火龙果•整理
uml.org.cn

目录

1 介绍 3
1.1 动机 3
1.2 STL 历史 3
1.3 STL 和 ANSI/ISO C++草案标准 3
1.4 内容安排 3
2 C++基础 4
2.1 类 4
2.2 函数对象(Function Objects) 5
2.3 模板(Template) 5
2.3.1 函数模板 5
2.3.2 类模板 6
2.3.3 模板特化 7
3 STL 概貌 7
3.1 STL 网上信息 8
3.2 STL 文档 8
3.3 编译 STL 程序 9
4 学习 STL 9
4.1 容器(Container) 9
4.1.1 向量(Vector) 10
4.2 迭代器(Iterator) 12
4.2.1 输入和输出迭代器 12
4.2.2 向前迭代器 13
4.2.3 双向迭代器 14
4.2.4 任意存取迭代器 14
4.3 算法和函数对象 15
4.3.1 如何创建基本算法 15
4.3.2 STL 算法 17
4.4 适应器(Adaptor) 19
4.4.1 容器适应器 19
4.4.2 迭代适应器 20
4.4.3 函数适应器 20
4.5 分配算符和内存处理 20
5 其余的 STL 部件 21
5.1 各部件如何协同工作 21
5.2 向量(Vector) 21
5.3 线性表(List) 22
5.4 双向队列(Deque) 22
5.5 迭代标签(Iterator Tag) 22
5.6 关联容器(Container) 22
6 版权信息 24
7 文献 24
8 后话 25

火龙果•整理
uml.org.cn

1 介绍

1.1 动机

在七十年代末，Alexander Stepanov 第一个发现一些算法不依赖于数据结构的特定实现，而仅仅和结构的一些基本语义属性相关。这些属性表达了一种能力，比如可以从数据结构的一个成员取得下一个成员，从头到尾“走过”结构中的元素〔就象排序算法不关心元素是存放在数组中或是线性表中〕。Stepanov 研究过一些算法可以用一种抽象的方式实现，而且不会影响效率。

1.2 STL 历史

1985 年，Stepanov 开发了基本 Ada 库，有人要求他在 C++中也这样做。但直到 1987 年，模板(Template)在 C++中还未实现，所以他的工作推迟了。1988 年，Stepanov 到 HP 实验室工作，并在 1992 年被任命为一个算法项目的经理。在此项目中，Alexander Stepanov 和 Meng Lee 写了一个巨大的库---标准模板库(STL：Standard Template Library)，意图定义一些通用算法而不影响效率。现在 STL 在国外已经成了新的编程手段。

1.3 STL 和 ANSI/ISO C++草案标准

1994 年 7 月 14 日，ANSI/ISO C++标准化委员会将 STL 采纳为草案标准。现在 Microsoft Visual C++ 5.0 以上及 Borland C++ 4.0 以上都支持 STL。STL 已经并将继续影响软件开发的方法，有了 STL，程序员可以写更少且更快的代码，把精力集中在问题解决上，而不必关心低层的算法和数据结构了。

1.4 内容安排

第 2 部分介绍 STL 需要的 C++基础，主要是类、函数对象和模板。
第 3 部分是概貌，介绍了关键的思想。
第 4 部分 step-by-step 的教 STL。
第 5 部分介绍剩余的 STL 部分。
第 6 部分是版权信息。
第 7 部分是参考文献。

火龙果•整理
uml.org.cn

2 C++基础

2.1 类

请读下面一段代码:

```
class shape
{
private:
	int x_pos;
	int y_pos;
	int color;
public:
	shape() : x_pos(0), y_pos(0), color(1) {}
	shape(int x, int y, int c = 1) : x_pos(x), y_pos(y), color(c) {}
	shape(const shape& s) : x_pos(s.x_pos), y_pos(s.y_pos), color(s.color) {}
	~shape() {}
	shape& operator=(const shape& s)
	{
		x_pos = s.x_pos;
		y_pos = s.y_pos;
		color = s.color;
		return *this;
	}

	int get_x_pos() { return x_pos; }
	int get_y_pos() { return y_pos; }
	int get_color() { return color; }

	void set_x_pos(int x) { x_pos = x; }
	void set_y_pos(int y) { y_pos = y; }
	void set_color(int c) { color = c; }

	virtual void DrawShape() {}

	friend ostream& operator<<(ostream& os, const shape& s);
};
ostream& operator<<(ostream& os, const shape& s)
{
	os << “shape: (“ << s.x_pos << “,” << s.y_pos << “,” << s.color << “)”;
	return os;
}
```

