**STL(Standard Template Library)标准模板库：**

1. 提供了一些常用的数据结构和算法。
2. STL更大的意义在于，它定义了一套概念体系，为泛型程序设计提供了逻辑基础。STL中的各个类模板、函数模板的参数都是用这个体系中的概念来规定的。使用STL的一个模板时所提供的类型参数既可以是C++标准库中已有的类型，也可以是自定义的类型——只要这些类型是所要求概念的模型，因此，STL是一个开放的体系。

**STL六大组件简介(前4种为基本组件)：**

**1.容器（Containers）：**各种数据结构，用来存放数据，STL容器是一种Class Template

(1)序列式

　　向量vector、双端队列deque、列表list

(2)关联式

　　集合set、多重集合multiset、映射map、多重映射multimap

1. **迭代器（Iterators）:**

扮演容器与算法之间的胶合剂，是所谓的“泛型指针”，共有五种类型，以及其它衍生变化，从实现的角度来看，迭代器是一种将：Operators\*,Operator->,Operator++,Operator--等相关操作予以重载的Class Template。所有STL容器都附带有自己专属的迭代器——是的，只有容器设计者才知道如何遍历自己的元素，原生指针（Native pointer）也是一种迭代器。

1. **仿函数（Functors）:**

*仿函数（functor）也称为函数对象（function object）*，行为类似函数，可作为算法的某种策略（Policy）,从实现的角度来看，仿函数是一种重载了Operator()的Class 或 Class Template。一般函数指针可视为狭义的仿函数。

1. **算法（Algorithms）:**

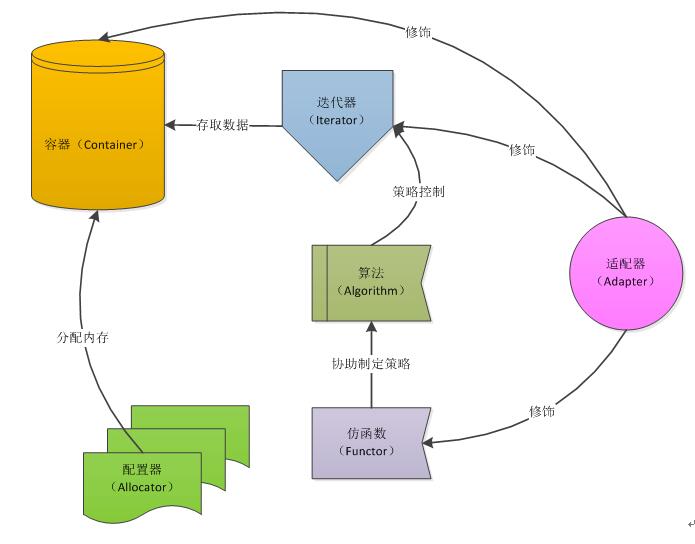
各种常用算法如Sort,Search,Copy,Erase,从实现的角度来看，STL算法是一种Function Templates。

1. **配接器（适配器）（Adapters）:**

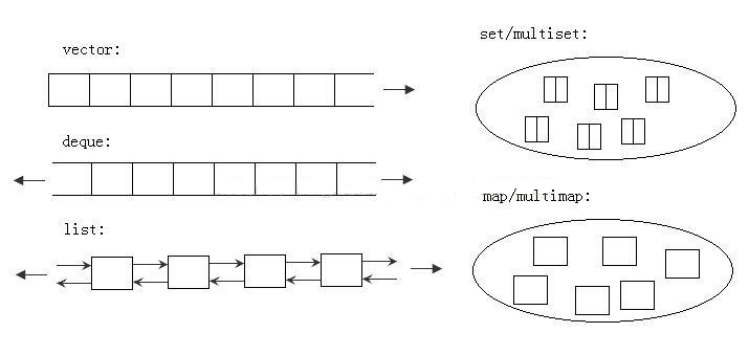
一种用来修饰容器（Containers）或仿函数（Functors）或迭代器（Iterators）接口的东西，例如：STL提供的Queue和Stack，虽然看似容器，其实只能算是一种容器配接器，因为 它们的底部完全借助Deque，所有操作有底层的Deque供应。改变Functor接口者，称为Function Adapter;改变Container接口者，称为Container Adapter;改变Iterator接口者，称为Iterator Adapter。配接器的实现技术很难一言蔽之，必须逐一分析。

