffer+j);
};
```

```
void main() {  
    TestClass<char, 5> ClassInstA;  
    char cArr[6]="abcde";  
  
    strcpy(ClassInstA.buffer, cArr);  
  
    for(int i=0; i<5; i++) {  
        char res=ClassInstA.getData(i);  
        cout<<res<<"    ";  
    }  
    cout<<endl;
```

```
TestClass<double, 6> ClassInstF;  
double fArr[6]={12.1, 23.2, 34.3, 45.4,  
56.5, 67.6};  
for(int i=0; i<6; i++)  
    ClassInstF.buffer[i]=fArr[i]-10;  
for(int i=0; i<6; i++) {  
    double res=ClassInstF.getData(i);  
    cout<<res<<"    ";  
}  
cout<<endl;  
}
```

程序执行后的显示结果如下:

a	b	c	d	e	
2.1	13.2	24.3	35.4	46.5	57.6

# 类模板的静态成员

类模板也允许有静态成员。实际上，它们是类模板之实例化类的静态成员。也就是说，对于一个类模板的每一个实例化类，其所有的对象共享其静态成员。

- 【例如】

```
template <typename T> class C{  
    static T t; //类模板的静态成员t  
};
```

# 类模板的静态成员

类模板的静态成员在模板定义时是不会被创建的，其创建是在**类的实例化之后**

- **【例如】**

```
CA<int>aiobj1, aiobj2;
```

```
CA<char>acobj1, acobj2;
```

- 对象 aiobj1 和 aiobj2 将共享实例化类 CA<int>的静态成员 int t，而对象acobj1, acobj2 将共享实例化类CA<char>的静态成员 char t。

# 类模板的友元

类模板定义中允许包含友元。讨论类模板中的友元函数，因为说明一个友元类，实际上相当于说明该类的成员函数都是友元函数。

- 该友元函数为一般函数，则它将是该类模板的所有实例化类的友元函数。
- 该友元函数为一函数模板，但其类型参数与类模板的类型参数无关。则该函数模板的所有实例化（函数）都是类模板的所有实例化类的友元。

# 类模板的友元

- 更复杂的情形是，该友元函数为一函数模板，且它与类模板的类型参数有关。例如，函数模板可以用该类模板作为其函数参数的类型。在友元函数模板定义与相应类模板的类型参数有关时，该友元函数模板的实例有可能只是该类模板的**某些特定实例化**（而不是所有实例化）类的友元



# 类型参数检测与特例版本

大多数类模板不能任意进行实例化。也就是说类模板的类型参数往往在实例化时不允许用任意的类（类型）作为“实参”。模板的“实参”不当，主要会在实例化后的函数成员调用中体现出来

## 【例如】

```
template <typename T> class stack {  
    //栈中元素类型为T 的stack 类模板  
    T num [MAX]; //num 中存放栈的实际数据  
    int top; //top 为栈顶位置  
public:  
    stack () { top=0; } //构造函数  
    void push (T a) { num[top++]=a; }  
    //将数据a“压入” 栈顶  
    void showtop() { // 显示栈顶数据  
        //模板中通用的showtop, 显示栈顶的那一个T 类型的数据  
        //（必须为可直接通过运算符“<<”来显示的数据）  
        if (top==0) cout << "stack is empty!"<< endl;  
        else cout<<"Top_Member:"<<num[top-1]<<endl;  
    }  
};
```

在类模板stack 中，以下实例化都是可行的：

```
stack<int>i1,i2;  
stack<char>c1,c2;  
stack<float>f1,f2;
```

- 但如果采用用户定义类型而又未在该类中对运算符“<<”进行重载时，就会产生问题，例如：

```
stack<complex>com1,com2;
```

- 由于在执行com1.showtop() 函数时，将需要对complex 类型的数据num[top-1] 通过使用运算符“<<”来进行输出，而系统和用户都没有定义过这种操作，因此，类模板stack 的实例化stack<complex>就是不可行的了

# 类型参数检测与特例版本

如果用户在上述情况下，需要使  
**stack<complex>**可行，可有几个办法：

- 对于类complex 追加插入运算符 “<<”的重载定义；
- 也可在类模板stack 的定义中增加一个“特例版本”（也称“特殊版本”）的定义。例如在上例中，可以在类模板定义之后给出如下形式的特例版本：

```
void stack<complex>::showtop() { /*专用于
    complex 类型的showtop（专门补充的“特例版本”
    ），显示栈顶的//那一个complex 型数据。其中的
    stack<complex>为一个实例化后的模板类*/
    if (top==0)
        cout << "stack is empty!"<< endl;
    else
        cout<<"Top_Member:"<<num[top-1].get_r()
        <<", " <<num[top-1].get_i()<<endl;
}
```

# 类型参数检测与特例版本

假设自定义的复数类型`complex` 中具有公有的成员函数`get_r()`以及`get_i()`，用于获取复数的实部和虚部。如此，当实例化`stack<complex>`时将按该特例版本的定义进行。

概括地说，当处理某一类模板中的可变类型`T`型数据时，如果处理算法并不能对所有的`T`类型取值做统一的处理，此时可通过使用专门补充的所谓特例版本来对具有特殊性的那些`T`类型取值做特殊处理。

## 类型参数检测与特例版本

也可以对函数模板，或类模板的个别函数成员补充其“特例版本”定义。

例如，可将该例的showtop功能进一步划分，让showtop调用另一个新增加的show函数，而由show函数具体考虑对两种情况的处理：一种处理可直接通过运算符“<<”来显示的数据，另一种“特例版本”专用于处理complex类型的数据。

```
class complex {  
    double real, image;  
public:  
    ...  
};
```

//复数类型complex

```
template <typename T>  
class stack {  
    T data [20];  
    int top;  
public:  
    void showtop(void);  
    ...  
};
```

//显示栈顶数据



```
template <typename T>
void stack<T>::showtop(void)
    //通用的showtop
    //T类型数据，可直接通过“<<”来一次性输出的数据
{
    if (top==0)
        cout << "stack is empty!"<< endl;
    else
        cout<<"Top_Member:"<<data[top-1]<<endl;
}
```

```
void stack<complex>::showtop(void)
//专用于complex类型的showtop
//显示栈顶的那一个complex型数据，它不可直接通过
// “<<” 一次性输出！
{
    if (top==0)
        cout << "stack is empty!"<< endl;
    else
        cout<<"Top_Member:"<<data[top-1].get_r()
        <<","<<data[top-1].get_i()<<endl;
}
```

```
void main() {  
    stack<int> s1;  
    for (int i=1; i<=6; i++)  
        s1.push(2*i);  
        //“压入” : 2,4,6,8,10,12 (栈顶为12)  
    s1.showtop();           //调用模板中通用的showtop  
    stack<complex> s1c;  
    complex c1(1.1, 1.111), c2(2.2, 2.222);  
    s1c.push(c1);  
    s1c.push(c2); //“压入” 复数c2 (处于栈顶)  
    s1c.showtop();  
    //调用专门补充的“特例函数” showtop  
}
```



## 函数模板



## 类模板的基本概念



## 类模板的继承和派生



## 类模板综合示例



## 标准模板库程序设计

一般类（其中不使用类型参数的类）作基类，派生出类模板（其中要使用类型参数）

```
class CB {  
    //CB 为一般类（其中不使用类型参数），它将作为类模板CA的基类  
    ...  
};  
template <typename T> class CA:public CB {  
    //被派生出的CA 为类模板，使用了类型参数T，其基类CB为一般类  
    T t; //私有数据为T类型的  
public:  
    ...  
};
```

类模板作基类，派生出新的类模板。但仅基类中用到类型参数T，而派生的类模板中不使用T

```
template <typename T> class CB {  
    //CB 为类模板（其中使用了类型参数T），它将作为类模板CA的基类  
    T t; //私有数据为T类型的  
public:  
    T gett() {return t; } ...//用到类型参数T  
};  
template <typename T> class CA : public CB<T> {  
    //CA 为类模板，其基类CB 也为类模板。注意，类型参数T  
    //将被“传递”给基类CB，本派生类中并不使用该类型参数T  
    double t1; //私有数据成员  
public: ...  
}; /*基类的名字应为实例化后的“CB<T>”而非仅使用“CB”。例如，  
    在本例的派生类说明中，要对基类进行指定时必须使用“CB<T>”  
    而不可只使用“CB”*/
```

类模板作基类，派生出新的类模板，且基类与派生类中均使用同一个类型参数T。

```
template <typename T> class CB {  
    //CB 为类模板（其中使用了类型参数T），它将作为类模板CA的基类  
    T t; //数据成员为T 类型的  
public:  
    T gett() { //用到类型参数T  
        return t;  
    }  
    ...  
};  
template <typename T> class CA : public CB<T> {  
    /*CA 为类模板，其基类CB 也为类模板。注意，类型参数T 将被  
    “传递”给基类CB；本派生类中也将使用这同一个类型参数T */  
    T t1; //数据为T 类型的  
public: ...  
};
```

类模板作基类，派生出新的类模板，但基类中使用类型参数T2，而派生类中使用另一个类型参数T1(而不使用T2)。

```
template <typename T2> class CB {  
    //CB 为类模板（其中使用了类型参数T2），它将作为类模板CA的基类  
    T2 t2; //数据为T2 类型的  
public:  
    ...  
};  
template <typename T1, typename T2> class CA : public CB<T2>  
{ /*CA 为类模板，其基类CB 也为类模板。注意，类型参数T2 将被  
“传递”给基类CB；本派生类中还将使用另一个类型参数T1*/  
    T1 t1; //数据为T1 类型的  
public:  
    ...  
};
```





## 函数模板



## 类模板的基本概念



## 类模板的继承和派生



## 类模板综合示例



## 标准模板库程序设计

## 【例9.7】 队列类模板

队列与栈不同，对数据采用“先进先出”的管理方式（而栈则使用“先进后出”方式）。队列数据放于作为类成员的动态数组queue之中，在构造函数中，将通过new来生成该动态数组，动态数组queue的大小由类的私有数据成员Maxsize之值来确定。

主要成员函数为：

- 队尾增加数据Add
- 队首删除数据Delete

```
#include <iostream>
using namespace std;
#include <process.h> //exit(0)
template <typename keytype>
class Queue {
    int Maxsize; //队列的大小
    int front,rear; //元素从queue[front+1]到queue[rear]
    keytype *queue; //动态数组queue, 用来存放队列数据
public:
    Queue(int size){ //构造函数, 生成动态数组来存放队列数据
        Maxsize=size;
        queue=new keytype[Maxsize];
        front=rear=-1; //意味着队列为空
    };
};
```

