



第八章 类属机制 - 模板



主要内容

- 类属（泛型）的基本概念
- 函数模板
- 类模板
- 模板的复用
- C++标准模板库简介

类属（泛型）程序设计

- 在程序设计中，经常需要用到一些功能完全相同的程序实体（函数、类等），但它们所涉及的数据类型不同。

- 对不同元素类型的数组进行排序的函数：

```
void int_sort(int x[],int num);
```

```
void double_sort(double x[],int num);
```

```
void A_sort(A x[],int num);
```

- 存放不同类型元素的栈

```
class IntStack
```

```
class DoubleStack
```

```
class AStack
```



类属（泛型）程序设计

```
class IntStack
```

```
{ int buf[100];
```

```
    public:
```

```
        void push(int);
```

```
        void pop(int&);
```

```
};
```

```
class DoubleStack
```

```
{ double buf[100];
```

```
    public:
```

```
        void push(int);
```

```
        void pop(int&);
```

```
};
```

```
class AStack
```

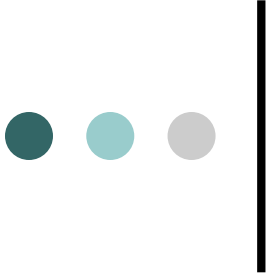
```
{ A buf[100];
```

```
    public:
```

```
        void push(int);
```

```
        void pop(int&);
```

```
};
```

- 
- 在程序设计中，一个程序实体能对多种类型的数据进行操作或描述的特性称为类属性（generics）。
 - 具有类属性的程序实体通常有：
 - 类属函数
 - 类属类
 - 基于具有类属性的程序实体进行程序设计的技术称为：类属程序设计（或泛型程序设计，Generic Programming）。



类属函数

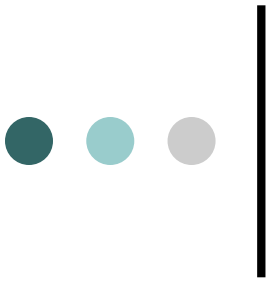
- 类属函数是指一个函数能对不同类型的参数完成相同的操作。
- C++提供了两种实现类属函数的机制：
 - 采用通用指针类型的参数和函数指针
 - 函数模板

函数模板 (function template)

- 函数模板是指带有类型参数的函数定义，其格式为：

```
template <class T1, class T2, ...>  
<返回值类型> <函数名>(<参数表>)  
{ .....  
}
```

其中，T1、T2等是函数模板的类型参数；<返回值类型>、<参数表>中的参数类型以及函数体中的局部变量的类型可以是T1、T2等



```
template <class T>
```

```
void sort(T elements[], unsigned int count)
```

```
{ 1、取第i个元素
```

```
    elements[i]
```

```
2、比较第i个和第j个元素的大小
```

```
    elements[i] < elements[j]
```

```
3、交换第i个和第j个元素
```

```
    T temp = elements[i];
```

```
    elements[i] = elements[j];
```

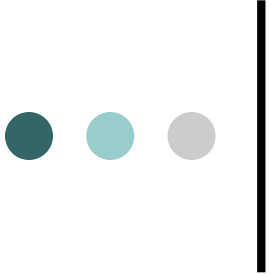
```
    elements[j] = temp;
```

```
}
```


函数模板实例化

- 函数模板定义了一系列重载的函数。使用前需要对函数模板进行实例化，即生成具体的函数（称为模板函数）
- 函数模板的实例化通常是隐式的，由编译程序根据实参的类型自动地把函数模板实例化为具体的函数。

```
int a[100];  
sort(a,100); //生成并调用void sort(int elements[], unsigned int count)  
double b[200];  
sort(b,200); //生成并调用void sort(double elements[], unsigned int count)  
A c[300]; //需重载操作符<, 必要时需自定义拷贝构造函数和=。  
sort(c,300); //生成并调用void sort(A elements[], unsigned int count)
```

