《面向对象程序设计》朋辈课程 · 第十二辑

泛型程序设计

信息学院 赵家宇

PART 01

模板

模板

- 假如设计一个求两参数最大值的函数，在实践中我们可能需要定义四个函数，这些函数几乎相同，唯一的区别就是形参类型不同。

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

模板

- 在设计很多具有相近功能的函数/类时，都会出现代码整体几乎相同，仅有部分数据类型不同的情况。

```
int abs(int a)
{
    return a<0?-a:a;
}
double abs(double a)
{
    return a<0?-a:a;
}
```

```
class Square1 {
public:
    int f(){
        return x*x;
    }
private:
     int x;
};
```

```
class Square2 {
public:
    double f(){
        return x*x;
    }
private:
    double x;
};
```

模板

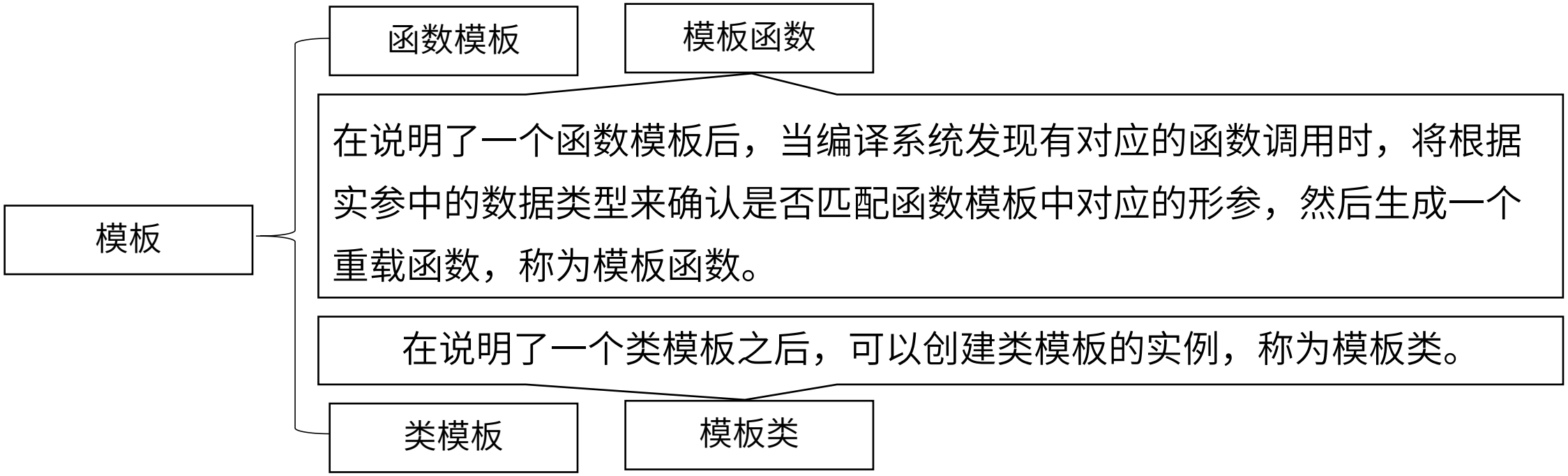
- 模板是一种使用无类型参数来产生一系列函数或类的机制。
- 若一个程序的功能是对某种特定的数据类型进行处理，则可以将所处理的数据类型说明为参数，以便在其他数据类型的情况下使用。
- 模板以一种完全通用的方法来设计函数或类而不必预先说明将被使用的每个对象。通过模板可以产生类或函数的集合，使它们操作不同的数据类型，从而避免需要为每一种数据类型产生一个单独的类或函数。

模板

- 模板并非通常意义上可直接使用的函数或类，它仅仅是对一族函数或类的描述，是参数化的函数和类。
- 模板是一种使用无类型参数来产生一族函数或类的机制。

模板

- 模板分为类模板（class template）和函数模板（function template）两种（从需求出发，抽象成模板，在应用时再具体化）



函数模板与模板函数

- 函数模板是对一批模样相同的函数的说明描述，它不是某一个具体的函数。而模板函数则是将函数模板内的“数据类型参数”具体化后得到的重载函数。
- 函数模板是抽象的，而模板函数则是具体的。实际上，将数据类型作为参数就得到了模板。将参数实例化就得到了模板类或者模板函数。