```
int IsFull () {  
    if (rear==Maxsize-1)  
        return 1;  
    else  
        return 0;  
};  
int IsEmpty () {  
    if (front==rear)  
        return 1;  
    else  
        return 0;  
};  
void Add(const keytype &);  
keytype Delete(void);  
};
```

//Delete在类体外定义，函数名前要加类限定符“Queue<keytype>::”

```
template <class keytype>
keytype Queue<keytype>::Delete(void) {
    if (IsEmpty()) {
        cout << "the queue is empty"<<endl;
        exit (0);
    }
    return queue[++front];
}
```

//Add在类体外定义

```
template <class keytype>
void Queue<keytype>::Add(const keytype & item) {
    if (IsFull())
        cout << "the queue is full"<<endl;
    else
        queue[++rear]=item;
};
```

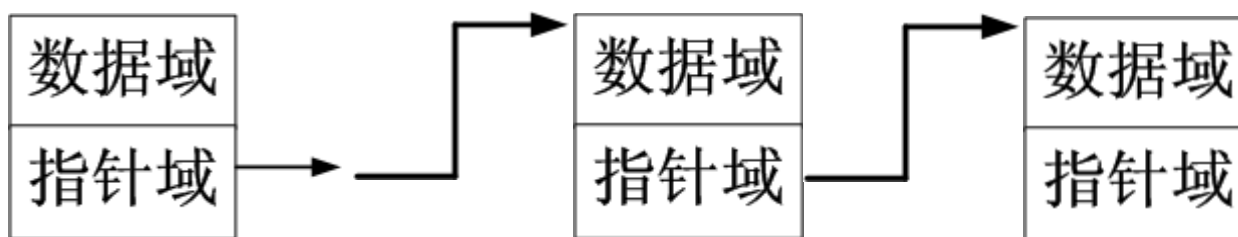
```
void main() {  
    int i=0;  
    Queue<int> Qi(10);  
    Queue<double> Qf1(10), Qf2(10);  
    while (!Qi.IsFull()) { //Qi中只能存10个数  
        Qi.Add(2*i++);  
        Qf1.Add(3.0*i);  
    }  
    for (i=0; i<4; i++) {  
        //四次循环，每次总先往Qf2的队列尾部加入两个数  
        //而后又从首部删取一个数并输出  
        Qf2.Add(4.5*Qi.Delete());  
        //从Qi首删取一元素，乘以4.5，而后将其加入到Qf2尾部  
        Qf2.Add(Qf1.Delete()/2.0);  
        cout<<Qf2.Delete()<<endl;  
        //四次循环往Qf2队列尾加入：0*4.5, 2*4.5, 4*4.5, 6*4.5  
    }  
}
```

## 【例9.8】用类模板实现有序单向链表，能够处理整型、浮点型和字符型数据并按照由小到大顺序排列链表节点

- 链表的结构
- 链表类模板的设计
- 链表的操作与相应成员函数
  - 插入节点
  - 删除节点
  - 查找节点
- 链表的创建与使用

## 链表的结构

- 数据域
- 指针域





## 链表相关类模板的设计

- 链表节点类模板

```
template <typename T>
class Node
{
public:
    T num;
    Node* next;
    static int TotalCount; //统计链表节点的数量
public:
    Node(T n);
    ~Node();
};
```

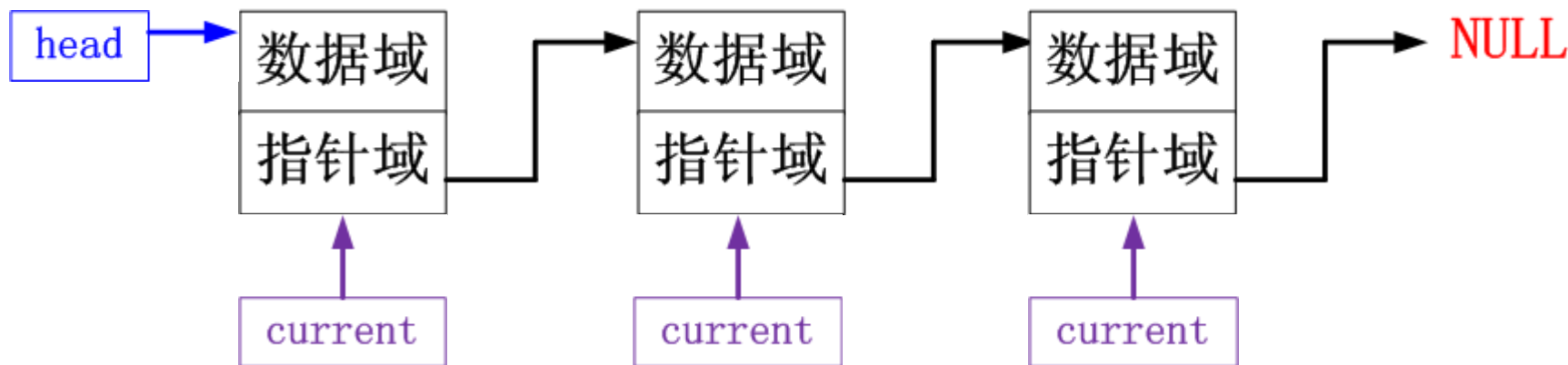
## 链表类的设计

- 链表类模板

```
template <typename T>
class List
{
private:
    Node<T>* head;
    Node<T>* tail;
    int nodeCount; // 链表节点的数量
public:
    void Insert(T n); // 插入节点
    void Remove(T n); // 删除节点
    void Find(T n); // 查找节点
    List();
    void Print(); // 打印链表的数据项
};
```

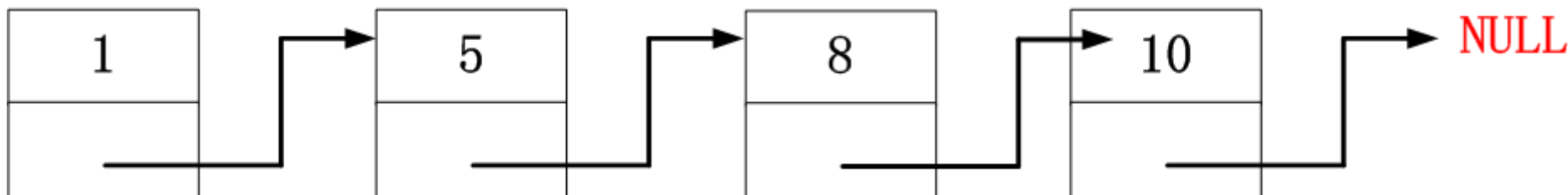
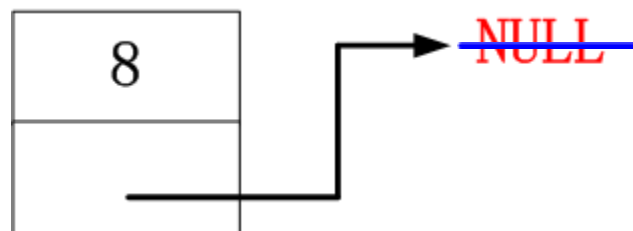
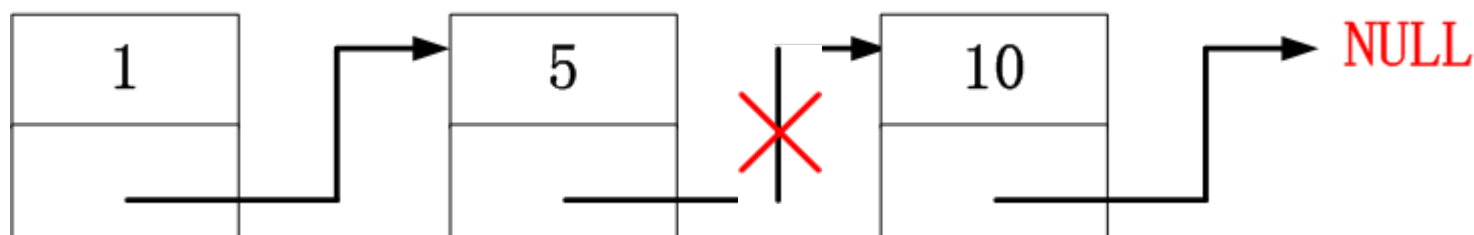
## 链表的访问

- 根据链表的头指针（\*head）确定链表的入口地址
- 建立临时指针，通过该指针的移动，访问链表的每一个节点，直到链表的尾节点
  - 临时指针根据当前节点的指针域所指地址进行移动



## 链表的操作

- 链表节点的插入



## 链表的操作

- 链表节点的插入的成员函数

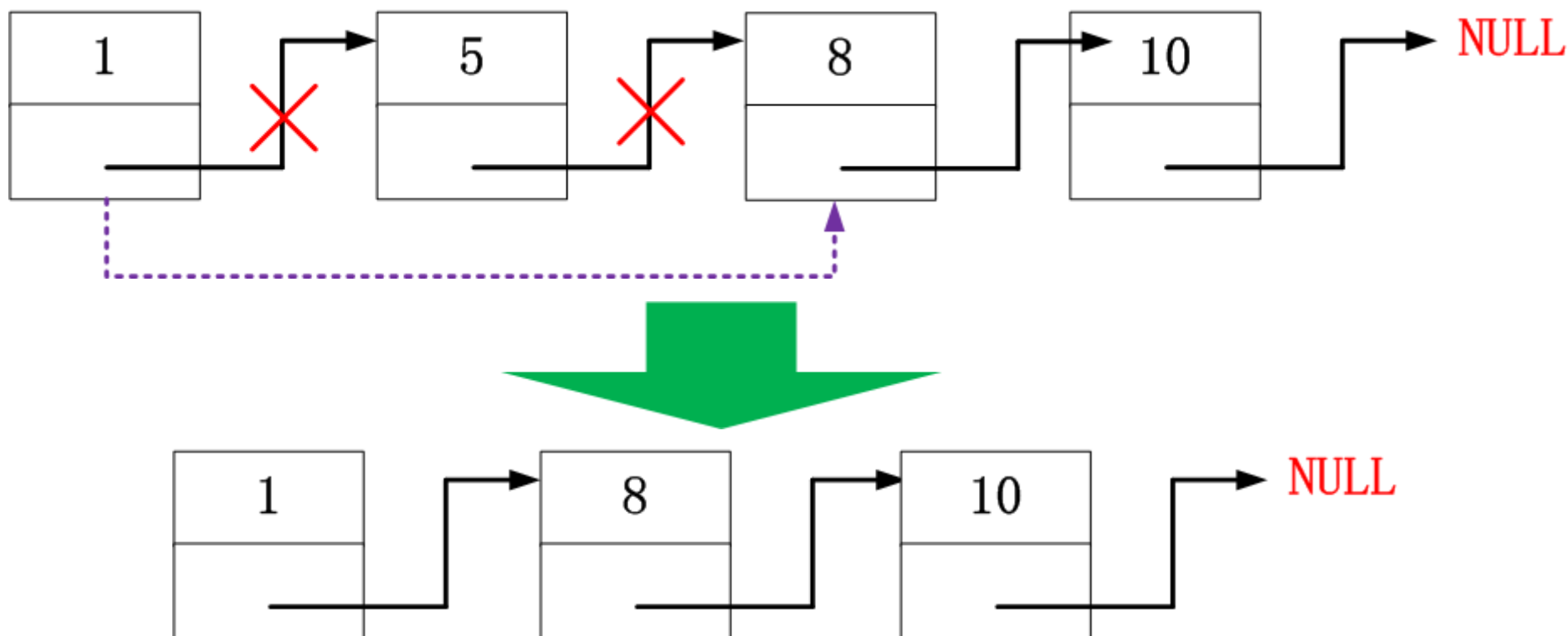
```
template <typename T> void List<T>::Insert(T n)
{
    Node<T>* tmp=new Node<T>(n);
    if(head==NULL)
    {
        head=tail=tmp;
        nodeCount++;
    }
}
```