- 
- 有时，编译程序无法根据调用时的实参类型来确定所调用的模板函数，这时，需要向函数模板提供实参来**显式地**实例化函数模板。

例如：

```
int a[100];
```

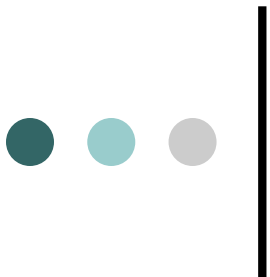
```
sort<int>(a,100);
```

```
double b[200];
```

```
sort<double>(b,200);
```

```
A c[300];
```

```
sort<A>(c,300);
```



○ 例如：

```
template <class T>
```

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

```
int x, y, z;
```

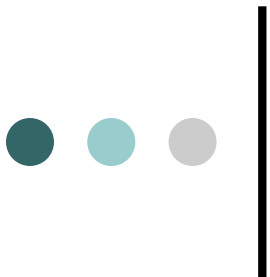
```
double l, m, n;
```

```
.....
```

```
z = max(x, y);    //隐式调用int max(int a, int b)
```

```
l = max(m, n);    //隐式调用double max(double a, double b)
```

```
max(x, m);        //问题： 调用哪一个模板函数？
```



○ 解决方法:

1. 显式实例化:

`max <double> (x, m) //或 max <int> (x, m);`

2. 显式类型转换

`max((double)x, m) //或max(x, (int)m);`

3. 再定义一个max的重载函数

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



- 除了类型参数外，函数模板也可以带有非类型参数。

例如：

```
template <class T, int size> //size为一个int型的普通参数。
```

```
void f(T a)
```

```
{   T temp[size];
```

```
.....
```

```
}
```

```
.....
```

```
f<int,10>(1); //调用模板函数f(int a)，该函数体中的size为10
```

。



类模板

- 如果一个类定义中用到的类型可变但操作不变，则该
类称为类属类。类属类一般用类模板实现。
- 类模板的格式为：
template <class T1,class T2,...>
class <类名>
{ <类成员说明>
}
其中，T1、T2等为类模板的类型参数，在类成员（
数据成员和成员函数）的说明中可以用T1、T2等
来说明它们的类型。



类模板

```
template <class T>
```

```
class Stack
```

```
{    T buffer[100];
```

```
    int top;
```

```
    public:
```

```
        Stack() { top = -1; }
```

```
        bool push(const T &x);
```

```
        bool pop(T &x);
```

```
};
```

```
template <class T>
```

```
bool Stack <T>::push(const T &x) { ..... }
```

```
template <class T>
```

```
bool Stack <T>::pop(T &x) { ..... }
```

类模板的使用 - 实例化

- 类模板定义了若干个类，在使用这些类之前，编译程序将会对类模板进行**显式地实例化**。

例如：

```
Stack<int> st1; //创建一个元素为int型的栈对象。
```

```
int x;
```

```
st1.push(10); st1.pop(x);
```

```
Stack<double> st2; //创建一个元素为double型的栈对象。
```

```
double y;
```

```
st2.push(1.2); st2.pop(y);
```

```
Stack<A> st3; //创建一个元素为A类型的栈对象。
```

```
A a,b;
```

```
st3.push(a); st3.pop(b);
```


- 
- 类模板中的静态成员仅属于实例化后的类（模板类），不同模板类之间不共享静态成员。

例如：

```
template <class T>
```

```
class A
```

```
{  static int x;
```

```
    T y;  .....
```

```
};
```

```
template <class T> int A<T>::x=0;
```

```
.....
```

```
A<int> a1,a2; //a1和a2共享一个x。
```

```
A<double> a3,a4; //a3和a4共享另一个x。
```

- 
- 除了类型参数外，类模板也可以包括非类型参数。

例如： `template <class T, int size>`

```
class Stack
```

```
{    T buffer[size];    int top;
```

```
    public:
```

```
        Stack() { top = -1; }
```

```
        bool push(const T &x);    bool pop(T &x);
```

```
};
```

```
template <class T, int size>
```

```
bool Stack <T, size>::push(const T &x) { ..... }
```

```
template <class T, int size>
```

```
bool Stack <T, size>::pop(T &x) { ..... }
```

```
Stack<int, 100> st1; //st1为元素个数最多为100的int型栈
```

```
Stack<int, 200> st2; //st2为元素个数最多为200的int型栈
```