函数模板与模板函数

- 函数模板的基本格式如下：

```
template <模板形参表>
    <返回值类型> <函数名>（模板函数形参表）{
        //函数定义体
    }
```

- 例如求最大值的函数模板：

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

函数模板与模板函数

- 具体类型代替模板参数的过程称为实例化
  - 将T实例化的参数称为模板实参;
  - 用模板实参实例化的函数称为模板函数;
- 函数模板的实例化通常是隐式进行的。具体：当编译系统发现有一个函数调用：函数名(模板实参表)；时将根据模板实参表中的类型生成一个函数即模板函数。该模板函数的函数体与函数模板的函数定义体相同。

函数模板与模板函数

- 在模板函数的具体生成时，实参到形参类型不能自动转换。

```
template<class T>
    T max(T x, T y){ return(x>y)?x:y; }
int i1=1, i2=2; cout<max(i1,i2)<<endl:
float f1=3.4, f2=5.6; cout<<max(f1,f2)<<endl:
double d1=7.8, d2=9.0; cout<<max(d1,d2)<<endl;
char c1='c', c2='p'; cout<<max(c1,c2)<<endl;
cout<<max(23,-5.6)<<endl; //error
```

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

cout<<max(23,-5.6)<<endl;
    //OK
```

函数模板与模板函数

- 编写一个对具有n个元素的数组a[]求最小值的函数模板。

```
template <class T,class T1>
    T min(T a[], T1 n) {
        T minv=a[0];
        for(int i=1; i<n; i++){
                if(minv>a[i])
                    minv=a[i];
        } return minv;
    }
```

```
int a[]={1,3,0,2,7,6,4,5,2};
double b[]={1.2,-3.4,6.8,9,8};
cout << min(a, 9) << endl;
cout << min(b, 4) << endl;
    //输出结果为:
        0
        -3.4
```

函数模板与模板函数

- 使用函数模板的方法是：先说明函数模板，然后实例化成相应的模板函数，最后才可以调用模板函数，并执行。
- 广泛使用模板，将程序写得尽可能通用，减少了程序员编写代码的工作量。将算法从特定的数据结构中抽象出来成为通用方法，为泛型程序设计奠定了关键的基础 。

函数模板与模板函数

- 除了基本类型外，用户可以在函数模板形参表和对模板函数的调用中使用类类型和其他自定义的类型。如果这样，就可能需要对模板函数中对类对象产生作用的运算符进行重载（使之适应类对象的操作）。

```
class number {
    private: int x, y;
    public:
    int operator> (number& c){
        if (x+y > c.x+c.y) return 1;
        return 0;
    }
};
```

```
template<class obj>
    obj& max(obj& o1,obj& o2) {
        if (o1>o2) return o1;
        return o2;
}
number a(1,2), b(3,4);
max(a, b);
```

函数模板重载

- 定义同名的函数模板与函数，C++允许这种函数模板与函数同名的重载使用方法。在这种情况下每当遇见函数调用时，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。

```
template <class T>
    T min (T a, T b) {
        return ((a<b)?a:b);
    }
char* min (char* a, char* b) {
    return (strcmp(a,b)<0?a:b);
}
```

```
char* str1 = "The C program";
char* str2 = "The C++ program";
min(3, -10);
min(2.5, 99.5);
min('m', 'c'); //生成模板函数并调用
min(str1, str2); //调用重载函数
```

函数模板重载

- 定义两个同名且使用类型参数相同的函数模板，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”。
- 编译器处理函数模板重载的基本方法：首先利用重载函数来寻找完全匹配重载函数，如果没有则利用函数模板来寻找完全匹配项，如没有则报错。

类模板和模板类

- 可以定义用来生成具体类的类模板，然后用这个抽象的类模板来生成具体的模板类。类模板与函数模板类似，将数据类型定义为参数；具体化为模板类后，可以用于生成具体对象。所以类模板描述了代码类似的部分类的集合。
- 模板类的成员函数必须是函数模板。类模板中的成员函数的定义，若放在类模板的定义之中，则与类的成员函数的定义方法相同。

类模板和模板类

- 类模板的基本格式如下：