火龙果•整理
uml.org.cn

如果你不能轻松的读懂上面的代码，或者对以下概念：

· 缺省构造函数(default constructor)、拷贝构造函数(copy constructor)、析构函数(destructor)
· 操作符重载(operator overloading)
· 虚函数(virtual function)
· put-to 操作符(operator <<)
· 友元(friend)、inline 函数

不是很熟悉的话，说明你还需要看看 C++的基础书籍。

2.2 函数对象(Function Objects)

所谓函数对象(function object)是定义了函数调用操作符(funiton-call operator，即 operator())的对象。请看下面的例子：

```
class less
{
public:
    less(int v) : val(v) {}
    int operator()(int v)
    {
        return v < val;
    }
private:
    int val;
};
```

声明一个 less 对象：

```
less less_than_five(5);
```

当调用 function-call operator 时，看判断出传入的参数是否小于 val：

```
cout << “2 is less than 5: “ << (less_than_five(2) ? “yes” : “no”);
```

输出是：

```
2 is less than 5: yes
```

函数对象在使用 STL 时非常重要，你需要熟悉这样的编码形式。在使用 STL 时，经常需要把函数对象作为算法的输入参数，或着实例化一个容器(container)时的输入参数。

2.3 模板(Template)

2.3.1 函数模板

请看下面的代码：

```
void swap(int& a, int& b)
{
    int tmp = a;
```

火龙果•整理
uml.org.cn

```
    a = b;
    b = tmp;
}
```

这个函数 swap 很简单：把两个整数的值交换一下。但当你写一个大程序时，经常需要交换 float、long、char 等变量，甚至如上面定义的 shape 变量，这时候你就需要把上面的代码修改一下，适应各自的参数类型，然后拷贝到很多处各自需要的地方。一旦需要对这块代码做些改动的时候，就必须把所有用到这块代码的地方一一做相应的修改，其烦无比！

解决这个问题的方法就是用模板(template)：

```
template <class T>
void swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp
}
```

注意这里 T 是任意的类型名字。看例子：

```
int a = 3, b =5;
shape MyShape, YourShape;
float fa = 3.01, fb = 5.02;
swap(a , b); //交换整数
swap(MyShape, YourShape); //交换自定义类的两个对象的值
swap(fa, fb); //交换浮点数
```

还可以看看两个函数模板例子：

```
template <class T>
T& min(T& a, T& b) //  取两个元素中的最小者
{
    return a < b ? a : b;
}
template <class T>
void pritn_to_cout(char* msg, T& obj) //  把一个元素输出到 cout 上
{
    cout << msg << ": " << obj << endl;
}
```

使用后者时，要求类型 T 中定义了 put-to 操作符(operator <<)。

2.3.2 类模板

创建类模板的动机和容器类的使用是密切相关的。比如一个容器类栈(stack)，定义者并不关心栈中对象的类型，而使用 stack 者自己指定到底包含什么对象类型。

下面看一个向量(vector)例子：

```
template <class T>
class vector
{
```

火龙果•整理
uml.org.cn

```
    T* v;
    int sz;
public:
    vector(int s) { v = new T[sz = s]; }
    ~vector() { delete [] v; }
    T& operator[] (int i) { return v[i]; }
    int get_size() { return sz; }

};
```

现在你可以实例化不同类型的 vector 容器了：

```
vector<int>     int_vector(10);
vector<char>    char_vector(10);
vector<shape> shape_vector(10);
```

2.3.3 模板特化

也许在某些情况下，编译器对某种类型产生的模板代码不另你满意，你也可以为这种类型给出一个特定的实现，此之谓“模板特化(template specialization)”。比如，你想让 shape 类型的 vector 只包含一个对象，则可以特化 vector 模板如下：

```
class vector<shape>
{
    shape v;
public:
    vector(shape& s) : v(s) {}
    shape& operator[] (int i) { return v; }
    int get_size() { return 1; }
};
```

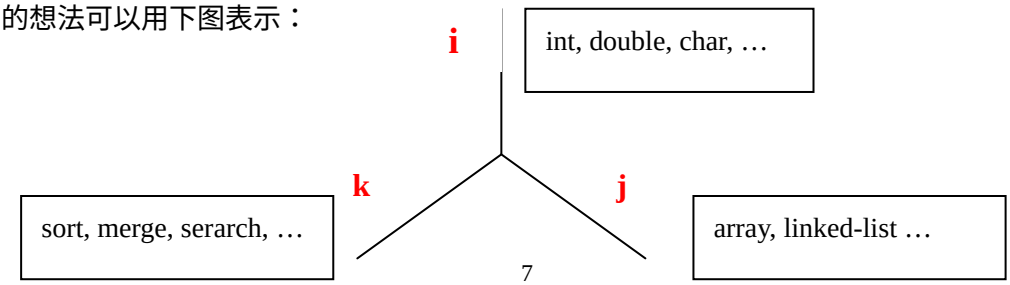
使用时：

```
shape MyShape;
vector<shape> single_shape_vector(MyShape);
```