模板的复用

- 模板机制是一种特殊的多态（函数重载、虚函数等），称为参数化多态。
- 模板是带有**类型参数**的代码，给该参数提供不同的类型就能得到多个不同的代码，一段代码有多种解释。
- **实例化**：类模板=>模板类，在编译时进行，必须要见到相应的源代码。
- 函数模板的实例化，可以是隐式的，也可以是显式的。
- 类模板的实例化，必须是显式的。

模板的复用

- 一个模板可以有很多的实例。是否实例化模板的某个实例要由使用情况来决定。 .cpp文件是分开编译的。

例如: // main.cpp

```
#include "file.h"
```

```
int main()
```

```
{ S<float> s1; //实例化S<float>, 并创建该类的一个对象s1。
```

```
  s1.f(); //调用void S<float>::f()
```

```
  S<int> s2; //实例化 "S<int>" 并创建该类的一个对象s2。
```

```
  s2.f(); //Error, "void S<int>::f()" 不存在。
```

```
  sub();
```

```
  return 0;
```

```
}
```

```
// file.h
template <class T>
class S //类模板S的定义
{
    T a;
    public:
        void f();
};
extern void sub();

// file.cpp
#include "file.h"
template <class T>
void S<T>::f() //类模板s的实现
{ .....
}

void sub()
{ S<float> x; //实例化 “S<float>” 并创建该类的一个对象x。
  x.f(); //实例化 “ void S<float>::f() ” 并调用之。 }
```

- 解决上述问题的通常做法是把模板的定义和实现都放在头文件中，使用时包含该头文件。

```
// file.h
```

```
template <class T>
```

```
class S //类模板s的定义
```

```
{    T a;
```

```
    public:
```

```
        void f();
```

```
};
```

```
template <class T>
```

```
void S<T>::f() //类模板s的实现
```

```
{ .....
```

```
}
```

```
extern void sub();
```

```
// main.cpp  
#include "file.h"  
int main()  
{ S<float> s1; //实例化 “S<float>” 并创建该类的一个对象s1。  
  s1.f(); //实例化： void S<float>::f()并调用之。  
  S<int> s2; //实例化 “S<int>” 并创建该类的一个对象s2。  
  s2.f(); //实例化： void S<int>::f()并调用之。  
  sub();  
  return 0;  
}
```

```
// file.cpp  
#include "file.h"  
void sub()  
{ S<float> x; //实例化 “S<float>” 并创建该类的一个对象x。  
  x.f(); //实例化： void S<float>::f()并调用之。  
}
```



C++标准模板库 (STL)

- C++标准库提供的功能强的**标准模板库** (Standard Template Library, 简称STL)。
 - **容器类模板**。用于存储数据元素，它是元素的序列（如：集合类模板、栈类模板等）。
 - **迭代器类模板**。实现了抽象的指针功能，用于对容器中的数据元素进行遍历和访问。
 - **算法（函数）模板**。用于对容器进行操作（如：排序函数模板、查找函数模板等）。


```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
using namespace std;
int main()
{  vector<int> v; //创建容器对象v
   //生成容器v中的元素
   int x;
   cin >> x;
   while (x > 0)
   {  v.push_back(x); //往容器v中增加一个元素
      cin >> x;
   }
   //创建容器v的一个迭代器it1并使其指向v中的第一个元素位置
   vector<int>::iterator it1 = v.begin();
```

//创建容器v的一个迭代器it2并使其指向v中的最后一个元素的下一个位置

vector<int>::iterator it2 = v.end();

//计算并输出容器v中的最大元素

cout << "Max = " << *max_element(it1,it2) << endl;

//计算并输出容器v中所有元素的和

cout << "Sum = " << accumulate(it1,it2, 0) << endl;

//对容器v中的元素进行排序

sort(it1,it2);