```
else
{
    if (n<head->num)
    {
        tmp->next=head;
        head=tmp;
        nodeCount++;
        return;
    }
    if (n>tail->num)
    {
        tail->next=tmp;
        tail=tmp;
        nodeCount++;
        return;
    }
}
```

```
Node<T>* curr=head;
while (curr->next!=NULL)
{
    if ( (curr->num<=n) && (curr->next->num>n) )
    {
        tmp->next=curr->next;
        curr->next=tmp;
        nodeCount++;
        return;
    }
    curr=curr->next;
}
}
```

## 链表的操作

- 链表节点的删除





## 链表的操作

- 链表节点的查找
  - 输入数据值
  - 返回该数据值所属链表节点的位置
- 链表全部节点数据项的输出
  - 从第一个节点开始，逐项输出节点存储的数据

## 链表的创建与使用

- 主函数中进行链表的创建以及对链表进行各类操作

```
void main() {  
    List<int> list;  
    int count;  
    cout<<endl<<"Please input the count of the node  
of the list: ";  
    cin>>count;  
    srand(time(0));  
    for(int i=1;i<=count;i++) {  
        int tmp = rand()%100;  
        //double tmp_double = tmp/7.0;  
        list.Insert(tmp);  
    }  
    list.Print();  
}
```

```
int number;  
cout<<endl<<"Please input the number of the  
node you want to insert: ";  
cin>>number;  
list.Insert(number);  
list.Print();  
cout<<endl<<"Please input the number of the  
node you want to delete: ";  
cin>>number;  
list.Remove(number);  
list.Print();  
cout<<endl<<"Please input the number of the  
node you want to search: ";  
cin>>number;  
list.Find(number);  
list.Print();  
}
```

## 练习9.2

上机实现【例9.8】链表类模板程序，观察链表类模板的程序与普通链表类程序之间的区别，并完善：

- 成员函数：Remove( ),链表节点删除
- 成员函数：Find( ),链表节点查找
- 成员函数：Print( ),按顺序输出链表全部节点的数据项
- 两个类模板中的其它未定义成员函数



## 函数模板



## 类模板的基本概念



## 类模板的继承和派生



## 类模板综合示例



## 标准模板库程序设计

# C++标准库

提供海量的类型（类模板）和函数（函数模板）

- `iostream`
- `fstream`
- `string`
- `clock()`
- `pow()`
- `sqrt()`
- `pause()`
- .....

# C++标准库

标准库以“头文件”的形式呈现

```
#include<iostream>
```

```
#include<cmath>
```

```
#include<string>
```

头文件的组件封装于命名空间std中

- using namespace std;

# C++标准模板库

模板机制的主要目标是程序的通用性和可重用性

C++编译系统为用户提供一个标准模板库（Standard Template Library, STL）

- 系统已经编好的类模板和函数模板
- 编写程序时可直接调用
- 主要类模板：array、vector、list、deque、queue、stack、map、multimap、set、multiset
- 主要函数模板：sort、copy、search、reverse



# 标准模板库程序设计

**STL (Standard Template Library)**，即标准模板库，是一个高效的C++程序库。STL是ANSI/ISO C++标准库的一个子集，它提供了大量可扩展的类模板，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，类似于Microsoft Visual C++中的**MFC** (Microsoft Foundation Class Library)。

# STL程序设计基本思想

从逻辑结构和存储结构来看，基本数据结构的数量是有限的。对于其中的数据结构，用户可能需要反复的编写一些类似的代码，只是为了适应不同数据的类型变化而在细节上有所出入。如果能够将这些经典的数据结构，采用**类型参数**的形式，设计为**通用的类模板和函数模板**的形式，允许用户**重复利用已有的数据结构**构造自己特定类型下的、符合实际需要的数据结构，无疑将简化程序开发，提高软件的开发效率，这就是STL编程的基本设计思想。

# 标准模板库程序设计

从**逻辑层面**来看，STL提倡使用现有的模板程序代码开发应用程序，是一种代码的重用技术（reusability）。许多程序设计语言通过提供标准库来实现代码重用的机制。STL是一个通用组件库，它的目标是将常用的数据结构和算法标准化、通用化，这样用户可以直接套用而不用重复开发它们，从而提高程序设计的效率。

# 标准模板库程序设计

从**实现层面**看，STL是一种**类型参数化**（type parameterized）的程序设计方法，是一个基于模板的标准类库，称之为**容器类**。每种容器都是一种已经建立完成的标准数据结构。在容器中，放入任何类型的数据，很容易建立一个存储该类型（或类）的数据结构。

# 泛型程序设计 (Generic Programming)

**泛型**即是指具有在多种数据类型上皆可操作的含义，与模板有些相似。

泛型编程是实现一个通用的标准容器库。所谓通用的标准容器库，就是要能够做到，比如用一个List类存放所有可能类型的对象这样的事；泛型编程让你编写完全一般化并可重复使用的算法，其效率与针对某特定数据类型而设计的算法相同。

实现算法与数据**分离**

# 标准模板库的六大部件

容器 (container)

迭代器 (iterator)

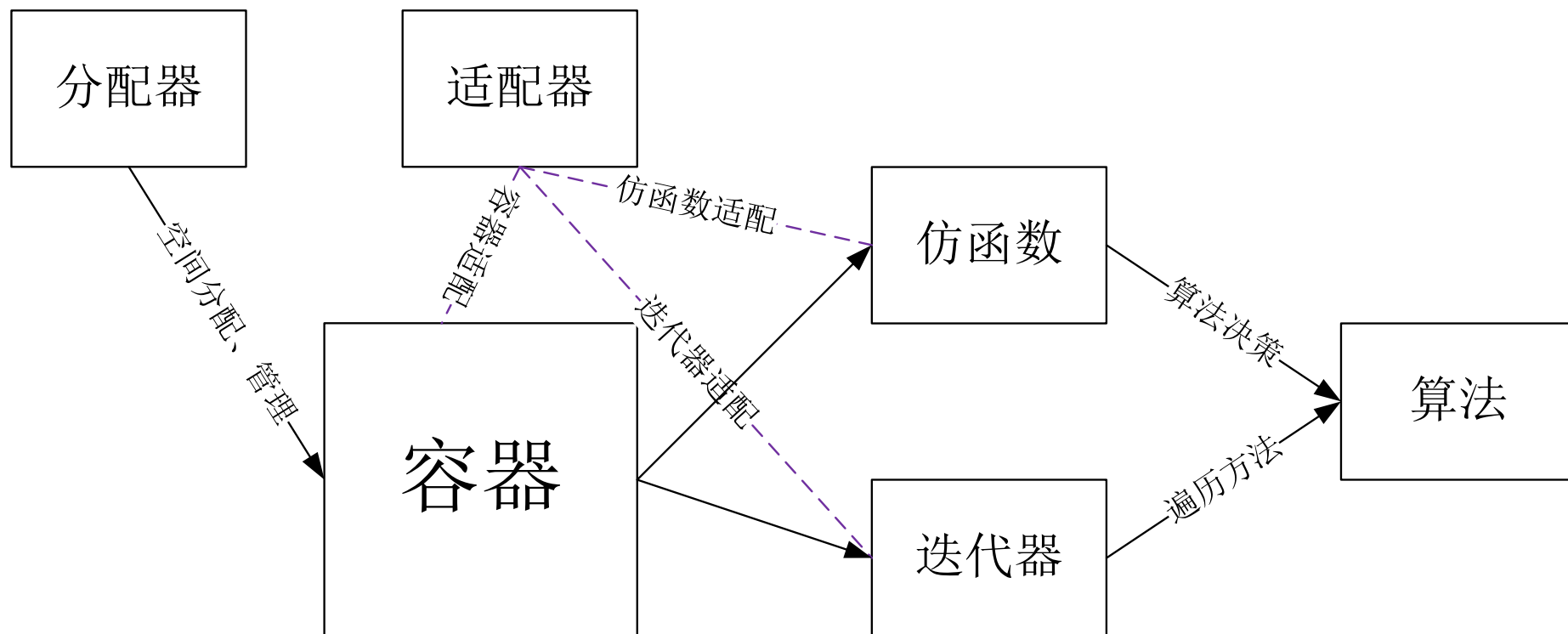
算法 (algorithm)

适配器 (Adaptor)

分配器 (Allocator)

仿函数 (Functor)

# 标准模板库的六大部件



# 容器 (Container)

容器是存放其他对象的对象。比如我们常见的C++内置数组，从广义上讲也属于一种容器。容器可以存放同一种类型的一组元素或对象，称为同类容器类 (homogenous container)；或者存放不同类型的元素或对象时，称为异类容器类 (heterogenous container)。对于STL容器库，其包含了两类容器，一种为顺序容器 (sequence container)，另一种为关联容器 (associative container)。



# 迭代器 (iterator)

在C++中，我们经常使用指针。而迭代器就是相当于指针，它提供了一种一般化的方法使得C++程序能够访问不同数据类型的顺序或者关联容器中的每一个元素，我们可以称它为“泛型指针”。

STL定义了五种迭代器类型，前向迭代器 (forward iterator)，双向迭代器 (bidirectional iterator)，输入迭代器 (input iterator)，输出迭代器 (output iterator)，随机访问迭代器 (random access iterator)。

# 算法 (Algorithm)

算法是STL中的核心，它包含了70多个通用算法。可以分为四类：不可变序列算法（non-modifying sequence algorithms）、可变序列算法(mutating sequence algorithms)、排序及相关算法(sorting and related algorithms)和算术算法（numeric algorithms）。

# 仿函数 (Functor)

也称为函数对象，它是定义了操作符operator( )的对象。

在C++中，除了定义了操作符operator( )的对象之外，普通函数或者函数指针也满足函数对象的特征。结合函数模板的使用，函数对象使得STL更加灵活和方便，同时也使得代码更为高效。

# 适配器 (Adapter)

适配器是一种接口类，可以认为是标准组件的改装。通过修改其它类的接口，使适配器满足一定需求，可分为容器适配器、迭代器适配器和函数对象适配器三种。

主要的容器适配器：

- stack
- queue
- priority\_queue

# 分配器 (Allocator)

分配器是STL提供的一种内存管理类模块。每种STL容器都是用了一种分配器类，用来封装程序所用的内存分配模式的信息。不同的内存分配模式采用不同的方法从操作系统中检索内存。

分配器类可以封装许多方面的信息，包括指针、常量指针、引用、常量引用、对象大小、不同类型指针之间的差别、分配函数与释放函数、以及一些函数的信息。分配器上的所有操作都具有分摊常量的运行时间。

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<functional>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int a[6] = { 27, 210, 12, 47, 109, 83 };
```

```
    vector<int, allocator<int>> va(a, a + 6);
```

```
    cout << count_if(va.begin(), va.end(),
                     not1(bind2nd(less<int>(), 40))) ;
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```

container

allocator

iterator

algorithm

function  
adaptor  
(negator)function  
adaptor  
(binder)

functor

# 容器

**容器**（container）就是**通用的数据结构**

- 数组（array）
- 向量（vector）
- 链表（list，双向链表）
- 单向链表（forward\_list）
- 双端队列（deque）
- 集合（set/ multiset）
- 映射（map/multimap）
- 无序集合（unordered\_set）
- 无序映射（unordered\_map）

容器用来装载数据对象

# 容器

## STL的所有容器都是类模板

- 每个容器只允许存储**相同类型的数据**
- 可创建不同的容器存储不同类型的数据
  - 容器的实例化类

不同的容器有不同的插入、删除和存取行为和性能特征，用户需要分析数据之间逻辑关系，为给定的任务选择最合适的容器

## 容器的分类

- 顺序容器
- 关联容器



# 容器的通用计算接口

通用运算	说明
$a == b$	同类容器的相等比较操作，判断是否相等，相等则为true
$a != b$	同类容器的不等比较操作，判断是否不等，不等则为true
$a < b$	两个容器大小判断，首先判断size，接着判断元素值， $a < b$ 则true
$a > b$	与上同，不过 $a > b$ 时为true
$a \leq b$	等同于！（ $a > b$ ）
$a \geq b$	等同与！（ $a < b$ ）
$r = b$	赋值操作

# 容器的通用迭代器接口

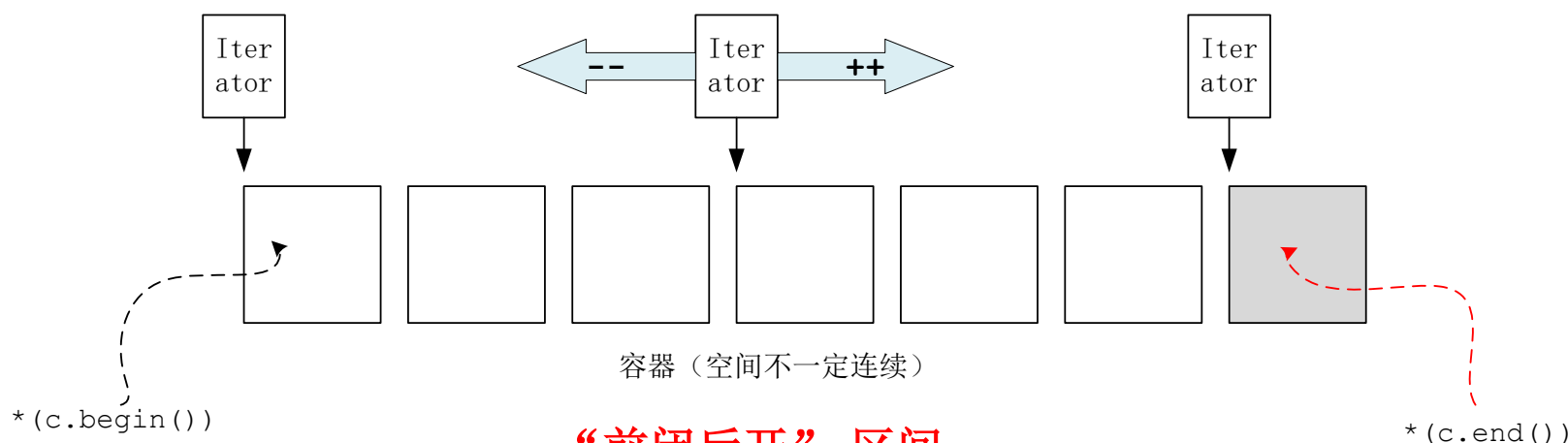
迭代方法	说明
begin ( )	返回一个指向容器第一个元素的迭代器
end ( )	返回一个指向容器末尾元素的迭代器
rbegin ( )	返回一个逆向迭代器，指向反序后的首元素
rend ( )	返回一个逆向迭代器，指向反序后的末尾元素

# 容器的其它接口

操作	说明
size()	返回容器元素个数
max_size()	返回容器最大的规模
empty()	判断容器是否为空，是，则返回true
swap()	交换两个容器的所有元素
clear()	清空容器的所有元素

# 顺序容器

顺序容器包含array, vector, list, forward\_list和deque, 其中array、vector和deque属于直接访问容器, list和forward\_list属于顺序访问容器。