STL 中有时会需要定义特定类型的算法或容器实现。

3 STL 概貌

STL 是个部件库(component library)，其中的部件包括有容器(container，储存任意类型对象的对象)和算法。只要用户对自己定义的部件做点小小的要求，STL 中的算法就可以工作在用户自定义的容器上，或者 STL 中的容器可以使用用户自定义的算法。隐藏在 STL 后面的想法可以用下图表示：



火龙果•整理
uml.org.cn

我们可以把软件部件想象成一个三位空间。第一维表示数据类型(int, double, char, …)，第二维表示容器(array, linked-list, …)，第三维表示算法(sort, merge, search, …)。

根据这个图示，需要设计 i*j*k 个不同的代码版本，比如整数数组的排序算法、double 数组的排序算法，double linked-list 的搜索算法…。通过使用数据类型作为参数的模板函数，第一维(i 轴)就可取消，而仅需要设计 j*k 个不同的代码。下一步是让算法可以工作在不同的容器上，这就意味着排序算法既可以用在 array 上，也可用在 linked-list 上。最后，只需要设计 j+k 个不同的代码版本了。

STL 具体化了上述思想，期望通过减少开发时间以简化软件开发，简化调试、维护并增加代码的可移植性。

STL 包含 5 个主要的部分，以后会有较详细的陈述：

· 算法(Algorithm)：能运行在不同容器(container)上的计算过程
· 容器(Container)：能够保留并管理对象的对象
· 迭代器(Iterator)：算法存取容器(algorithm-access to containers)的抽象，以便算法可以应用在不同的容器上
· 函数对象(Function Object)：定义了函数调用操作符(operator())的类
· 适应器(Adaptor)：封装一个部件以提供另外的接口(例如用 list 实现 stack)

3.1 STL 网上信息

STL 的 FTP 站点

HP 实验室(Alexander Stepanov 和 Meng Lee) ---

ftp://butler.hpl.hp.com/stl/

STL 的 WWW 站点

David Mussers 的 STL 主页 ---

http://www.cs.rpi.edu/~musser/stl.html

Link2go 上有关 C++编程的主页：

http://www.links2go.com/topic/C_and_C++_Programming_Languages

这三个地方我都去访问过。尤其第 3 个网址，其内容及其丰富，不仅仅有关 STL。

3.2 STL 文档

参考文献之

[6] 附件二

The Standard Template Libray 以及源代码, Alexander Stepanov & Meng Lee, 1995-10-31

是个 ZIP 文件，其中包含 Alexander Stepanov 和 Meng Lee 写的<<The Standard Template Library>>(1995-10-31 版)，这是 STL 鼻祖写的 STL 正式文档，定义严格、没有例子，很难懂(一些地方我还没理解)；但是 ZIP 文件中的 STL 源码却比较好懂，看起来不算太费劲 --- VC++中 STL 的实现源代码读起来可是极其痛苦的。

火龙果•整理
uml.org.cn

3.3 编译 STL 程序

Borland C++4.0 及以上的版本支持 STL。Microsoft Visual C++ 5.0 及以上的版本支持 STL。(VC++4.2 号称支持 STL，但支持的很差，极难用。现在有 VC++6.0，我们还是用最新的编译器吧!)

STL 使用时很简单：

```
//MyFile.cpp
#include “stdafx.h”
…
#include <vector> //include 你想用的 STL 头文件

using namespace std; //一定要写上这句话，因为 STL 都是在 std 名字空间中定义的

void F1()
{
…
vector <int> v(10); //现在你就可以放心大胆的使用 STL 了
…
}
```

4 学习 STL

4.1 容器(Container)

容器(Container)是能够保存其他类型的对象的类。容器形成了 STL 的关键部件。当书写任何类型软件的时候，把特定类型的元素集聚起来都是很至关重要的任务。

STL 中有顺序容器(Sequence Container)和关联容器(Associative Container)。

顺序容器组织成对象的有限线性集合，所有对象都是同一类型。STL 中三种基本顺序容器是：向量(Vector)、线性表(List)、双向队列(Deque)。

关联容器提供了基于 KEY 的数据的快速检索能力。元素被排好序，检索数据时可以二分搜索。STL 有四种关联容器。当一个 KEY 对应一个 Value 时，可以使用集合(Set)和映射(Map)；若对应同一 KEY 有多个元素被存储时，可以使用多集合(MultiSet)和多映射(MultiMap)。