```
template <模板形参表>
    class   类模板名 {
        成员的声明;
    }
```

- 在类模板之外的成员函数的定义格式如下：

```
template <模板形参表>
    返回值类型   类模板名<类型名表>::成员函数名(参数表) {
        成员函数体
    }
```

类模板和模板类

- 类模板必须用类型参数将其实例化为模板类后，才能用来生成具体对象。
- 当类模板在程序中被引用时，系统根据引用处的参数匹配情况将类模板中的模板参数置换为确定的参数类型，生成一个具体的类。这种由类模板实例化生成的类称为模板类。
- 类模板 → 模板类 → 模板类对象

类模板和模板类

- 类模板实例化为模板类的格式如下：

```
类模板名　<实际类型>;
```

- 定义模板类的对象的格式如下：

```
类模板名  <实际类型>　对象名(实参表);
```

类模板和模板类

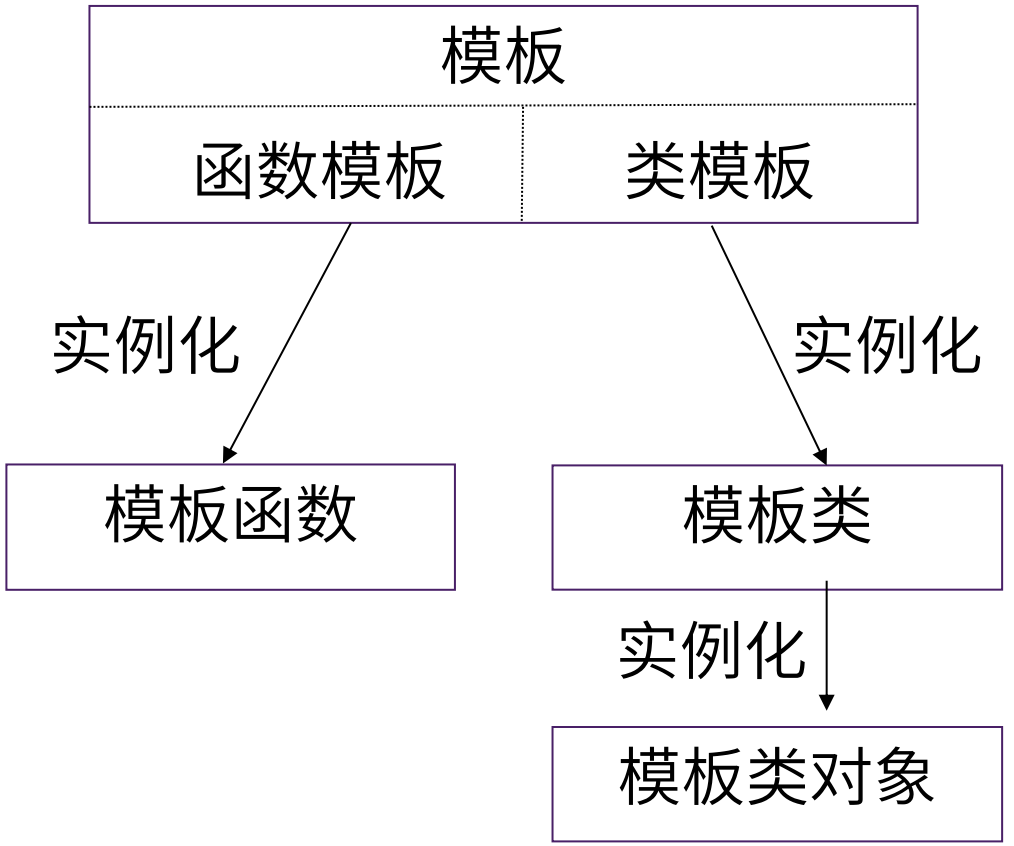
```
template <class T>
    class myclass {
        private: T n;
        public: myclass(T, a);
            T getn();
            void setn(T, b);
};
```