//输出排序结果

cout << "Sorted result is:\n";

while (it1 != it2)

{ cout << *it1 << ' ';

++it1;

}

cout << '\n';

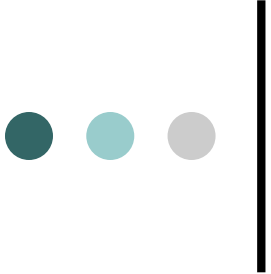
return 0;

}



容器 (container)

- ❑ **容器**是由长度可变的同类型元素所构成的序列。
- ❑ 容器由类模板来实现的，模板的参数是容器的元素类型。
- ❑ STL中包含了很多种容器，如：vector、list等。
 - ❑ 这些容器提供了很多相同的操作。
 - ❑ 由于它们采用了不同的内部实现方法，因此，不同的容器往往适合于不同的应用场合。



STL的主要容器

○ **vector<元素类型>**:

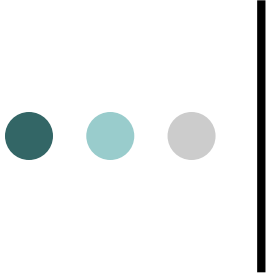
- 用于需要快速定位（访问）任意位置上的元素以及主要在元素序列的尾部增加/删除元素的场合。
- 在头文件vector中定义，用动态数组实现。

○ **list<元素类型>**:

- 用于经常在元素序列中任意位置上插入/删除元素的场合。
- 在头文件list中定义，用双向链表实现。

○ **deque<元素类型>**:

- 用于主要在元素序列的两端增加/删除元素以及需要快速定位（访问）任意位置上的元素的场合。
- 在头文件deque中定义，用分段的连续空间结构实现。



STL的主要容器

○ **stack<元素类型>**:

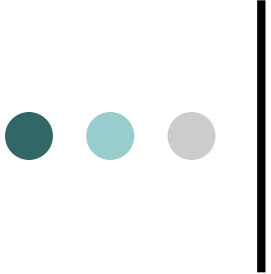
- 用于仅在元素序列的尾部增加/删除元素的场合。
- 在头文件stack中定义，一般基于deque来实现。

○ **queue<元素类型>**:

- 用于仅在元素序列的尾部增加、头部删除元素的场合。
- 在头文件queue中定义，一般基于deque来实现。

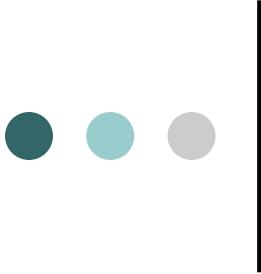
○ **priority_queue<元素类型>**:

- 它与queue的不同之处在于：每次增加元素之后，将对元素的位置进行调整，使得头部元素总是最大的。即，每次删除的总是最大（优先级最高）的元素。
- 在头文件queue中定义，一般基于vector和heap结构来实现。



STL的主要容器

- **map<关键字类型, 值类型>**,
- **multimap<关键字类型, 值类型>**:
 - 容器中每个元素由<关键字, 值>构成 (属于一种pair结构类型, 该结构有两个成员: first和second, 关键字对应first成员, 值对应second成员), 元素是根据其关键字排序的, 用于需要根据关键字来访问元素的场合。
 - 对于map, 不同元素的关键字不能相同;
对于multimap, 不同元素的关键字可以相同。
 - 它们在头文件map中定义, 常常用某种二叉树来实现。
- **set<元素类型>和multiset<元素类型>**:
 - 它们分别是map和multimap的特例。在set和multiset中, 每个元素只有关键字而没有值, 或者说关键字与值合一了。
 - 在头文件set中定义。



STL的主要容器

○ **basic_string<字符类型>:**

- 与vector类似，不同之处在于其元素为字符类型，并提供了一系列与字符串相关的操作。
- string和wstring分别是它的两个实例：
basic_string<char>和basic_string<wchar_t>。
- 在头文件string中定义。