```
Container<T> c;
```

```
...
```

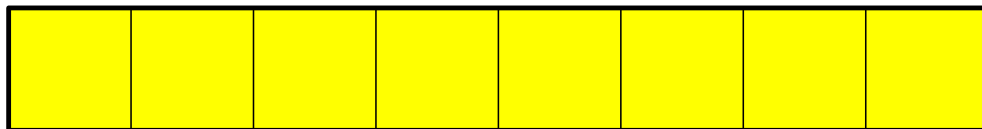
```
Container<T>::iterator itc = c.begin();
```

```
for ( ; itc != c.end(); itc++) ...
```

容器类名	特性	何时使用	头文件
<b>array</b> 数组	在内存中占有一块连续的空间，存储一个元素序列，但是，必须指定元素的数量	需要快速查找，不在意插入/删除的速度快慢。	<b>&lt;array&gt;</b>
<b>vector</b> 向量	在内存中占有一块连续的空间，存储一个元素序列。可以看作一个可自动扩充的动态数组，而且提供越界检查。可用[]运算符直接存取数据。	需要快速查找，不在意插入/删除的速度快慢。能使用数组的地方都能使用向量。	<b>&lt;vector&gt;</b>
<b>list</b> 链表	双向链表，每个节点包含一个元素。列表中的每个元素均有指针指向前一个元素和下一个元素。	需要快速的插入/删除，不在意查找的速度慢，就可以使用列表。	<b>&lt; list &gt;</b>
<b>forward_list</b> 单向链表	单向链表，每个节点包含一个元素。	同上	<b>&lt;forward_list&gt;</b>
<b>Deque</b> 双端队列	在内存中不占有一块连续的空间，介于向量和列表之间，更接近向量，适用于由两端存取数据。可用[]运算符直接存取数据。	可以提供快速的元素存取。在序列中插入/删的速度除较慢。一般不需要使用双端队列，可以转而使用 <b>vector</b> 或 <b>list</b> 。	<b>&lt; deque &gt;</b>

# 顺序容器

array



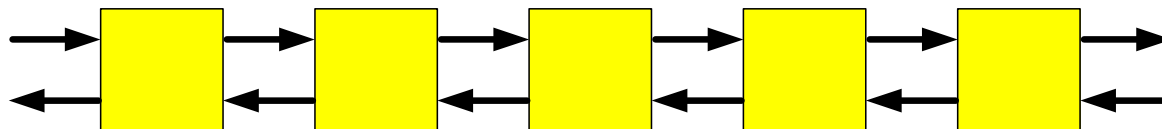
vector



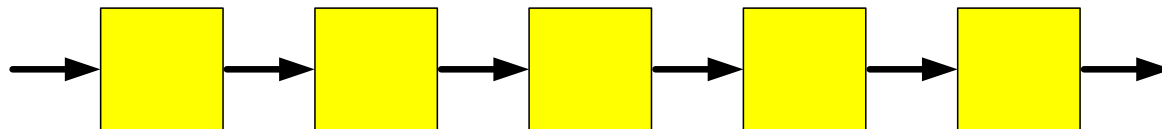
deque



list



forward\_list



# 顺序容器提供的操作

操作	V	A	L	F	D	描述
push_front() pop_front()	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	在前端添加或删除元素
push_back() pop_back()	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	在后端添加或删除元素
insert() erase()	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	在任意位置插入或删除一个或多个元素
front()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	返回第一个元素
back()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	返回最后一个元素
operator[] at()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	返回指定位置的元素
data()	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	返回开始位置的指针

# 向量

向量（vector）相当于一个动态数组，其可以动态存储元素，并提供对容器元素的随机访问。为了提高效率，vector并不是随着每一个元素的插入而增加长度，而是当vector要增加长度的时候，他分配的空间比当前所需的空间要多一些。这多一些的内存空间使需要添加新元素的时候不必再重新分配内存。



## 【例9.9】分析程序的运行结果

```
#include <iostream>
#include <vector> //使用向量容器须包含的头文件
#include <iomanip>
using namespace std;
const int n = 5;
int main() {
    int array[n] = { 12,4,5,9,1 };
    vector<int> vec1; // 构造1: 定义一个空的整型向量容器vec1
    int i;
    for (i = 0; i<n; i++) //赋值
        //vec1[i]=array[i]; //错误,因为vec1还没有分配内存空间
        vec1.push_back(array[i]); //压入向量尾部
    vector<int> vec2(vec1); // 构造2: 拷贝构造vec2
    vector<int> vec3(array, array + 3);
    // 构造3: 用array到array+3的值初始化
    vector<int> vec4(n, 3); //构造4: 用n个3初始化向量
```

```
for (i = 0; i<n; i++)  
    cout << setw(5) << vec1[i];  
cout << endl;  
for (i = 0; i<n; i++)  
    cout << setw(5) << vec2[i];  
cout << endl;  
for (i = 0; i<3; i++)  
    cout << setw(5) << vec3[i];  
cout << endl;  
for (i = 0; i<n; i++)  
    cout << setw(5) << vec4[i];  
cout << endl;  
}
```

程序运行结果：

12	4	5	9	1
12	4	5	9	1
12	4	5		
3	3	3	3	3

## 【例9.10】将学生成绩转换为标准分

学生的成绩一般是原始成绩，要将学生的成绩转换为标准分，必须首先比较所有学生的成绩，取得最高分，将学生原始成绩除以最高分，然后乘上100。

由于程序没有给出学生人数，所以采用向量作为数据存储结构，因为向量的元素个数可以自动的动态增长