```
template <class T>
    myclass <T>::myclass(T a) { n=a; }
template <class T>
    T myclass <T>::getn(  ) { return n; }
template <class T>
    void myclass <T>::setn(T b) { n=b; }
```

```
myclass <int> obj(10); cout<<obj1.getn()<<endl;
obj1.set(20); cout<<obj1.getn()<<endl;
myclass <char> obj2('a'); cout<<obj2.getn()<<endl;
obj2.setn('b'); cout<<obj2.getn();<<endl;
```

模板

- 模板为程序员提供了一种机制，该机制解决了通用函数或通用类的设计：将数据类型参数化。



PART 02

标准模板库STL

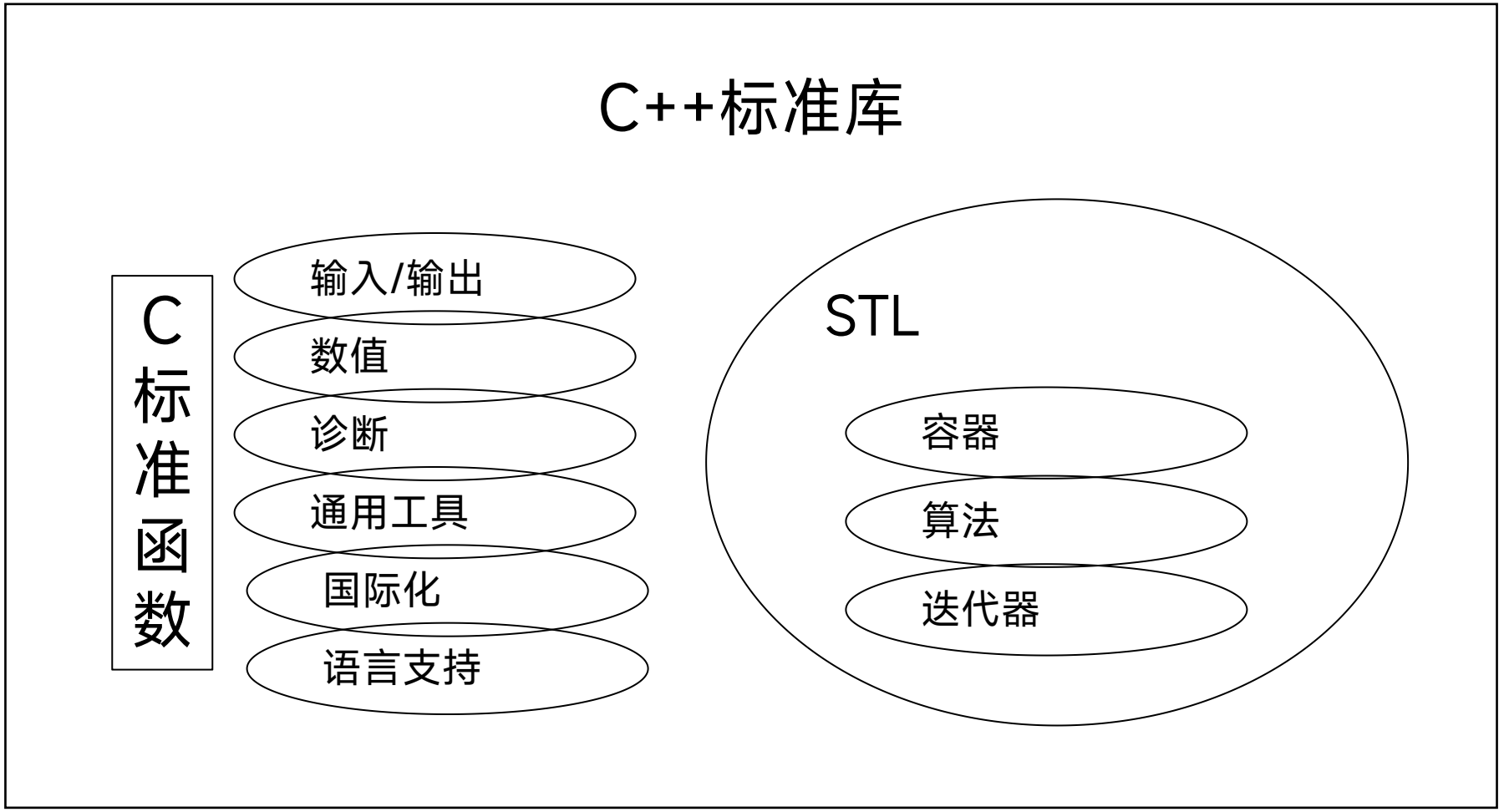
泛型程序设计

- 泛型程序设计是以模板为核心的程序设计法。标准模板库（Standard Template Library）就是一些常用数据结构和算法的模板的集合。
- 将一些常用的数据结构（比如链表，数组，二叉树）和算法（比如排序，查找）写成模板，以后则不论数据结构里放的是什么对象，算法针对什么样的对象，则都不必重新实现数据结构，重新编写算法。有了STL，不必再从头写大多的标准数据结构和算法，并且可获得非常高的性能。

标准模块库STL

- STL是最新的C++标准函数库中的一个子集，占据了整个库的大约80%，是C++标准程序库的核心，是所有C++编译器和所有操作系统平台都支持的一种库；其利用先进、高效的算法来管理数据；
- 在C++标准函数库中，STL主要包含容器、算法、迭代器。

标准模块库STL



标准模块库STL

- 在C++标准函数库中，STL主要包含容器、算法、迭代器。
- 容器：可容纳各种数据类型的数据结构，是由同类型的元素所构成的长度可变的序列，通过类模板实现。
- 迭代器：可依次存取容器中元素，它类似指针。
- 算法：用来操作容器或数组中的元素，如排序、查找等，通过函数模板来实现。函数本身与他们操作的数据的结构和类型无关，因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用。

容器

- 容器由类模板来实现，模板的参数是容器的元素类型。
- 容器分为三大类：
  - 顺序容器：vector、deque、list
  - 关联容器：set、multiset、map、multimap
  - 容器适配器：stack、queue、priority_queue（优先队列）

容器：顺序容器

- 顺序容器中元素的插入位置同元素的值无关。
- vector：一种动态数组。随机存取任何元素都能在常数时间完成。在尾端增删元素具有较佳的性能。头文件 <vector>
- deque：一种动态数组，随机存取任何元素都能在常数时间完成，在两端增删元素具有较佳的性能。头文件 <deque>
- list：双向链表，在任何位置增删元素都能在常数时间完成，不支持随机存取。头文件 <list>

容器：关联式容器

- 关联式容器内元素是有序的，插入任何元素，都按相应的排序准则来确定其位置，在查找时具有非常好的性能。