STL 中支持的容器总结如下：

顺序容器(Sequence Container)	**向量(Vector)**
	双向队列(Deque)
	线性表(List)
关联容器(Associative Container)	**集合(Set)**
	多集合(MultiSet)
	映射(Map)
	多映射(MultiSet)

火龙果•整理
uml.org.cn

4.1.1 向量(Vector)

如果你要将所有 shape 对象保存在一个容器中，用 C++代码可以这样写：

```
shape my_shapes[max_size];
```

这里 max_size 是可以保存在 my_shapes 数组中的最大数量。

当你使用 STL 时，则可以这样写：

```
#include <vector>
using namespace std;
int main()
{
    vector<shape> my_shapes;
    // … 使用 my_shapes…
    return 0;
}
```

【此后的代码不再写#inlucde 行、using namespace std，以及 main()函数，直接写示例代码】

现在想得到容器中能保存的最大元素数量就可以用 vector 类的成员函数 max_size()：

```
vector<shape>::size_type max_size = my_shapes.max_size();
```

当前容器的实际尺寸 --- 已有的元素个数用 size()：

```
vector<shape>::size_type size = my_shapes.size();
```

就像 size_type 描述了 vector 尺寸的类型，value_type 说明了其中保存的对象的类型：

```
cout << “value type: “ << typeid(vector<float>::value_type).name();
```

输出：

```
value type: float
```

可以用 capacity()来取得 vector 中已分配内存的元素个数：

```
vector<int> v;
vector<int>::size_type capacity = v.capacity();
```

vector 类似于数组，可以使用下标[]访问：

```
vector<int> v(10);
v[0] = 101;
```

注意到这里预先给 10 个元素分配了空间。你也可以使用 vector 提供的插入函数来动态的扩展容器。成员函数 push_back()就在 vector 的尾部添加了一个元素：

```
v.push_back(3);
```

也可以用 insert()函数完成同样的工作：

```
v.insert(v.end(), 3);
```

这里 insert()成员函数需要两个参数：一个指向容器中指定位置的迭代器(iterator)，一个待插入的元素。insert()将元素插入到迭代器指定元素之前。

现在对迭代器(Iterator)做点解释。Iterator 是指针(pointer)的泛化，iterator 要求定义 operator*，它返回指定类型的值。Iterator 常常和容器联系在一起。例子：

```
vector<int> v(3);
v[0] = 5;
v[1] = 2;
v[2] = 7;
```

火龙果•整理
uml.org.cn

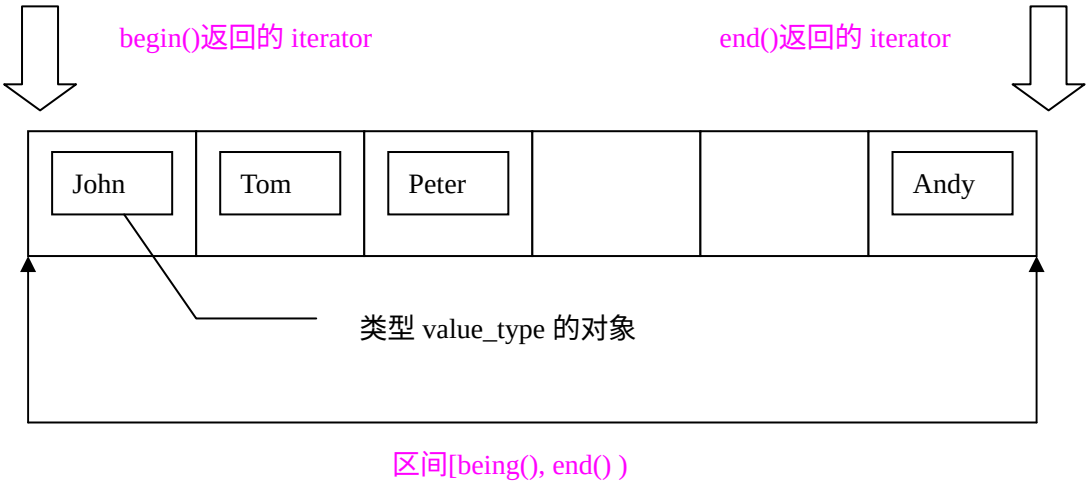
```
vector<int>::iterator first = v.begin();
vector<int>::iterator last = v.end();

while (first != last)
cout << *first++ << " ";
```

上面代码的输出是：

5 2 7

begin()返回的是 vector 中第一个元素的 iterator，而 end()返回的并不是最后一个元素的 iterator，而是 past the last element。在 STL 中叫 **past-the-end** iterator。