```
#include<iostream>
#include<vector>
using namespace std;
int main() {
    vector<double> scorevector; //创建向量
    double max,temp;
    int i;
    cout<<"Input -1 to stop:"<<endl;
    cout<<"Enter the original score 1: ";
    cin>>max;
    scorevector.push_back(max);
```

```
for (i=1; true; i++) {  
    cout<<"Enter the original"  
    cout<<" score "<<i+1<<": ";  
    cin>>temp;  
    if (temp==-1) {  
        break;  
    }  
    scorevector.push_back(temp);  
    if (temp>max)  
        max=temp;  
}
```

```
max/=100;
cout<<"Output the standard scores: "
cout<<endl;
for (i=0;i<scorevector.size();i++) {
    scorevector[i]/=max;
    cout<<scorevector[i]<<" ";
}
cout<<endl;
return 0;
}
```

### 程序运行结果:

```
Input -1 to stop:
Enter the original score 1: 76
Enter the original score 2: 92
Enter the original score 3: 84
Enter the original score 4: -1
Output the standard scores:
82.6087 100 91.3043
```

## 【例9.11】 向量容器元素的插入和删除

```
#include <iostream>
#include <vector>
#include <iomanip>
#include <string>
using namespace std;

void display(vector<string>& _str) {
    // 向量元素显示函数
    cout<<"there are "<<_str.size()<<" elements in
the vector."<<endl;
    for(int i=0; i<_str.size();i++)
        cout<<setw(15)<<_str[i];    // 逐个显示元素
    cout<<endl;
}
```

```
void main() {  
    string str[3]={ "Hello", "C++", "Love"};  
    vector<string> vec1; //将str至str+3之间的元素插入vec1  
    vec1.insert(vec1.begin(), str, str+3);  
    vector<string> vec2;  
    //将vec1.begin()至vec1.end()之间的元素插入到vec2, 等效于将  
    //vec1复制到vec2中  
    vec2.insert(vec2.end(), vec1.begin(), vec1.end());  
    vec2.insert(vec2.begin(), 1, "welcome to C++");  
    //在vec2.begin()前插入字符串  
    display(vec1);  
    display(vec2);  
    vec1.clear(); //清除整个vec1  
    display(vec1);  
    vec2.erase(vec2.begin()); //删除vec2的首元素  
    display(vec2);  
    vec2.pop_back(); //删除vec2的末尾元素  
    display(vec2);  
}
```



## 程序运行结果：

```
there are 3 elements in the vector.
```

```
    Hello          C++          Love
```

```
there are 4 elements in the vector.
```

```
  welcome to C++      Hello      C++      Love
```

```
there are 0 elements in the vector.
```

```
there are 3 elements in the vector.
```

```
    Hello          C++          Love
```

```
there are 2 elements in the vector.
```

```
    Hello          C++
```

# 双端队列

双端队列是一种增加了访问权限的队列。在队列中，我们只允许从队列的一端添加元素，在队列的另一端提取元素；在双端队列中，其支持两端的出队和入队，这个我们可以通过前面所述的顺序容器接口看出。vector与deque同属于随机访问容器，vector拥有的成员函数deque也都含有。这个我们在顺序容器一般接口表中可以看出。

## 【9.12】建立双端队列并插入数据

```
#include <iostream>
#include <deque> //使用deque需要包含的头文件
#include <iomanip>
using namespace std;
const int n = 10;
int main() {
    deque<int> de;
    int array[n] = { 10,1,3,4,5,7,2,9,8,6 };
    int i;
    for (i = 0; i<5;i++) {
        de.push_back(array[i]);
        de.push_front(array[n - 1 - i]);
    }
```

```
for (i = 0; i<n; i++)  
    cout << setw(5) << de[i];  
cout << endl;  
for (i = 0; i<n; i++) {  
    de.pop_front();  
    de.push_back(array[i]);  
}  
for (i = 0; i<n; i++)  
    cout << setw(5) << de[i];  
cout << endl;  
}
```

程序运行结果：

7	2	9	8	6	10	1	3	4	5
10	1	3	4	5	7	2	9	8	6

# 链表

链表（list）是由节点组成的双向链表，每一个节点都包括一个元素（即实际存储的数据）、一个前驱指针和一个后继指针，可提供两个方向的遍历功能。list无需分配指定的内存大小且可以任意伸缩，这是因为它存储在非连续的内存空间中，并且由指针将各元素链接起来。

# 链表的其它成员函数

成员函数	功能描述
sort	排序( list 不支持STL 的算法sort)
remove	删除和指定值相等的所有元素
unique	删除所有和前一个元素相同的元素
merge	合并两个链表，并清空被合并的那个
reverse	颠倒链表
splice	在指定位置前面插入另一链表中的一个或多个元素,并在另一链表中删除被插入的元素

## 【例9.13】 建立一个链表，并将元素由小到大排序

```
#include <iostream>
#include <list>
#include <iomanip>
using namespace std;
const int n = 5;

void display(list<int> _list) {
    if (!_list.empty()) {
        list<int>::iterator it;
        for (it = _list.begin(); it != _list.end(); it++)
            cout << setw(5) << *it;
        cout << endl;
    }
    else
        cout << setw(5) << "Null list" << endl;
}
```

```
int main() {  
    int array[n] = { 2,7,5,3,34 };  
    list<int> list1;  
    list1.insert(list1.begin(), array, array + n);  
    display(list1);  
    list1.sort(); //默认按升序排列  
    display(list1);  
    list<int> list2 = list1;  
    for (int i = 0; i < 4; i++)  
        list2.remove(i);  
    //将列表中小于4的元素移除, remove的作用是删除指定值的节点  
    display(list2);  
  
    list1.merge(list2);  
    display(list1);  
    display(list2); //合并后, list2变为空链表  
    list1.reverse(); //逆序  
    display(list1);  
}
```



## 程序的运行结果：

```
2  7  5  3  34
2  3  5  7  34
5  7  34
2  3  5  5  7  7  34  34
Null list
34  34  7  7  5  5  3  2
```

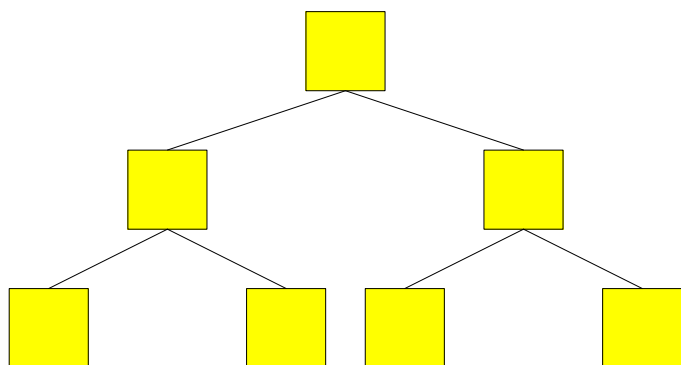
# 关联容器

关联容器也是一组特定类型对象的集合，它通过关键字（key）高效地查找和读取元素，而顺序容器通过位置查找元素。

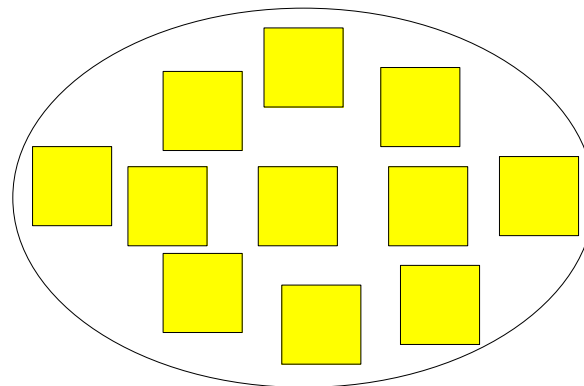
- 集合：只存储关键字，也就是值，作为集合的元素
  - set：元素有序，每个元素在集合中最多只能出现一次
  - multiset：元素有序，每个元素在集合中可以出现多次
  - unordered\_set：元素不排序
- 映射：存储值和相应的关键字，键值对作为映射的元素
  - map：元素有序，每个关键字在映射中最多只能出现一次
  - multimap：元素有序，每个关键字在映射中可以出现多次
  - unordered\_map：元素不排序

# 关联容器

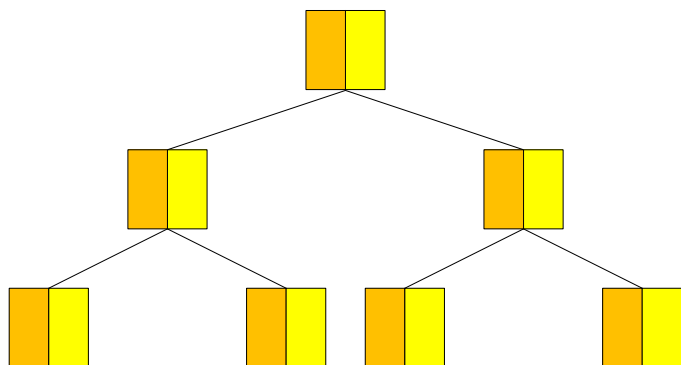
set/multiset



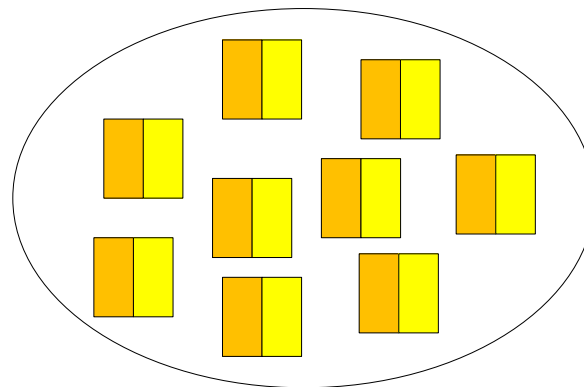
unordered\_set



map/multimap



unordered\_map



# 集合

集合的存储方式是一棵红黑树，每个节点都包含着一个元素（即是key，也是value），节点之间以某种顺序进行排列（如key的升序或降序）

每个元素在集合中只能出现一次，没有两个不同的元素能够拥有相同的次序。

## 【例9.14】测试集合，分析程序运行结果

```
#include <iostream>
#include <set>           // For the std::set<>
                          container template
using namespace std;

void printSet(const set<int>& my_set)
{
    cout << "There are " << my_set.size() << "
elements in my_set: ";
    for (int element : my_set)
        cout << element << ' ';
    // A set, like all containers, is a range
    cout << endl;
}
```