- set/multiset：集合（multiset允许存在相同元素）头文件<set>
- map/multimap：map中存放的是成对的key/value，并根据key对元素进行排序（multimap允许相同key值）头文件 <map>

容器：容器适配器

- stack：栈。是项的有限序列，并满足序列中被删除、检索和修改的项只能是最近插入序列的项（“先进后出，后进先出”）头文件 <stack>
- queue：队列。只可以在尾部插入，删除、检索和修改只允许从头部进行（“先进先出”）头文件 <queue>
- priority_queue：优先队列。最高优先级元素总是第一个出列头文件。头文件<queue>

容器模板vector

- 头文件：#include <vector>
- vector（容器）类模板是一种更加健壮，且有许多附加功能的数组，其附加功能包括提供下标越界检查、提供数组用相等运算和大小比较、提供数组间赋值等运算。
- vector可以用来存放不同类型的对象。容器中每个对象都有一个对应的整数索引值。在数组生存期内，数组的大小是不会改变的，vector容器则可在运行中动态地增长或缩小；

vector对象的定义和初始化

- vector类提供4种构造函数，用来定义且初始化vector对象。若用T表示数据类型，对象名为v1和v2，则有：vector<vector<int>>

vector<T> v1;	定义容器对象v1,保存类型为T.默认构造函数v1为空.
vector<T> v1(length);	定义length个元素，元素初始值由T来决定.若int,初始为0.
vector<T> v1(length,a);	定义length个元素,元素初始化为a,a的类型为T.
vector<T> v2(v1);	使用已定义的容器构造新容器.要求v2和v1中必须保存同一种元素类型.

vector对象的操作

- vector类对象操作的基本形式：对象名.成员函数

s.empty()	如果s为空，则返回true，否则返回false	
s.size()	返回s中元素个数	
s[n]	返回s中位置为n的元素	
s.push_back(t)	在s的末尾增加一个值为t的元素	
s1=s2	把s1内容替换s2的副本	这在之前数组的操作(整体操作)中是无法实现的！
v1==v2	比较v1和v2内容	
!=,<,<=,>,>=	保持这些操作符惯有含义	

vector对象的操作

- 可以使用下标来访问vector容器中的元素，其仅能对确知已存在的元素进行操作；若事先定义vector容器为空或是较小的长度，则必须通过push_back()来实现元素的添加(实现长度的动态扩展)；
- 赋值运算符=是运算符重载的结果，vector定义的赋值运算符“=”允许同类型的vector对象相互赋值，而不管它们的长度如何；它可以改变赋值目标的大小，使它的元素数目与赋值源的元素数目相同。

vector对象与迭代器iterator

- vector对象的元素除了使用下标来访问，还可以通过迭代器iterator访问元素，迭代器是一种检查容器内元素并遍历元素的数据类型；标准库为每一种标准容器（包括vector）定义了相应的迭代器类型；迭代器类型提供了比下标操作更通用化的方法。
- 迭代器是配套类型访问vector中的元素，可理解为面向对象版本的指针，可作解引用操作来访问元素。

vector使用实例