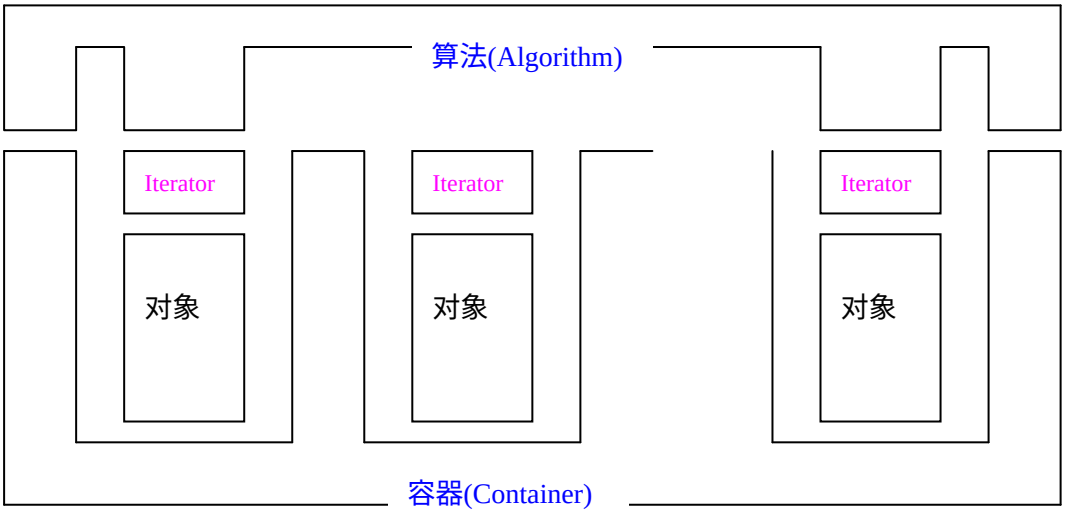


两个迭代器给定的区间在 STL 中非常重要，因为 STL 算法中大量的使用区间。比如排序算法：

```
sort(begin_iterator, past_the_end_iterator);
```

第一个参数指向区间的第一个元素，而第二个参数指向区间的 path-the-end 元素。

有了迭代器作为媒介，我们就可以把算法(algorithm)和容器(container)分开实现了：



下面继续 vector 的内容。

你可以用以下的几种方法声明一个 vector 对象：

火龙果•整理
uml.org.cn

```
vector<float> v(5, 3.25); //初始化有 5 个元素，其值都是 3.25
vector<float> v_new1(v);
vector<float> v_new2 = v;
vector<float> v_new3(v.begin(), v.end());
```

这四个 vector 对象是相等的，可以用 operator==来判断。

其余常用的 vector 成员函数有：

empty()：判断 vector 是否为空

front()：取得 vector 的第一个元素

back()：取得 vector 的最后一个元素

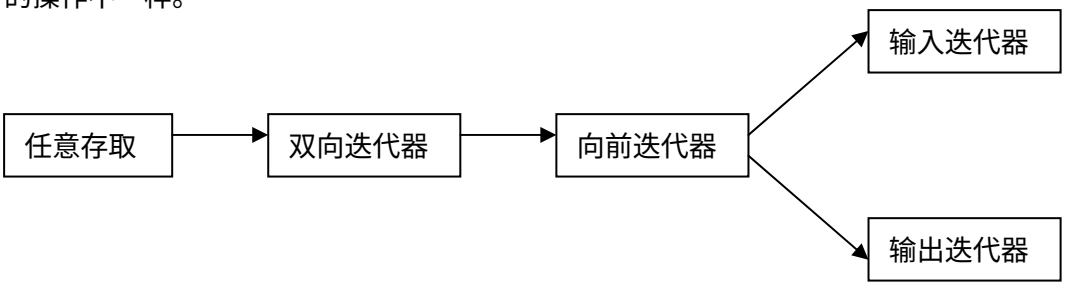
pop_back()：去掉最后一个元素

erase()：去掉某个 iterator 或者 iterator 区间指定的元素

到现在我们已经可以把一些对象存储在容器(container)中，并有几种手段来进行管理和维护。为了将算法应用到容器中的元素上，下面要对迭代器(Iterator)做进一步的解释。

4.2 迭代器(Iterator)

迭代器(Iterator)是指针(pointer)的泛化，它允许程序员以相同的方式处理不同的数据结构(容器)。STL 中有 5 中类型的迭代器，它们分别满足一定的要求。不同的迭代器要求定义的操作不一样。



箭头表示左边的迭代器一定满足右边迭代器需要的条件。比如某个算法需要一个双向迭代器(Bidirctional Iterator)，你可以把一个任意存取迭代器(Random Access Iterator)作为参数；但反之不行。

4.2.1 输入和输出迭代器

输入迭代器(Input Iterator)需要满足的条件：

- constructor
- assignment operator
- equality/inequality operator
- dereferenc operator
- pre/post increment operator