1. **分配器（Allocators）:**

负责空间配置与管理，从实现的角度来看，配置器是一个实现了动态空间配置、空间管理、空间释放的Class Template。



这六大组件的交互关系：container（容器） 通过 allocator（配置器） 取得数据储存空间，algorithm（算法）通过 iterator（迭代器）存取 container（容器） 内容，functor（仿函数） 可以协助 algorithm（算法） 完成不同的策略变化，adapter（配接器） 可以修饰或套接 functor（仿函数）。



### **1.vector**

拥有一段连续的内存空间，因此它能非常好的支持随机访问，即[]操作符和.at()，随机访问快。（优点）

总结：相当于可拓展的数组（动态数组），随机访问快，在头部和中间插入或删除效率低，但在尾部插入或删除效率高。

**例子**

以下例子针对整型定义了一个 vector，插入 6 个元素，然后打印所有元素：

#include <iostream>

#include <vector>

using namespace std;

int main(int argc, char\* argv[]){

vector<int> vecTemp;

for (int i = 0; i<6; i++)

vecTemp.push\_back(i);

for (int i = 0; i<vecTemp.size(); i++)

cout << vecTemp[i] <<" "; // 输出：0 1 2 3 4 5

return 0;

}

### **2.list**

List 由双向链表（doubly linked list）实现而成，元素存放在堆中，每个元素都是放在一块内存中。没有空间预留习惯，所以每分配一个元素都会从内存中分配，每删除一个元素都会释放它占用的内存。

**特点**

内存空间可以是不连续的，通过指针来进行数据的访问，这个特点使得它的随机存取变得非常没有效率，因此它没有提供 [] 操作符的重载。（缺点）

由于链表的特点，在任意位置的插入和删除效率都较高。（优点）

只支持首尾两个元素的直接存取，想获取其他元素（访问时间一样），则需要遍历链表。（缺点）

总结：不支持随机访问，在任意位置的插入和删除效率都较高。

**适用场景**

|  |
| --- |
| 适用于经常进行插入和删除操作并且不经常随机访问的场景。 |

**例子**

以下例子产生一个空 list，准备放置字符，然后将 'a' 至 'z' 的所有字符插入其中，利用循环每次打印并移除集合的第一个元素，从而打印出所有元素：

#include <iostream>#include <list>

using namespace std;

int main(int argc, char\* argv[]){

list<char> listTemp;

for (char c = 'a'; c <= 'z'; ++c)

listTemp.push\_back(c);

while (!listTemp.empty())

{

cout <<listTemp.front() << " ";

listTemp.pop\_front();

}

return 0;

}

成员函数empty()的返回值告诉我们容器中是否还有元素，只要这个函数返回 false，循环就继续进行。循环之内，成员函数front()会返回第一个元素，pop\_front()函数会删除第一个元素。

注意：list<指针> 完全是性能最低的做法，还不如直接使用 list<对象> 或使用 vector<指针> 好，因为指针没有构造与析构，也不占用很大内存。

### **3.deque**

deque（double-ended queue）是双向开口的连续内存空间（动态将多个连续空间通过指针数组接合在一起），随时可以增加一段新的空间。deque 的最大任务就是在这些分段的连续空间上，维护其整体连续的假象，并提供随机存取的接口。

**特点**

一旦要在 deque 的头部和尾部增加新空间，便配置一段定量连续空间，串在整个 deque 的头部或尾部，因此不论在头部或尾部插入元素都十分迅速。 (优点）

在中间部分安插元素则比较费时，因为必须移动其它元素。（缺点）

deque 是 list 和 vector 的折中方案。兼有 list 的优点，也有 vector 随机访问效率高的优点。

总结：支持随机访问，但效率没有 vector 高，在头部和尾部插入或删除效率高，但在中间插入或删除效率低。

**适用场景**

|  |
| --- |
| 适用于既要频繁随机访问，又要关心两端数据的插入与删除的场景。 |

**例子**

以下例子声明了一个浮点类型的 deque，并在容器尾部插入 6 个元素，最后打印出所有元素。

#include <iostream>#include <deque>

using namespace std;

int main(int argc, char\* argv[]){

deque<float> dequeTemp;

for (int i = 0; i<6; i++)

dequeTemp.push\_back(i);

for (int i = 0; i<dequeTemp.size(); i++)

cout << dequeTemp[i] << " "; // 输出：0 1 2 3 4 5

return 0;