容器的操作（成员函数）

- 上述容器类模板提供了一些公共的操作（成员函数），其中包括：
 - 往容器中增加元素
 - 从容器中删除元素
 - 获取指定位置的元素
 - 在容器中查找元素
 - 获取容器首/尾元素的迭代器
 -



容器的操作（成员函数）

- **T& front(); 和 T& back();**

- 分别用于获取容器中第一个、最后一个元素的引用。
- 适用于vector、list、deque和queue。

- **void push_front(const T& x); 和 void pop_front();**

- 分别在容器的头部增加和删除一个元素。
- 适用于list和deque。

- **void push_back(const T& x); 和 void pop_back();**

- 分别在容器的尾部增加和删除一个元素。
- 适用于vector、list和deque。

- **iterator begin(); 和 iterator end();**

- 分别用于获取指向容器中第一个元素位置、最后一个元素的下一个位置的迭代器。
- 适用于除queue和stack以外的所有容器。



容器的操作（成员函数）

- **iterator insert(iterator pos,const T& x);**
- **void insert(iterator pos,InputIt first,InputIt last);**
 - 分别用于在容器中的指定位置pos（迭代器）插入一个和多个元素。
 - 适用于vector、list和deque。
- **iterator erase(iterator pos);**
- **iterator erase(iterator first,iterator last);**
 - 分别用于在容器中删除指定位置pos（迭代器）上的一个和某范围内的多个元素。
 - 适用于vector、list、deque、map/multimap、set/multiset以及basic_string。

容器的操作（成员函数）

- **T& operator[](size_type pos);**
 - 获取容器中某位置pos（序号）上的元素。适用于vector、deque和basic_string。
- **ValueType& operator[](const KeyType& key);**
 - 获取容器中某个关键字所关联的值的引用。适用于map。
- **T& at(size_type pos);**
 - 获取容器中某位置pos（序号）上的元素的引用，并进行越界检查。适用于vector、deque和basic_string。
- **iterator find(const T& key);**
 - 根据关键字在容器中查找某个元素，返回指向元素的迭代器（若找到）或最后一个元素的下一个位置（未找到）。适用于map/multimap和set/multiset。

○



容器的操作（成员函数）

- 如果容器的元素类型是一个类，则可能针对该类定义**拷贝构造函数**和**赋值操作符重载函数**，因为在对容器进行的操作中可能会创建新的元素（对象，拷贝构造）或进行元素间的赋值。
- 另外，可能还需要针对元素的类重载**小于操作符**（<），以适应容器的一些操作（如排序）所需的元素比较运算。

8.3.3 迭代器

- ❑ **迭代器** (iterator) 实现了抽象的指针 (智能指针), 它们指向容器中的元素, 用于对容器中的元素进行访问和遍历。
- ❑ 在STL中, 迭代器是作为类模板来实现的, 它们可分为以下几种:
 - **输出迭代器 (output iterator, OutIt)**
 - 用于修改它所指向的容器元素
 - 间接访问操作 (*): `*<输出迭代器> = ...`
 - ++操作
 - **输入迭代器 (input iterator, InIt)**
 - 用于读取它所指向的容器元素
 - 间接访问操作 (*): `... = *<输入迭代器>`
 - 元素成员间接访问 (->)
 - ++、==、!=操作。



迭代器

○ 前向迭代器（**forward iterator, FwdIt**）

- 读取和修改它所指向的容器元素
- 元素间接访问操作（*）和元素成员间接访问操作（->）
- ++、==、!=操作

○ 双向迭代器（**bidirectional iterator, BidIt**）

- 读取/修改它所指向的容器元素
- 元素间接访问操作（*）和元素成员间接访问操作（->）
- ++、--、==、!=操作

○ 随机访问迭代器（**random-access iterator, RanIt**）

- 用于读取/修改它所指向的容器元素
- 元素间接访问操作（*）、元素成员间接访问操作（->）和随机访问元素操作（[]）
- ++、--、+、-、+=、-=、==、!=、<、>、<=、>=操作