```
int main()
{
    set<int> my_set;
    // Insert elements 1 through 4 in arbitrary order:
    my_set.insert(1);
    my_set.insert(4);
    my_set.insert(3);
    my_set.insert(3); //The elements 3 and 1 are added twice
    my_set.insert(1);
    my_set.insert(2);
    printSet(my_set);
    cout << "The element 1 occurs " << my_set.count(1) << "
time(s)" << endl;
    my_set.erase(1); // Remove the element 1 once
    printSet(my_set);
    my_set.clear(); // Remove all elements
    printSet(my_set);
}
```

## 程序的运行结果：

```
There are 4 elements in my_set: 1 2 3 4
The element 1 occurs 1 time(s)
There are 3 elements in my_set: 2 3 4
There are 0 elements in my_set:
```

# 映射

映射以键/值对（key-value）的方式组织数据，键即关键字，起到索引的作用，值即为关键字所对应的数据值。

## 【例如】

```
map<Type1, Type2> my_map;
```

my\_map就是一个key为Type1类型，value为Type2类型的容器。



# 映射

基于键的查询，能够迅速查找到键相对应的所需的值

- map支持下标运算
- 以“key”为下标，可以获取该key所对应的value

## 【例如】

```
my_map["abc"]=5;
```

即将key “abc” 对应的value值设置为5

```
cout<<my_map["abc"];
```

即将key “abc” 对应的value值输出到屏幕上

# pair类型

键值对<key, value>的类型为pair，是C++标准模板库提供的数据类型

- 类型定义于头文件utility
- 公有数据成员first和second，分别对应key和value
- 公有函数成员make\_pair可以创建pair对象

# Pair类型的主要操作

表达式	含义
<code>pair&lt;T1,T2&gt; p1;</code>	创建一个空的pair对象，它的两个元素分别是T1和T2类型，采用值初始化
<code>pair&lt;T1,T2&gt; p1(v1,v2);</code>	创建一个pair对象，它的两个元素分别是T1和T2类型，其中first成员初始化为v1，second成员初始化为v2。
<code>make_pair(v1,v2)</code>	以v1,v2值创建一个新的pair对象，其元素类型分别是v1，v2类型
<code>p1&lt;p2</code>	两个pair对象之间的小于运算，遵循字典顺序
<code>p1==p2</code>	如果两个pair对象的first和second值依次相等，则它们相等
<code>p.first</code>	返回p中名为first的数据成员
<code>p.second</code>	返回p中名为second的数据成员

## 【例9.15】测试映射，分析程序的运行结果

```
#include <map>
#include <iostream>
#include <string>
int main()
{
    map<string, unsigned long long> phone_book;
    phone_book["Donald Trump"] = 2024561111;
    phone_book["Melania Trump"] = 2024561111;
    phone_book["Francis"] = 39066982;
    phone_book["Elizabeth"] = 4402079304832;
    cout << "The president's number is " <<
phone_book["Donald Trump"] << endl;
    for (const auto& person : phone_book)
        cout << person.first << " can be reached at " <<
person.second << endl;
}
```

## 程序运行结果：

The president's number is 2024561111

Donald Trump can be reached at 2024561111

Elizabeth can be reached at 4402079304832

Francis can be reached at 39066982

Melania Trump can be reached at 2024561111

## 练习9.3

按字典序统计下面这段文字中单词出现的次数

It was the best of times, and it was the worst of times. It was the age of wisdom, and it was the age of foolishness. It was the epoch of belief, and it was the epoch of incredulity. It was the season of light, and it was the season of darkness. It was the spring of hope, and it was the winter of despair. We had everything before us, and we had nothing before us. We were all going direct to Heaven, and we were all going direct the other way. In short, the period was so far like the present period. That some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only.

# 迭代器

**迭代器**（iterator）是STL的一个重要组成部分。在STL中，迭代器如同一个特殊的**指针**（用以指向容器中某个位置的数据元素，也有人据此将之意译为“**泛型指针**”、“指位器”或“游标”），可以用来存取容器内存储的数据。每种容器都定义了自己的迭代器。迭代器和指针很像，功能很像指针，但是实际上，迭代器是通过重载一元的“\*”和“->”来从容器中间接地返回一个值

# 迭代器

不同的容器，STL提供的**迭代器功能**各不相同。对于vector容器，可以使用“+=”、“--”、“++”、“-=”中的任何一种操作符和“<”、“<=”、“>”、“>=”、“==”、“!=”等比较运算符。list容器是一个标准双向链表，其迭代器也是双向的，但不能进行加、减运算，不像vector迭代器那样能够随机访问容器中的数据元素。deque容器的迭代器与vector容器类似，但应用相对较少。



# 迭代器的类别

## 输入迭代器

- 提供对数据的只读访问

## 输出迭代器

- 提供对数据的只写访问

## 前向迭代器

- 提供读写操作，并能一次一个地向前推进迭代器

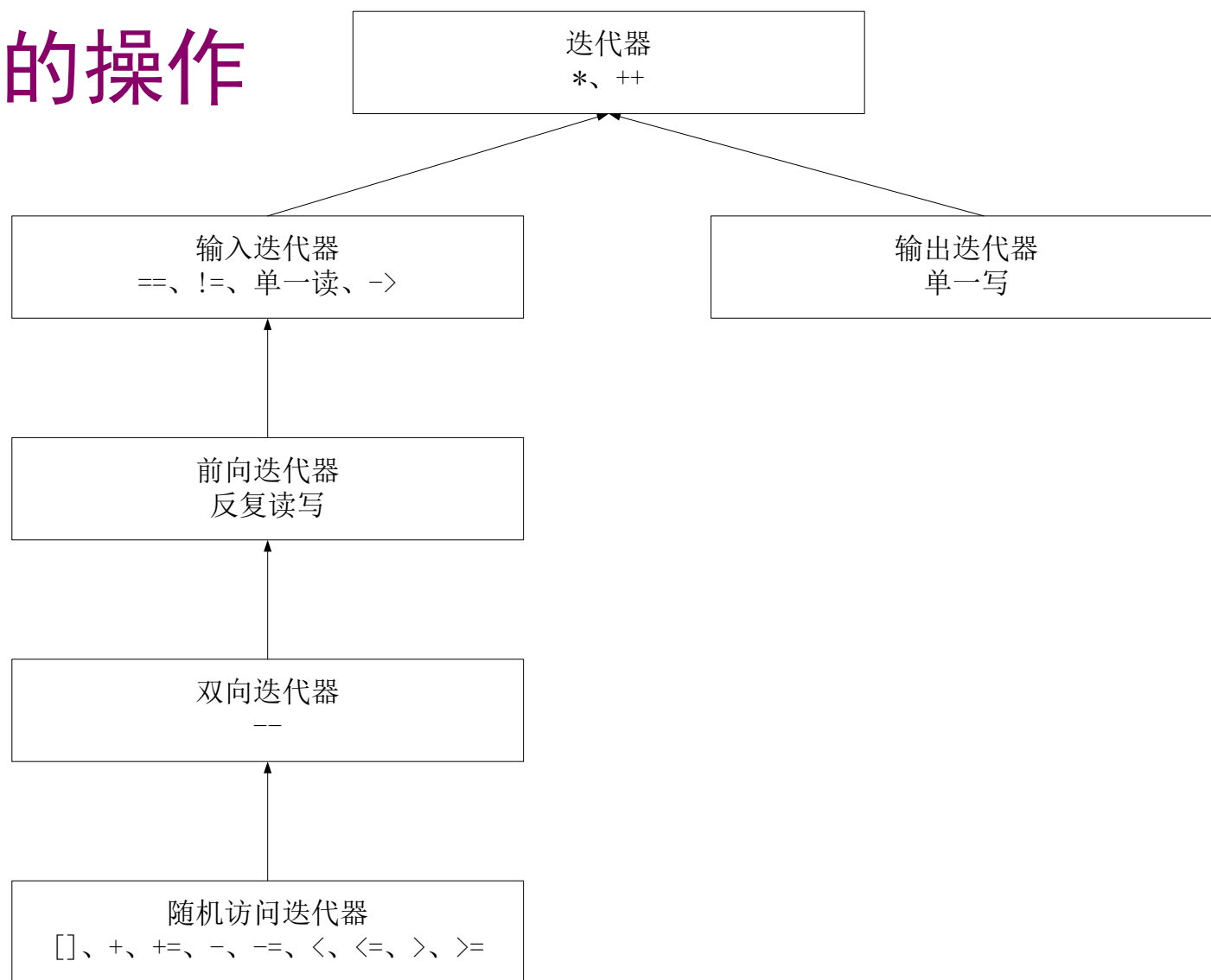
## 双向迭代器

- 提供读写操作，并能一次一个地向前和向后移动

## 随机访问迭代器

- 提供读写操作，并能在数据中随机移动

# 迭代器的操作



# 迭代器与容器

容器	支持的迭代器
array	随机
vector	随机
deque	随机
list	双向
forwad_list	前向
set/multiset	双向
map/multimap	双向
unordered_set/multiset	前向
unordered_map/multimap	前向

## 【例9.16】分析程序运行结果

```
#include <iostream>
#include <vector>
using namespace std;
const int N = 7;

void display(vector<int>& _vec) {
    vector<int>::iterator iter;
    for (iter = _vec.begin(); iter != _vec.end(); iter++)
        cout << (*iter) << " ";
    cout << endl;
}
```

```
int main() {  
    vector<int> invec;  
    int array[N] = { 2,3,5,2,8,18,4 };  
    invec.insert(invec.begin(), array, array + N);  
    display(invec);  
    vector<int> invec2;  
    invec2.insert(invec2.begin(), invec.begin(),  
invec.begin() + invec.size() / 2);  
    display(invec2); //测试insert  
}
```

程序运行结果：

2 3 5 2 8 18 4

2 3 5

2 3 5 2 8 18 4

## 增加第二个display函数

