模板与STL

1. 函数模板

一般格式为：
`template<[模板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}

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
- 自定义数据类型的比较

C++允许函数模板和函数同名的所谓重载使用方法，但是需要注意，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

函数模板的重载：
定义两个参数模板，允许同名，都使用了一个类型参数Type，但两者的形参个数不同。

注意：参数表中允许出现与类型形参Type无关的其他类型的参数，比如int size

2. 类模板的基本概念

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以用于处理多种不同类型的对象（数据）

2.1 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

2.2 类模板的说明

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：

- 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。

2.3 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（参数表）
{函数体};

::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限定符

2.4 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>
利用类模板生产对象：类模板名<具体的实参表>对象名称

2.5 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
}
```

Fence 1

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。

2.6 类模板的友元

类模板的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。
- 更复杂的情况是，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。

2.7 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。

除非重载运算符（特殊情况下）或者在类模板的定义中增加一个“特例版本”的定义，例如

```
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;
}

```

Fence 2

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。
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。

3. 类模板的继承与派生

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:
    ...
};

```

Fence 3

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适用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>”而并非仅使用“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，派生出系的呢类模板，且基类与派生类中均使用同一个类型参数

```
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: ...
};

```

Fence 5

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:
    ...
};
```

Fence 6

4. 标准模板库程序设计

4.1 C++标准库

提供大量的类型（类模板）和函数（函数模板）

例如：`iostream,fstream,string,clock(),pow(),sqrt(),pause()...`

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;`

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（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,multilist
- 主要函数模板:
  sort,copy,search,reverse

4.2 标准模板库程序设计

STL，即标准模板库，是一个高效的C++程序库。STL是ANSI/ISOC++标准库下的一个子集，它提供了大量可拓展的类模板，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MFC。

STL程序设计基本思想

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。

从实现层面看，STL是一种类型参数化的程序设计方法，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。

4.3 泛型编程

泛型即是指具有在多种数据类型上皆可操作的含义，与模板有些类似
泛型编程是实现一个通用的标准容器库
泛型编程让你编写完全一般化并可重复利用的算法，其效率与针对某特定数据类型而设计的算法相同
实现算法与数据的分离

4.4 标准模板库的六大部件

容器、迭代器（iterator）、算法、适配器、分配器、仿函数

- 空间配置器：内存池实现小块内存分配,对应到设计模式--单例模式（工具类，提供服务，一个程序只需要一个空间配置器即可），享元模式（小块内存统一由内存池进行管理，享元模式（Flyweight Pattern）是一种**结构型设计模式**，旨在通过共享技术高效支持大量细粒度对象的复用，从而**减少内存占用、提升系统性能**。其核心思想是将对象状态分为可共享的**内部状态**（Intrinsic State）和不可共享的**外部状态**（Extrinsic State），通过共享内部状态避免重复创建相似对象。）
- 迭代器：迭代器模式，模板方法
  具体：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，我们可以称他为“泛型指针”。STL定义了五种迭代器类型，前向迭代器，双向迭代器，输入迭代器，输出迭代器，随机访问迭代器。
- 容器：STL的核心之一，其他组件围绕容器进行工作：迭代器提供访问方式，空间配置器提供容器内存分配，算法对容器中数据进行处理，仿函数伪算法提供具体的策略，类型萃取，实现对自定义类型内部类型提取。保证算法覆盖性。其中涉及到的设计模式：组合模式（树形结构），门面模式（外部接口提供），适配器模式（stack，queue通过deque适配得到），建造者模式（不同类型树的建立过程）。
  具体：容器是存放其他对象的对象。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；或者存放不同类型的元素或对象时，称为异类容器类。