迭代器之间的相容关系

- 在需要箭头左边迭代器的地方可以用箭头右边的迭代器去替代。

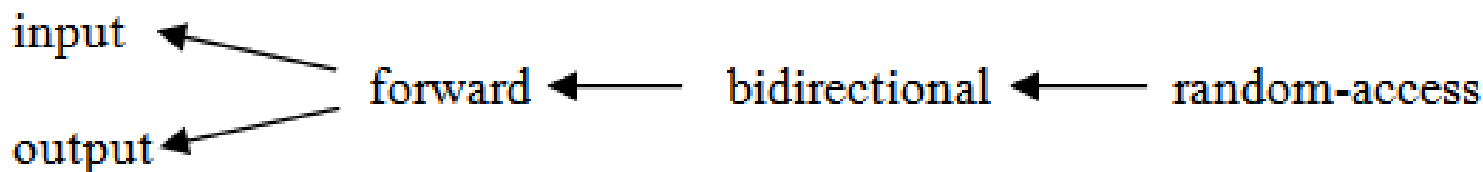


图 9-1 迭代器之间的相容关系



各容器的迭代器类型

- 对于不同的容器，与它们关联的迭代器种类也会有所不同：
 - 对于vector、deque以及basic_string容器类，成员函数begin/end返回的是随机访问迭代器（**RanIt**）。
 - 对于list、queue、stack、map/multimap以及set/multiset，成员函数begin/end则返回的是双向迭代器（**BidIt**）。



例：利用STL的容器map来实现一个电话号码簿的功能

```
#include <iostream>
```

```
#include <map>
```

```
#include <string>
```

```
using namespace std;
```

```
int main() {
```

```
    map<string,int> phone_book; //创建一个map类容器，用于存储电话号码簿
```

```
    //创建电话簿
```

```
    phone_book["wang"] = 12345678; //通过[]操作和关键字往容器中加入元素
```

```
    phone_book["li"] = 87654321;
```

```
    phone_book["zhang"] = 56781234;
```

```
    .....
```

//输出电话号码簿

```
cout << "电话号码簿的信息如下： \n";
```

```
map<string,int>::const_iterator it; //创建一个不能修改所指向的元素的迭代器
```

//遍历容器

```
for (it=phone_book.begin(); it != phone_book.end(); it++)
```

```
    cout << it->first << ": " << it->second << endl; //输出元素的关键字和值
```

//查找某个人的电话号码

```
string name;
```

```
cout << "请输入要查询号码的姓名： ";
```

```
cin >> name;
```

```
it = phone_book.find(name); //查找关键字为name的容器元素
```

```
if (it == phone_book.end()) //判断是否找到
```

```
    cout << name << ": not found\n"; //未找到
```

```
else
```

```
    cout << it->first << ": " << it->second << endl; //找到
```

```
return 0; }
```



例：利用STL的容器list实现求解约瑟夫问题

```
#include <iostream>
```

```
#include <list>
```

```
using namespace std;
```

```
int main() {
```

```
    int m,n; //m用于存储要报的数, n用于存储小孩个数
```

```
    cout << "请输入小孩的个数和要报的数: ";
```

```
    cin >> n >> m;
```

```
    //构建圈子
```

```
    list<int> children; //children是用于存储小孩编号的容器
```

```
    for (int i=0; i<n; i++) //循环创建容器元素
```

```
        children.push_back(i); //把i (小孩的编号) 从容器尾部放入容器
```

//开始报数

```
list<int>::iterator current; //current为指向容器元素的迭代器  
current = children.begin(); //current初始化成指向容器的第一个元素
```

//只要容器元素个数大于1，就执行循环

```
while (children.size() > 1) {  
    for (int count=1; count<m; count++) //报数，循环m-1次  
    {  
        current++; //current指向下一个元素  
        if (current == children.end()) //如果current指向的是容器末尾  
            current = children.begin(); //current指向第一个元素  
    }  
    //循环结束时，current指向将要离开圈子的小孩  
    current = children.erase(current); //小孩离开圈子，current指向下一个元素  
    if (current == children.end()) //如果current指向的是容器末尾  
        current = children.begin(); //current指向第一个元素  
} //循环结束时，current指向容器中剩下的唯一元素，即胜利者，输出其编号  
cout << "The winner is No." << *current << "\n";  
return 0;  
}
```