```
#include <iostream>
#include <vector>     //vector头文件
#include <algorithm> //其他算法头文件
#include <numeric>    //算术算法头文件
using namespace std;
int main(){
    vector<int> v;  //创建容器对象v
    int x; cin >> x;
    while (x>0) {
        v.push_back(x);//v中增加元素
        cin >> x; }
    vector<int>::iterator it1 = v.begin();
    vector<int>::iterator it2 = v.end();
    //创建容器迭代器it1和it2使其指向v中的第一个元素和最后元素的下一个位置
    cout << *max_element(it1, it2) << endl;
    //计算并输出容器v中的最大元素
    cout << accumulate(it1, it2, 0) << endl;
    //计算并输出容器v中所有元素的和
    sort(it1, it2);
    //对容器v中的元素进行排序
    while (it1 != it2) {
        cout << *it1 << ' ';
            ++it1;
    }
    cout << endl;
    return 0;
}
```

PART 03

STL详解

powered by cyr

STL的主要容器

- 创建vector容器

```
std::vector<double> values;
std::vector<int> primes {2, 3, 5, 7, 11, 13, 17};
std::vector<double> values(20);
std::vector<double> values(20, 1.0);
std::vector<char>value1(5, 'c');
std::vector<char>value2(value1);
```

STL的主要容器

- deque容器
- 特点
  - 在两端插入或删除元素快
  - 在中间插入或删除元素慢
  - 随机访问较快，但比向量容器慢
  - 和 vector 不同：deque 可在序列头部添加或删除元素

STL的主要容器

- 创建deque容器（#include <deque>）

```
std::deque<int> d;
```

```
std::deque<int> d(10);
```

```
std::deque<int> d(10, 5);
```

```
std::deque<int> d1(5);
std::deque<int> d2(d1);
```

STL的主要容器

- list容器
  - 特点
    - 在任意位置插入和删除元素都很快
    - 不支持随机访问
    - 又称为双向链表容器
  - 接合（splice）操作：
    - s1.splice(p, s2, q1, q2)：将s2中[q1, q2)移动到s1中p所指向元素之前

STL的主要容器

- 创建list容器（#include <list>）