  对于STL容器库，其包含了两类容器，一种是**顺序容器**（包括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），另一种是**关联容器。**
  顺序容器提供的操作：

push_front()/pop_front()
push_back()/pop_back()
insert()/erase()
front()/back()
operator[]/at()
data()：返回开始位置的指针
容器就是通用的数据结构
数组（array）集合（set/multiset）向量（vector）映射（map/multimap）链表（list,双向链表）无序集合（unordered_set）单向链表（forward_list）无序映射（unordered_map）双端队列（deque）
容器用来装载数据对象
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的逻辑关系，为给定任务选择最适合的容器
容器的通用计算接口：
==,!=,>,<,>=,<=,=
容器的通用迭代器接口：
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
容器的其他接口：
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

- 类型萃取：基于范型编程的内部类型解析，通过typename获取。可以获取迭代器内部类型value_type,Poter,Reference等。
- 仿函数：一种类似于函数指针的可回调机制，用于算法中的决策处理。涉及：策略模式，模板方法。即重载的小括号。
  具体：仿函数也称为函数对象，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：STL中的stack，queue通过双端队列deque适配实现，map，set通过RB-Tree适配实现。涉及适配器模式。
  具体: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
- 算法：算法是STL的核心，可以分为四类：不可变序列算法、可变序列算法、排序及相关算法和算术算法。
- 分配器：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。

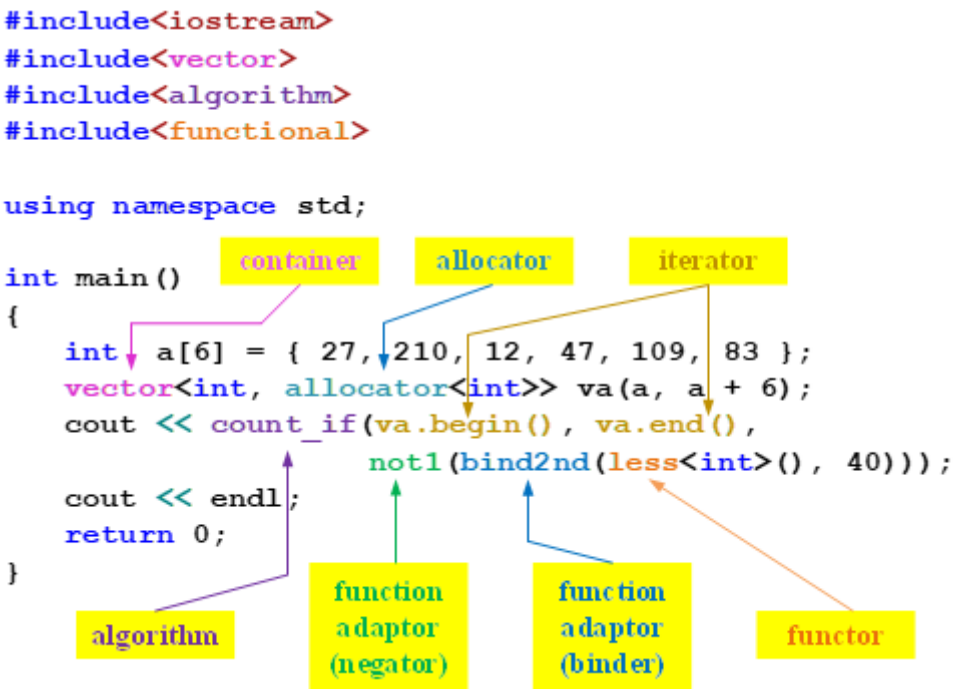

Figure 1

4.5 vector

相当于一个动态数组，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，它分配的空间比当前所需的空间要多一些。

4.6 deque

双端队列是一种增加了访问权限的队列。在队列中（queue），我们只允许从队列的一段添加元素，在队列的另一端提取元素；在双端队列中(deque)，其支持双端的出队和入队。vector与deque同属于随机访问容器，vector拥有的函数deque也都含有，

4.7 list

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：
sort排序，list不支持STL的算法sort（）
remove
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
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

4.8 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，而顺序元素通过位置查找元素。

- 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

4.9 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的升序和降序）
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。

4.10 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。
eg: `map<Type1, Type2>my_map;`
my_map就是一个key为Type1类型，value为Type2类型的容器。
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值所对应的value

4.11 pair类型

键值对<key,value>的类型是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：
pair<T1,T2>p1；
pair<T1,T2>p1(v1,v2);
make_pair(v1,v2);
p1<p2(遵循字典序);
p1==p2;
p.first
p.second

4.12 迭代器

iterator是STL中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，也有人据此将之译为：泛型指针），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'->'来从容器中间间接地返回一个值。

不同的容器，STL提供的迭代器功能不同。

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，并能一次一个地先前和向后移动
- 随机访问迭代器：提供读写操作，并能在数据中随机移动

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。

4.13 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()
分别将迭代器从pos开始移动n个单元、计算迭代器间的距离以及交换爹安蒂奇指向的元素

4.14 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

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。

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。

可变序列算法可以修改他们所操作的容器的元素

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

Figure 2

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

Figure 3

Figure 4

Figure 5

4.15 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。

4.16 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）

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（定义在头文件中），是对deque（双端队列）的一种限制。

- 队列(queue)

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