}

### **4.set**

set（集合）顾名思义，就是数学上的集合——每个元素最多只出现一次，并且 set 中的元素已经从小到大排好序。

**特点**

使用红黑树实现，其内部元素依据其值自动排序，每个元素值只能出现一次，不允许重复。

每次插入值的时候，都需要调整红黑树，效率有一定影响。（缺点）

map 和 set 的插入或删除效率比用其他序列容器高，因为对于关联容器来说，不需要做内存拷贝和内存移动。（优点）

总结：由红黑树实现，其内部元素依据其值自动排序，每个元素值只能出现一次，不允许重复，且插入和删除效率比用其他序列容器高。

**适用场景**

|  |
| --- |
| 适用于经常查找一个元素是否在某集合中且需要排序的场景。 |

**例子**

下面的例子演示 set（集合）的两个特点：

#include <iostream>#include <set>

using namespace std;

int main(int argc, char\* argv[]){

set<int> setTemp;

setTemp.insert(3);

setTemp.insert(1);

setTemp.insert(2);

setTemp.insert(1);

set<int>::iterator it;

for (it = setTemp.begin(); it != setTemp.end(); it++)

{

cout << \*it << " ";

}

return 0;

}

输出结果：1 2 3。一共插入了 4 个数，但是集合中只有 3 个数并且是有序的，可见之前说过的 set 集合的两个特点，有序和不重复。

当set集合中的元素为结构体时，该结构体必须实现运算符 ‘<’ 的重载：

#include <iostream>

#include <set>

#include <string>

using namespace std;

struct People{

string name;

int age;

bool operator <(const People p) const{

return age < p.age;

}

};

int main(int argc, char\* argv[]){

set<People> setTemp;

setTemp.insert({"张三",14});

setTemp.insert({ "李四", 16 });

setTemp.insert({ "隔壁老王", 10 });

set<People>::iterator it;

for (it = setTemp.begin(); it != setTemp.end(); it++){

printf("姓名：%s 年龄：%d\n", (\*it).name.c\_str(), (\*it).age);

}

return 0;

}

/\*

输出结果

姓名：王二麻子 年龄：10

姓名：张三 年龄：14

姓名：李四 年龄：16

\*/

可以看到结果是按照年龄由小到大的顺序排列。另外 string 要使用c\_str()转换一下，否则打印出的是乱码。

**另外 Multiset 和 set 相同，只不过它允许重复元素**，也就是说 multiset 可包括多个数值相同的元素。这里不再做过多介绍。

### **5.map**

map 由红黑树实现，其元素都是 “键值/实值” 所形成的一个对组（key/value pairs)。

map 主要用于资料一对一映射的情况，map 内部自建一颗红黑树，这颗树具有对数据自动排序的功能，所以在 map 内部所有的数据都是有序的。比如一个班级中，每个学生的学号跟他的姓名就存在着一对一映射的关系。

**特点**

每个元素都有一个键，且只能出现一次，不允许重复。

根据 key 值快速查找记录，查找的复杂度基本是 O(logN)，如果有 1000 个记录，二分查找最多查找 10次(1024)。（优点）

每次插入值的时候，都需要调整红黑树，效率有一定影响。（缺点）

增加和删除节点对迭代器的影响很小，除了那个操作节点，对其他的节点都没有什么影响。（优点）

对于迭代器来说，可以修改实值，而不能修改 key。

总结：元素为键值对，key 和 value 可以是任意你需要的类型，每个元素都有一个键，且只能出现一次，不允许重复，根据 key 快速查找记录。

**适用场景**

|  |
| --- |
| 适用于需要存储一个数据字典，并要求方便地根据key找value的场景。 |

**例子**

#include "stdafx.h"