输入迭代器表示要从其中取出一个值，即从输入迭代器 X 中取值只可有以下三种形式：

V = *X++

V = *X, ++X

V = *X, X++

火龙果•整理
uml.org.cn

输出迭代器(Output Iterator)需要满足的条件：

- constructor
- assignment operator
- dereferenc operator
- pre/post increment operator

一个输出迭代器 X 只能有一个值 V 存储其中，在下一个存储之前，必须将 X 加一。即只可有以下三种形式之一：

*X++ = V

*X = V, ++X

*X = V, X++

注意，对输入和输出迭代器，一旦取出或放入值后，就只能前进(increment)，不可多次取出或放入。

4.2.2 向前迭代器

向前迭代器(Forward Iterator)需要满足的条件：

- constructor
- assignment operator
- equality/inequality operator
- dereferenc operator
- pre/post increment operator

和输入和输出迭代器比较而言，两个向前迭代器 r 和 s，若 r == s 则++r == ++s。而且既可取值，也可赋值：*X = V, V = *X。看一个例子：

```
template <class ForwardIterator, class T>
ForwardIterator find_linear(ForwardIterator first, ForwardIterator last, T& value)
{
    while (first != last)
        if (*first++ == value)
            return first;
    return last;
}
```

函数 find_linear()在一个容器中循环，如果找到指定的值，则返回该值的迭代器位置；否则返回 past-the-end 迭代器。下面用这个函数：

```
vector<int> v(3, 1);
v.push_back(7); // vector v:    1    1    1    7
vector<int>::iterator i = find_linear(v.begin(), v.end(), 7);
if (i != v.end())
    cout << *i;
else
    cout << “NOT FOUND”;
```

输出结果：7

火龙果•整理
uml.org.cn

4.2.3 双向迭代器

在向前迭代器的基础上，双向迭代器(Bidirectional Iterator)还满足以下需求：

- pre/post decrement operator

即双向迭代器不仅允许++，而且允许--。它允许一个算法走过(pass through)容器中的元素时，即可想前走，也可向后走。

我们看一个使用双向迭代器的多遍走过(multi-pass)算法 --- 气泡排序(Bubble Sort)：

```
template <class BidirectionalIterator, class Compare>
void bubble_sort(BidirectionalIterator first, BidirectionalIterator last, Compare comp)
{
	BidirectionalIterator left_el = first, right_el = first;
	right_el++;

	while (first != last)
	{
		while (right_el != last)
		{
			if (comp(*right_el, *left_el))
				iter_swap(left_el, right_el);
			right_el++;
			left_el++;
		}
		last--;
		left_el = first;
		right_el = first;
		right_el++;
	}
}
```

二元函数对象 Compare 由用户提供，得出两个参数的断言结果(true/false)。用法：

```
list<int> l; // list 类似于 vector，但不支持下标[]存取
// 填充 list
bubble_sort(l.begin(), l.end(), less<int>()); // 递增排序
bubble_sort(l.begin(), l.end(), greater<int>()); //递减排序
```

4.2.4 任意存取迭代器

在双向迭代器的基础上，任意存取迭代器(Random Access Iterator)还满足以下需求：

- opetrator+(int)
- opetrator+=(int)
- opetrator-(int)
- opetrator-=(int)
- opetrator-(random access iterator)
- opetrator[](int)

火龙果•整理
uml.org.cn

- opetrator>(random access iterator)
- opetrator<(random access iterator)
- opetrator>=(random access iterator)
- opetrator<=(random access iterator)

也就是说，任意存取迭代器可以“任意的”前进和后退。

4.3 算法和函数对象

STL 库中的算法都以迭代器类型为参数，这就和数据结果的具体实现分离开了。基于此，这些算法被称作基本算法(generic algorithm)。

4.3.1 如何创建基本算法

这里给出一个二分搜索的基本算法(generic bianry search algorithm)，可以看看基本算法的实现思路。

给定一个排好序的整数数组，要求二分搜索某个值的位置：

```
const int* binary_search(const int* array, int n, int x)
{
    const int* lo = array, *hi = array+n, *mid;
    while (lo != hi)
    {
        mid = lo+(hi-lo)/2;
        if (x == *mid)
            return mid;
        if (x < *mid)
            hi = mid;
        else
            lo = mid+1;
    }
    return 0;
}
```

为了让上述算法对任意类型的整数数组起作用，下面将之定义成模板函数：

```
template <class T>
const T* binary_search(const T* array, int n, const T& x)
{
    const T* lo = array, *hi = array+n, *mid;
    while (lo != hi)
    {
        mid = lo+(hi-lo)/2;
        if (x == *mid)
            return mid;
        if (x < *mid)
            hi = mid;
```