```
std::list<int> values;
std::list<int> values(10);
std::list<int> values(10, 5);
std::list<int> value1(10);
std::list<int> value2(value1);
```

容器的操作

- 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

- T& front(); 和T& back();
  - 分别获取容器中首尾元素的引用，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

- 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插入元素，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

- 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

- vector容器的成员函数（1/3）

函数成员	函数功能
begin()	返回指向容器中第一个元素的迭代器。
end()	返回指向容器最后一个元素所在位置后一个位置的迭代器，通常和 begin() 结合使用。
rbegin()	返回指向最后一个元素的迭代器。
rend()	返回指向第一个元素所在位置前一个位置的迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。

容器的操作

- vector容器的成员函数（2/3）

size()	返回实际元素个数。
resize()	改变实际元素的个数。
capacity()	返回当前容量。
empty()	判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。
reserve()	增加容器的容量。
shrink _to_fit()	将内存减少到等于当前元素实际所使用的大小。
operator[ ]	重载了[]运算符，可以向访问数组中元素那样，通过下标即可访问元素。
at()	使用经过边界检查的索引访问元素。

容器的操作

- vector容器的成员函数（3/3）

front()	返回第一个元素的引用。
back()	返回最后一个元素的引用。
data()	返回指向容器中第一个元素的指针。
assign()	用新元素替换原有内容。
push_back()	在序列的尾部添加一个元素。
pop_back()	移出序列尾部的元素。
insert()	在指定的位置插入一个或多个元素。
erase()	移出一个元素或一段元素。
clear()	移出所有的元素，容器大小变为 0。

容器的操作

- deque 容器的成员函数（vs. vector）

push_front()	在序列的头部添加一个元素。
pop_front()	移除容器头部的元素。
emplace_front()	在容器头部生成一个元素。和 push_front() 的区别是，该函数直接在容器头部构造元素，省去了复制移动元素的过程。

容器的操作

- list容器的成员函数

函数成员	函数功能
begin()	返回指向容器中第一个元素的迭代器。
end()	返回指向容器最后一个元素所在位置后一个位置的迭代器，通常和 begin() 结合使用。
rbegin()	返回指向最后一个元素的迭代器。
rend()	返回指向第一个元素所在位置前一个位置的迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。

容器的操作

- list容器的成员函数

empty()	判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。
size()	返回当前容器实际包含的元素个数。
max_size()	返回容器所能包含元素个数的最大值。这通常是一个很大的值，一般是 $2^{32}-1$，所以我们很少会用到这个函数。
front()	返回第一个元素的引用。
back()	返回最后一个元素的引用。
assign()	用新元素替换容器中原有内容。

容器的操作

- list容器的成员函数

emplace_front()	在容器头部生成一个元素。该函数和 push_front() 的功能相同，但效率更高。
push_front()	在容器头部插入一个元素。
pop_front()	删除容器头部的一个元素。
emplace_back()	在容器尾部生成一个元素。该函数和 push_back()的功能相同，但效率更高。
push_back()	在容器尾部插入一个元素。
pop_back()	删除容器尾部的一个元素。
emplace()	在容器中的指定位置插入元素。该函数和 insert() 功能相同，但效率更高。
insert()	在容器中的指定位置插入元素。
erase()	删除容器中一个或某区域内的元素。

容器的操作

- list容器的成员函数

swap()	交换两个容器中的元素，必须保证这两个容器中存储的元素类型是相同的。
resize()	调整容器的大小。
clear()	删除容器存储的所有元素。
splice()	将一个 list 容器中的元素插入到另一个容器的指定位置。
remove(val)	删除容器中所有等于 val 的元素。
remove_if()	删除容器中满足条件的元素。
unique()	删除容器中相邻的重复元素，只保留一个。
merge()	合并两个事先已排好序的 list 容器，并且合并之后的 list 容器依然是有序的。
sort()	通过更改容器中元素的位置，将它们进行排序。

容器和迭代器的关系

- 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

- 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。例如：

```
void sort(RanIt first, RanIt last);  //按“<”排序
void sort(RanIt first, RanIt last, BinPred less);  //二元谓词less
```

- 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
```

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);
```

感谢倾听

给个好评！