```
void display2(vector<int>& _vec)
{
    vector<int>::value_type i;
    //value_type是模板的实参类型，这里相当于int i;
    for (i = 0; i < _vec.size(); i++)
        cout << _vec[i] << " ";
    cout << endl;
}
```

调用display2(invec)的运行结果仍为：

2 3 5 2 8 18 4

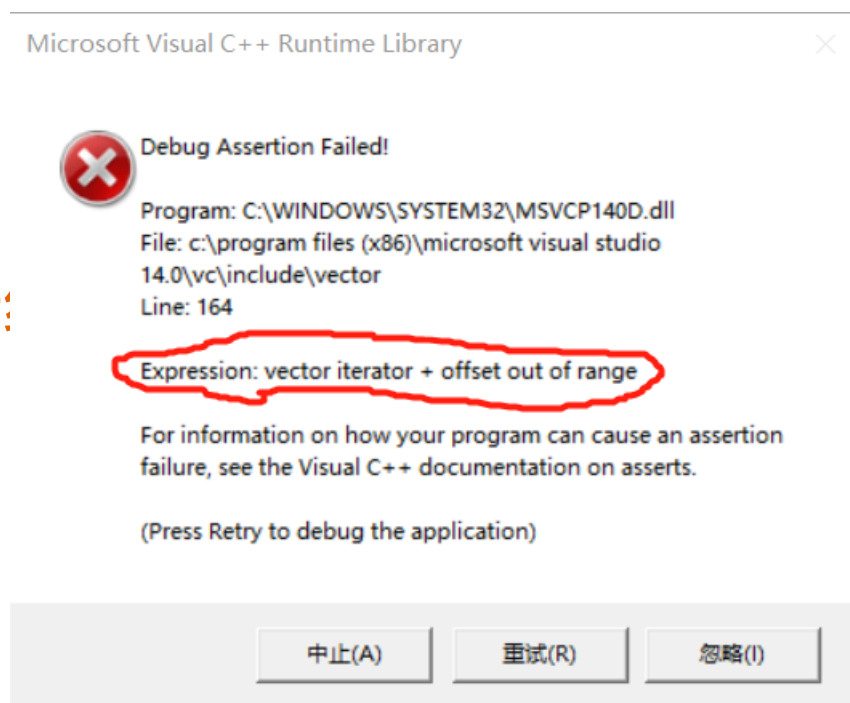
## 增加第三个display函数

```
void display3( vector<int>& _vec)
{
    vector<int>::iterator iter = _vec.begin();
    while ( iter < _vec.end() ) {
        cout << *iter << " ";
        iter = iter + 2;
    }
    cout << endl;
}
```

调用display2(invec)的运行:

2 5 8 4

但是会报迭代器溢出错误!



## 【例9.17】链表使用迭代器

```
#include <iostream>
#include <list>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
    srand((unsigned)time(NULL));
    list<int> numbers;
    int number;
    do{
        number = rand() % 100;
        numbers.push_back(number);
    } while (number != 1);
    for (auto iter{ numbers.begin() }; iter != numbers.end(); ++iter){
        cout << *iter << ' ';
    }
    cout << endl
}
```



## 【例9.18】使用迭代器插入和删除容器元素

```
#include <iostream>
#include <vector>
using namespace std;

void printVector(const vector<int>& v)
{
    for (auto i : v)
        cout << i << ' ';
    cout << endl;
}
```

```
int main()
{
    vector<int> numbers{ 2, 4, 5 };
    numbers.insert(numbers.begin(), 1);
    // Add single element to the beginning of the sequence
    printVector(numbers); //输出 1 2 4 5
    numbers.insert(numbers.begin() + numbers.size() / 2, 3);
    // Add in the middle of the sequence
    printVector(numbers); //输出 1 2 3 4 5
    vector<int> more_numbers{ 6, 7, 8 };
    numbers.insert(numbers.end(), more_numbers.begin(),
more_numbers.end());
    printVector(numbers); //输出 1 2 3 4 5 6 7 8
    numbers.erase(numbers.end() - 3, numbers.end());
    // Erase last 3 elements
    numbers.erase(numbers.begin() + numbers.size() / 2);
    // Erase the middle element
    numbers.erase(numbers.begin());
    // Erase the first element
    printVector(numbers); //输出 2 4 5
}
```

除了标准的迭代器`iterator`外，STL中还有三种迭代器：

- `reverse_iterator`：如果想用向后的方向而不是向前的方向的迭代器来遍历除vector之外的容器中的元素，可以使用`reverse_iterator`来反转遍历的方向，也可以用`rbegin()`来代替`begin()`，用`rend()`代替`end()`，而此时的“++”操作符会朝向后的方向遍历。
- `const_iterator`：一个向前方向的迭代器，它返回一个常数值。可以使用这种类型的游标来指向一个只读的值。
- `const_reverse_iterator`：一个朝反方向遍历的迭代器，它返回一个常数值。

# 迭代器辅助函数

STL为迭代器提供了三个辅助函数：`advance( )`、`distance( )`、`iter_swap( )`。

三个函数的原型如下：

```
void advance (InputIterator& pos, Dist n);
```

- 将迭代器从pos开始移动n个单元

```
dist distance(InputIterator pos1, InputIterator pos2);
```

- 计算迭代器pos1和pos2的距离

```
void iter_swap(ForwardIterator pos1, ForwardIterator  
pos2);
```

- 交换迭代器pos1和pos2指向的元素

## 【9.19】使用迭代器的辅助函数

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
const int N = 7;
int main() {
    list<int> iList;
    int array[N] = { 7,2,6,3,8,10,9 };
    iList.insert(iList.begin(), array, array + N);
    list<int>::iterator it = iList.begin();
    list<int>::iterator itend = iList.end();
    int size = distance(it, itend);
    cout << "the size of the list is:";
    cout << size << endl;
    advance(it, 5);
    cout << "the 6th element is:";
    cout << (*it) << endl;
    iter_swap(it, --iList.end());
    for (it = iList.begin(); it != itend; it++)
        cout << (*it) << " ";
    cout << endl;
```

程序运行结果：

the size of the list is:7  
the 6th element is:10  
7 2 6 3 8 9 10

# 算法

**算法 (algorithm)** 就是一些常用的数据处理方法，如向容器中插入、删除容器中的元素、查找容器中的元素、对容器中的元素排序、复制容器中的元素等等，这些数据处理方法是以函数模板的形式实现的实现的。

算法并非容器的一部分，而是工作在迭代器基础之上，通过迭代器存取容器中的元素，算法并没有和特定的容器进行绑定

# 算法

STL采用C++模板机制实现了算法与数据类型的无关性。

STL实现了**算法与容器（数据结构）的分离**。这样，同一算法适用于不同的容器和数据类型，成为通用性算法，可以最大限度地节省源代码。因此STL比传统的函数库或类库具有更好的代码**重用性**。

# STL算法的分类

第一类非可变序列算法，通常这类算法在对容器进行操作的时候不会改变容器的内容；

第二类是可变序列算法，这类算法一般会改变所操作的容器的内容；

第三类是排序以及相关的算法，包括排序和合并算法、二分查找算法、有序序列的集合操作算法；

第四类算法是通用数值算法。



# 非可变序列算法

非可变序列算法可以修改所操作容器的元素。支持这类算法的迭代器的为输入迭代器和前向迭代器。

循环	<code>for_each()</code>	对序列中的每个元素执行某操作
查找	<code>find()</code>	在序列中找出某个值的第一次出现的位置
	<code>find_if()</code>	在序列中找出符合条件的第一个元素
	<code>find_end()</code>	在序列中找出一子序列的最后一次出现的位置
	<code>find_first_of()</code>	在序列中找出第一次出现指定值集中之值的位置
	<code>adjacent_find()</code>	在序列中找出相邻的一对值
计数	<code>count()</code>	在序列中统计某个值出现的次数
	<code>count_if()</code>	在序列中统计符合某个条件的值出现的次数
比较	<code>mismatch()</code>	找出两个序列相异的第一个元素
	<code>equal()</code>	两个序列中的对应元素都相同时为真
搜索	<code>search()</code>	在序列中找出一子序列的第一次出现的位置
	<code>search_n()</code>	在序列中找出一值的连续n次出现的位置

# find算法

```
template<typename InIt, typename T>
```

```
InIt find(InIt first, InIt last, const T& val);
```

first 和 last 这两个参数都是容器的迭代器，它们给出了容器中的查找区间起点和终点。

- 这个区间是个左闭右开的区间，即区间的起点是位于查找范围之中的，而终点不是

val参数是要查找的元素的值

函数返回值是一个迭代器。如果找到，则该迭代器指向被找到的元素。如果找不到，则该迭代器指向查找区间终点。

## 【例9.20】find算法示例

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
void main() {
    int array[10] = { 10,20,30,40 };
    vector<int> v;
    v.push_back(1);v.push_back(2);
    v.push_back(3);v.push_back(4);
    vector<int>::iterator p;
    p = find(v.begin(), v.end(), 3);
    if (p != v.end())
        cout << *p << endl;
```

```
p = find(v.begin(), v.end(), 9);  
if (p == v.end())  
    cout << "not found " << endl;  
p = find(v.begin() + 1, v.end() - 2, 1);  
if (p != v.end())  
    cout << *p << endl;  
int * pp = find(array, array + 4, 20);  
cout << *pp << endl;  
}
```

程序的运行结果为：

```
3  
not found  
3  
20
```

## 【例9.21】非可变序列算法示例

程序运行结果：

4 7 4 9 3 2 7 8

2

5