#include <iostream>

#include <map>

#include <string>

using namespace std;

int main(int argc, char\* argv[]){

map<int, string> mapTemp;

mapTemp.insert({ 5,"张三" });

mapTemp.insert({ 3, "李四"});

mapTemp.insert({ 4, "隔壁老王" });

map<int, string>::iterator it;

for (it = mapTemp.begin(); it != mapTemp.end(); it++){

printf("学号：%d 姓名：%s\n", (\*it).first, (\*it).second.c\_str());

}

return 0;

}

/\*

输出结果：

学号：3 姓名：李四

学号：4 姓名：隔壁老王

学号：5 姓名：张三

\*/

**multimap 和 map 相同，但允许重复元素**，也就是说 multimap 可包含多个键值（key）相同的元素。这里不再做过多介绍。

### **6.容器配接器**

除了以上七个基本容器类别，为满足特殊需求，STL还提供了一些特别的（并且预先定义好的）容器配接器，根据基本容器类别实现而成。包括：

**1、stack**

名字说明了一切，stack 容器对元素采取 LIFO（后进先出）的管理策略。

**2、queue**

queue 容器对元素采取 FIFO（先进先出）的管理策略。也就是说，它是个普通的缓冲区（buffer）。

**3、priority\_queue**

priority\_queue 容器中的元素可以拥有不同的优先权。所谓优先权，乃是基于程序员提供的排序准则（缺省使用 operators）而定义。Priority queue 的效果相当于这样一个buffer：“下一元素永远是queue中优先级最高的元素”。如果同时有多个元素具备最髙优先权，则其次序无明确定义。

## **总结**

**各容器的特点总结：**

vector 支持随机访问，在头部和中间插入或删除效率低，但在尾部插入或删除效率高。

支持随机访问，但效率没有 vector 高，在头部和尾部插入或删除效率高，但在中间插入或删除效率低。

list 不支持随机访问，在任意位置的插入和删除效率都较高。

set 由红黑树实现，其内部元素依据其值自动排序，每个元素值只能出现一次，不允许重复，且插入和删除效率比用其他序列容器高。

map 的元素为键值对，key 和 value 可以是任意你需要的类型，每个元素都有一个键，且只能出现一次，不允许重复，根据 key 快速查找记录。

在实际使用过程中，到底选择这几种容器中的哪一个，应该根据遵循以下原则：

|  |
| --- |
| 1、如果需要高效的随机访问，不在乎插入和删除的效率，使用 vector。 2、如果需要随机访问，并且关心两端数据的插入和删除效率，使用 deque。 3、如果需要大量的插入和删除元素，不关心随机访问的效率，使用 list。 4、如果经常查找一个元素是否在某集合中且需要排序，唯一存在的情况使用 set，不唯一存在的情况使用 multiset。 5、如果打算存储数据字典，并且要求方便地根据 key 找到 value，一对一的情况使用 map，一对多的情况使用 multimap。 |

**各容器的时间复杂度分析**

vector 在头部和中间位置插入和删除的时间复杂度为 O(N)，在尾部插入和删除的时间复杂度为 O(1)，查找的时间复杂度为 O(1)；

deque 在中间位置插入和删除的时间复杂度为 O(N)，在头部和尾部插入和删除的时间复杂度为 O(1)，查找的时间复杂度为 O(1)；

list 在任意位置插入和删除的时间复杂度都为 O(1)，查找的时间复杂度为 O(N)；

set 和 map 都是通过红黑树实现，因此插入、删除和查找操作的时间复杂度都是 O(log N)。

**各容器的共性**

各容器一般来说都有下列函数：默认构造函数、复制构造函数、析构函数、empty()、max\_size()、size()、operator=、operator<、operator<=、operator>、operator>=、operator==、operator!=、swap()。

顺序容器和关联容器都共有下列函数：

* begin() ：返回容器第一个元素的迭代器指针；
* end()：返回容器最后一个元素后面一位的迭代器指针；
* rbegin()：返回一个逆向迭代器指针，指向容器最后一个元素；
* rend()：返回一个逆向迭代器指针，指向容器首个元素前面一位；
* clear()：删除容器中的所有的元素；
* erase(it)：删除迭代器指针it处元素。