火龙果•整理
uml.org.cn

```
        else
            lo = mid+1;
    }
    return 0;
}
```

在上面的算法中，如果没有找到指定的值，则一个特殊的指针 NULL (0)被返回了，这就要求这个值存在。我们不想做这样的假定这样的值，可以不成功的搜索返回指针 array+n (就是 past-the-end)来代替：

```
template <class T>
const T* binary_search(const T* array, int n, const T& x)
{
    const T* lo = array, *hi = array+n, *mid;
    while (lo != hi)
    {
        mid = lo+(hi-lo)/2;
        if (x == *mid)
            return mid;
        if (x < *mid)
            hi = mid;
        else
            lo = mid+1;
    }
    return array+n;
}
```

为了代替给定数组及尺寸，我们可以给定第一个及 past-the-end 元素的指针：

```
template <class T>
const T* binary_search(T* first, T* last, const T& value)
{
    const T* lo = first, *hi = last, *mid;
    while (lo != hi)
    {
        mid = lo+(hi-lo)/2;
        if (value == *mid)
            return mid;
        if (value < *mid)
            hi = mid;
        else
            lo = mid+1;
    }
    return last;
}
```

现在可以看出，first 和 last 两个指针是任意类型的指针，即它们和类型 T 是没有关系的。这时就用到了迭代器(Iterator)的概念，因为这里排序要对容器任意存取，我们把 first 和 last 的类型命名为“ RandomAccessIterator ”。代码改写如下：

火龙果•整理
uml.org.cn

```
template <class RandomAccessIterator, class T>
RandomAccessIterator binary_search(RandomAccessIterator first, RandomAccessIterator last, const T& value)
{
    RandomAccessIterator not_found = last, mid;
    while (first != last)
    {
        mid = first+(last-first)/2;
        if (value == *mid)
            return mid;
        if (value < *mid)
            last = mid;
        else
            first = mid+1;
    }
    return not_found;
}
```

上面这个基本二分搜索算法即使对内部类型(build in types)功能也一样：

```
int x[10]; // 10 个整数数组
int search_value; //  太搜索的值
//  初始化变量
int* i = binary_search(&x[0], &x[10], search_value);
if (i == &x[10])
    cout << "value NOT FOUND";
else
    cout << "value found";
```

所有 STL 中的算法都是用以上类似的办法定义的。

4.3.2 STL 算法

STL 中的算法可以分成四组：

组(Group)	*算法类型(Algorithm Type)*
1	不改变顺序的操作(Non-mutating sequence Operations)
2	改变顺序的操作(Mutating sequence Operations)
3	排序及相关操作(Sorting and related Operations)
4	常用的数字操作(Generalized numeric Operations)

Group1 中的操作不改变容器中元素的顺序，而 Group2 中的要改变。当然 Group3 排序操作也会改变元素的顺序，但把排序相关的操作独立于 Group1 列出来了。Group4 中是些常用的数字操作。

先看个 Group1 中的一个算法“对每个(for_each)”，有三个参数，两个输入迭代器，一个函数对象。这个操作的意思是对[first, last)的每个元素都做一个 Function 操作：

```
template <class InputIterator, class Function>
```

火龙果•整理
uml.org.cn

```
Function for_each(InputIterator first, InputIterator last, Function f) {
    while (first != last) f(*first++);
    return f;
}
```

下面给一个使用 for_each 的例子：

```
template <class T>
class sum_up
{
public:
    void operator() (const T& value) { sum += value; }
    const T& read_sum()    { return sum; }
private:
    static T sum;
};

int sum_up<int>::sum = 0;

void main()
{
    deque<int> d(3, 2); //  两个元素：3 3
    sum_up<int> s;
    for_each(d.begin(), d.end(), s);
    cout << s.read_sum();
}
```

输出结果：6。注意到这里用到了函数对象 sum_up，它定义了 operator。所以函数对象是 STL 中一个非常重要的概念，屡屡用到。

Group1 中还有其他的操作，如“寻找(Find)”、“邻居寻找(Adjacent find)”、“计数(Count)”、“不匹配(Mismatch)”、“相等(Equal)”、“搜索(Search)”等。

说明，如果某个算法的后缀是_if，表示它自己提供断言(Predicate)函数，比如 find 算法有 find_if 的变种。

Group2 中的有算法“拷贝(Copy)”、“交换(Swap)”、“变换(Transform)”、“替换(Replace)”、“填充(Fill)”、“产生(Generate)”、“迁移(Remove)”、“唯一(Unique)”、“翻转(Reverse)”、“旋转(Rotate)”、“任意洗牌(Random shuffle)”、“分区(Partitions)”。