```
#include <iostream>
#include <algorithm> //算法需要包含的头文件
#include <functional> //使用函数对象需要包含的头文件
#include <vector>
using namespace std;
int main() {
    int array[] = { 2,4,7,4,9,3,2,7,8 };
    vector<int> ivec(array, array + sizeof(array) / sizeof(int));
    vector<int>::iterator iter;
    //找到第一个为4的元素位置，并输出此值和后面的所有值
    for (iter = find(ivec.begin(), ivec.end(), 4); iter !=
ivec.end(); iter++)
        cout << *iter << " ";
    cout << endl;
    //输出容器中等于4的元素个数
    cout << count(ivec.begin(), ivec.end(), 4) << endl;
    cout << count_if(ivec.begin(), ivec.end(),
bind2nd(less_equal<int>(), 4)) << endl; //输出小于等于4元素的个数
}
```

# 可变序列算法

可变序列算法可以修改他们所操作的容器的元素

复制	<code>copy()</code>	从序列的第一个元素起进行复制
	<code>copy_backward()</code>	从序列的最后一个元素起进行复制
交换	<code>swap()</code>	交换两个元素
	<code>swap_ranges()</code>	交换指定范围的元素
	<code>iter_swap()</code>	交换由迭代器所指的两个元素
变换	<code>transform()</code>	将某操作应用于指定范围的每个元素
替换	<code>replace()</code>	用一个给定值替换一些值
	<code>replace_if()</code>	替换满足条件的一些元素
	<code>replace_copy()</code>	复制序列时用一给定值替换元素
	<code>replace_copy_if()</code>	复制序列时替换满足条件的元素

# 可变序列算法

填充	fill()	用一给定值取代所有元素
	fill_n()	用一给定值取代前n个元素
生成	generate()	用一操作的结果取代所有元素
	generate_n()	用一操作的结果取代前n个元素
删除	remove()	删除具有给定值的元素
	remove_if()	删除满足条件的元素
	remove_copy()	复制序列时删除具有给定值的元素
	remove_copy_if()	复制序列时删除满足条件的元素
剔除	unique()	删除相邻的重复元素
	unique_copy()	复制序列时删除相邻的重复元素
反转	reverse()	反转元素的次序
	reverse_copy()	复制序列时反转元素的次序
循环	rotate()	循环移动元素
	rotate_copy()	复制序列时循环移动元素
随机	random_shuffle()	采用均匀分布来随机移动元素
划分	partition()	将满足某条件的元素都放到前面
	stable_partition()	将满足某条件的元素都放到前面并维持原顺序

# 排序以及相关算法

排序	sort()	以很好的平均效率排序
	stable_sort()	排序，并维持相同元素的原有顺序
	partial_sort()	将区间个数的元素排好序
	partial_sort_copy()	将区间个数的元素排序并复制到别处
第n个元素	nth_element()	将第n各元素放到它的正确位置
二分检索	lower_bound()	找到大于等于某值的第一次出现
	upper_bound()	找到大于某值的第一次出现
	equal_range()	找到（在不破坏顺序的前提下）可插入给定值的最大范围
	binary_search()	在有序序列中确定给定元素是否存在
归并	merge()	归并两个有序序列
	inplace_merge()	归并两个接续的有序序列
有序结构上的集合操作	includes()	一序列为另一序列的子序列时为真
	set_union()	构造两个集合的有序并集
	set_intersection()	构造两个集合的有序交集
	set_difference()	构造两个集合的有序差集
	set_symmetric_difference()	构造两个集合的有序对称差集（并-交）



# 排序以及相关算法

有序结构上的 集合操作	includes()	一序列为另一序列的子序列时为真
	set_union()	构造两个集合的有序并集
	set_intersection()	构造两个集合的有序交集
	set_difference()	构造两个集合的有序差集
	set_symmetric_difference()	构造两个集合的有序对称差集（并-交）
堆操作	push_heap()	向堆中加入元素
	pop_heap()	从堆中弹出元素
	make_heap()	从序列构造堆
	sort_heap()	给堆排序
最大和最小	min()	返回两个元素最小值
	max()	返回两个元素最大值
	min_element()	返回序列中的最小元素的位置
	max_element()	返回序列中的最大元素的位置
词典比较	lexicographical_compare()	两个序列按字典序的第一个在前
排列生成器	next_permutation()	按字典序的下一个排列
	prev_permutation()	按字典序的前一个排列

## 【例9.22】随机产生的10个整数，分别按照由小到大的顺序和由大到小的顺序进行排序

- 利用vector（容器）保存随机数
- 利用iterator（迭代器）进行数据遍历
- 调用sort函数（算法）实现排序，默认为由小到大排序
- 利用greater<>（仿函数）实现由大到小排序

```
#include <vector>           // For vector
#include <algorithm>         // For sort()
#include <functional>        // For greater<int>()
#include <iostream>
#include <iomanip>           // For srand() & rand()
#include <ctime>             // For time()
using namespace std;
```

```
int main()
{
    vector<int> v1; //设置容器, 保存随机数
    vector<int>::iterator Iter1; //设置迭代器
    srand((unsigned)time(NULL));
    for(int i=0; i<=10; i++)
        v1.push_back(rand()%100); //产生随机数并保存
    //开始输出随机数
    cout << " Original vector v1 = ";
    for(Iter1=v1.begin(); Iter1!=v1.end(); Iter1++)
        cout << *Iter1 << " ";
    cout << ")" << endl; //随机数输出完毕
```

```
sort( v1.begin(), v1.end() );//由小到大排序
```

```
//输出排序结果
```

```
cout << "Sorted vector v1 = ( " ;
```

```
for(Iter1=v1.begin();Iter1!=v1.end(); Iter1++)
```

```
    cout << *Iter1 << " ";
```

```
cout << ")" << endl;
```

```
//由大到小排序
```

```
sort(v1.begin(),v1.end(),greater<int>());
```

```
//输出排序结果
```

```
cout << "Resorted (greater) vector v1 = ( " ;
```

```
for(Iter1=v1.begin();Iter1!=v1.end(); Iter1++)
```

```
    cout << *Iter1 << " ";
```

```
cout << ")" << endl;
```

```
return 0;
```

```
}
```

## 程序的运行结果：

C:\Windows\system32\cmd.exe

```
Original vector v1 = ( 25 59 38 43 58 76 95 76 12 99 5 )  
Sorted vector v1 = ( 5 12 25 38 43 58 59 76 76 95 99 )  
Resorted (greater) vector v1 = ( 99 95 76 76 59 58 43 38 25 12 5 )  
请按任意键继续. . .
```

# 数值算法

数值算法包括4个算法，分别为

- accumulate（累积算法）
- partial\_sum（累加部分元素和算法）
- adjacent\_difference（相邻元素差）
- inner\_product（内积算法）

使用数值算法需要包含头文件<numeric>

## 【例9.23】数值算法示例

```
#include<iostream>
#include <vector>
#include <numeric> //所需要包含的头文件
using namespace std;
const int n = 6;
int main() {
    int array[n] = { 2,2,1,5,3,6 };
    vector<int> ivec1(array, array + n);
    vector<int> ivec2(ivec1);
    //对序列进行求和
    cout << accumulate(ivec1.begin(), ivec1.end(), 0) << endl;
    //对两个向量做内积
    cout << inner_product(ivec1.begin(), ivec1.end(),
    ivec2.begin(), 0) << endl;
}
```

程序的运行结果：

19

79



# 适配器

适配器 在STL中扮演着转换器的角色，本质上是一种**设计模式**，用于将一种接口转换成另一种接口，从而是原本不兼容的接口能够很好地一起运作。适配器**不提供**迭代器。

根据目标接口的类型，适配器可分为以下几类：

- 改变容器的接口，称为容器适配器；
- 改变迭代器的接口，称为迭代器适配器；
- 改变仿函数的接口，称为仿函数适配器。





# 容器适配器

是通过修改调整容器的接口，使得容器适用于另一种不同效果。

- 封装5种顺序容器之一
- 使用该容器实现一组特定的、非常有限的成员函数

修改顺序容器接口的容器适配器有stack和queue，其中stack是具有后进先出特性的访问受限的线性结构，而queue是具有先进先出特性的访问受限的线性结构，此外还有优先队列priority\_queue



# 栈

stack（栈）是一种容器适配器，它不是独立的容器，只是某种顺序容器的变化，它提供原容器的一个专用的受限接口。

栈是具有“后进先出”（LIFO）的语义，缺省的stack类（定义在<stack>头文件中），是对deque（双端队列）的一种限制。



## 【例9.24】简单的栈应用

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
    stack<int> sk1;
    for (int i = 0; i < 10; i++)
        sk1.push(i);
    cout << "pop from the stack:";
    while (!sk1.empty()) {
        cout << sk1.top() << " ";
        sk1.pop();
    }
    cout << endl;
}
```



# 队列

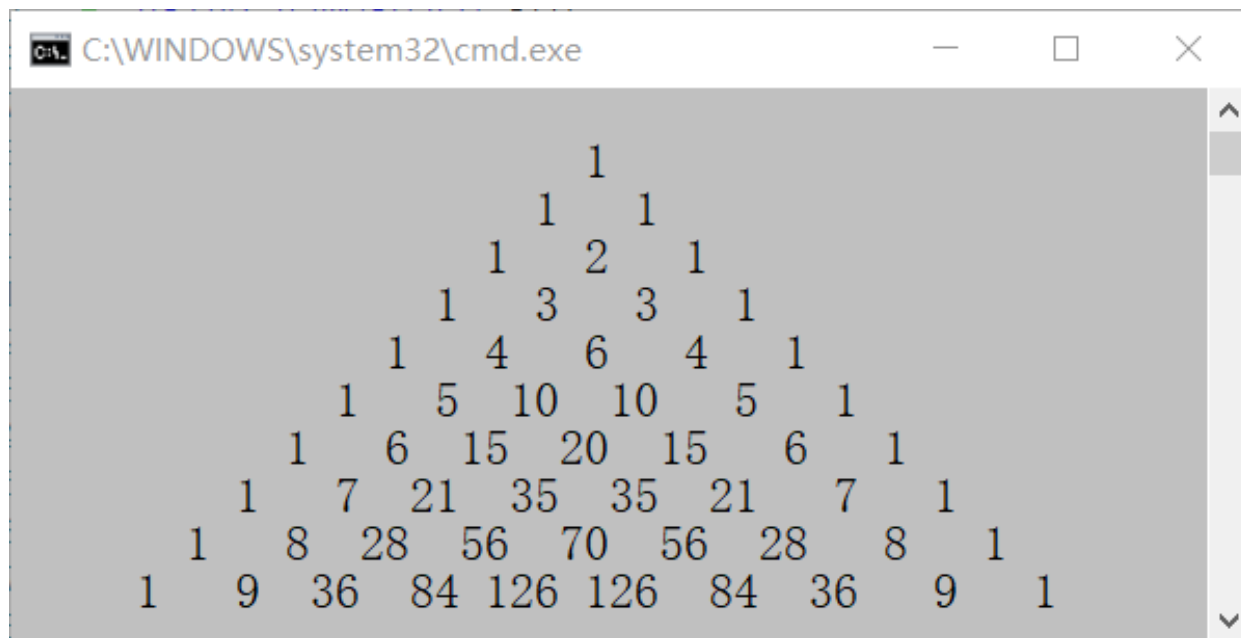
queue也是一种容器适配器，默认通过deque来实现队列，提供了如push，pop等成员函数，还包括测试队列的使用情况，元素个数，是否为空等等功能。

队列具有“先进先出”（FIFO）的语义

## 【9.25】基于队列，实现杨辉三角输出

```
#include <iomanip>
#include <queue>
using namespace std;
void Yanghui(int);
```

```
int main()
{
    Yanghui(10);
}
```



```
C:\WINDOWS\system32\cmd.exe

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
```



```
void Yanghui(int n) { //输出杨辉三角，n为行数
    queue<int> q;
    q.push(1); //首先在队列中存第一行元素1
    int s = 0;
    for (int i = 0; i <= n; i++) {
        cout << endl; //对于每一行输出换行
        cout << setw((n - i) * 2) << " "; //设置输出格式
        q.push(0); //在每一行数据中间添加0
        for (int j = 1; j <= i + 2; j++) {
            //对于每一行的输出i+2项元素
            int t = q.front(); //获取队首元素
            q.pop(); q.push(s + t); //保存两项之和
            s = t;
            if (j != i + 2)
                cout << setw(4) << s; //不打印i+2项的0
        }
    }
    cout << endl;
}
```

# 第九章 结束