算法

- 在STL中还提供了**通用算法 (algorithm) 模板**来操作容器中的元素。
 - 调序算法
 - 编辑算法
 - 查找算法
 - 算术算法
 - 集合算法
 - 堆算法
 - 元素遍历算法



算法与容器之间的关系

- ✓ 在STL中，不是把容器传给算法，而是把容器的迭代器传给它们，在算法中通过迭代器来访问和遍历容器中的元素。
- ✓ 不同算法所要求的迭代器种类会有所不同。例如：
 - `void replace(FwdIt first, FwdIt last, const T& val, const T& v_new);`
 - `OutIt copy(InIt src_first, InIt src_last, OutIt dst_first);`



自定义操作条件和操作

- 有些算法要求使用者提供一个函数或函数对象作为自定义的**操作条件**，其参数为元素类型，返回值类型为bool。

```
void sort(RanIt first, RanIt last); //按 “<” 排序
```

```
void sort(RanIt first, RanIt last, BinPred less); //二元谓词  
less
```

例如： `vector<int> v;`

```
bool greater(int x1, int x2) {return x1>x2;}
```

```
sort(v.begin(), v.end(), greater);
```

```
sort(v.begin(), v.end());
```



自定义操作条件和操作

- 有些算法需要使用者提供一个函数或函数对象作为操作，其参数和返回值类型由这些算法决定。例如：

```
T accumulate(InIt first, InIt last, T val); //按 “+” 操作
```

```
T accumulate(InIt first, InIt last, T val, BinOp op); //操作符op
```

例如：

```
Int f (int p, int x) {return p*x;}
```

```
vector<int> v;
```

```
cout << accumulate(v.begin(), v.end(), 1, f);
```

```
cout << accumulate(v.begin(), v.end(), 0);
```




STL算法举例

- 统计容器中满足条件的元素个数：
`size_t count_if(InIt first, InIt last, Pred cond);`
- 对容器中的元素按某条件排序：
`void sort(RanIt first, RanIt last, BinPred less);`



STL算法举例

▣ 学生的类型:

```
class Student {  
    int no;  
    string name;  
    Major major; //enum Major {SOFTWARE, DIGITAL_MEDIA, ... }  
    .....  
public:  
    int get_no() { return no; }  
    string get_name() { return name; }  
    Major get_major() { return major; }  
    .....  
};  
vector<Student> students;
```



STL算法举例

▣ 统计 “软件工程” 专业的人数

```
bool match_major(Student &st) {  
    return st.get_major() == SOFTWARE;  
}  
cout << count_if(students.begin(), students.end(), match_major);
```

▣ 统计 “数字媒体” 专业的人数

```
bool match_major2(Student &st) {  
    return st.get_major() == DIGITAL_MEDIA;  
}  
cout << count_if(students.begin(), students.end(), match_major2);
```



STL算法举例

- 利用函数对象来解决上面的问题

```
class MatchMajor {  
    Major major;  
public:  
    MatchMajor (Major m) {  
        major = m;  
    }  
    bool operator() (Student& st) {  
        return st.get_major() == major;  
    }  
};
```

```
count_if(students.begin(), students.end(), MatchMajor(SOFTWARE));  
count_if(students.begin(), students.end(), MatchMajor(DIGITAL_MEDIA));  
count_if(students.begin(), students.end(), MatchMajor(XXX));
```



STL算法举例

- 容器元素按 “专业” 排序：

```
bool compare_major (Student &st1, Student &st2) {  
    return st1.get_major() < st2.get_major();  
}  
sort(students.begin(), students.end(), compare_major);
```

- 容器元素按 “姓名” 排序呢？

```
bool compare_name (Student &st1, Student &st2) {  
    return st1.get_name() < st2.get_name();  
}  
sort(students.begin(), students.end(), compare_name);
```