说明，如果某个算法的后缀有_copy，表示它要把一个迭代器区间的内容拷贝到另一个迭代器中。比如 replace 有 replace_copy 的变种。

Group3 中的排序算法都有两个版本，一个用函数对象做比较，一个用 operator<做比较。比如算法“排序(sort)”有两个版本：

```
void sort(RandomAccessIterator first, RandomAccessIterator last);
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp) ;
```

Group3 中还有算法“第 N 个元素(Nth element)”、“二分搜索(Binary Search)”、“合并(Merge)”、“排好序的设置操作(Set operations on sorted structures)”、“堆操作(Heap Operations)”、“最大最小(Minimum and Maximum)”、“词典比较(Lexicographical

火龙果•整理
uml.org.cn

comparison) ” 、“ 置换产生器(Permutation generator) ” 。

Group4 中包含些常用的数字算法，比如“ 聚集(Accumulate) ” 、“ 内部乘积(Inner product) ” 、“ 局部和(Partial sum) ” 、“ 邻近不同(Adjacent difference) ”。

由于时间和篇幅的关系，这里不可能对所有的算法一一解释了。用的时候你可以再看具体算法的说明。如果你搞明白算法实现，建议你读读 STL 的源码，我认为写的很漂亮，读起来感觉不错。(目前我也只读了部分代码)

4.4 适应器(Adaptor)

适应器(Adaptor)是提供接口映射的模板类(Adaptors are template classes that provide interface mappings)。适应器基于其他类来实现新的功能，成员函数可以被添加、隐藏，也可合并以得到新的功能。

4.4.1 容器适应器

栈(Stack)

栈可以用向量(vector)、线性表(list)或双向队列(deque)来实现：

```
stack<vector<int>> s1;
stack<list<int>>   s2;
stack<deque<int>> s3;
```

其成员函数有“ 判空(empty) ” 、“ 尺寸(Size) ” 、“ 栈顶元素(top) ” 、“ 压栈(push) ” 、“ 弹栈(pop) ” 等。

队列(Queue)

队列可以用线性表(list)或双向队列(deque)来实现(注意 vector container 不能用来实现 queue，因为 vector 没有成员函数 pop_front!)：

```
queue<list<int>>   q1;
queue<deque<int>>q2;
```

其成员函数有“ 判空(empty) ” 、“ 尺寸(Size) ” 、“ 首元(front) ” 、“ 尾元(backt) ” 、“ 加入队列(push) ” 、“ 弹出队列(pop) ” 等操作。

优先级队列(Priority Queue)

优先级队列可以用向量(vector)或双向队列(deque)来实现(注意 list container 不能用来实现 queue，因为 list 的迭代器不是任意存取 iterator，而 pop 中用到堆排序时是要求 random access iterator 的!)：

```
priority_queue<vector<int>, less<int>>      pq1;      // 使用递增 less<int>函数对象排序
priority_queue<deque<int>, greater<int>>    pq2;      // 使用递减 greater<int>函数对象排序
```

其成员函数有“ 判空(empty) ” 、“ 尺寸(Size) ” 、“ 栈顶元素(top) ” 、“ 压栈(push) ” 、“ 弹栈(pop) ” 等。

火龙果•整理
uml.org.cn

4.4.2 迭代适应器

逆向迭代器(Reverse Iterator)

顾名思义，逆向迭代器的递增是朝反方向前进的。对于顺序容器(vector, list 和 deque)，其成员函数 rbegin()和 rend()都返回了相应的逆向迭代器：

```
list<int> l;
for (int i = 1; i < 5; i++)
    l.push_back(i);
copy(l.rbegin(), l.rend(), ostream_iterator<int> (cout, " "));
```

输出结果是：4 3 2 1

插入迭代器(Insert Iterator)

插入迭代器简化了向容器中插入元素的工作。插入迭代器指定了向容器中插入元素的位置。STL 中有三种插入迭代器：

· 向后插入(back_insert_iterator)，在容器尾部插入
· 向前插入(front_insert_iterator)，在容器头部插入
· 插入(insert_iterator)，在容器中任一位置

STL 中提供了三个函数分别构造相应的插入迭代器：

· back_inserter
· front_inserter
· inserter

原始存储迭代器(Raw Storage Iterator)

该迭代器允许算法将其结果保存进没有初始化的内存中。

4.4.3 函数适应器

否认者(Negator)

有两个否认者 not1 和 not2 分别是一元和二元函数，它们分别使用一元和二元的谓词(Predicate)，返回的是谓词的结果的取反。

包扎者(Binder)

STL 中有两个包扎者函数 bind1st 和 bind2nd，可以将一些值限定在指定区间中。

函数指针的适应器(Adaptors for pointers to function)

STL 中的算法和容器一般都需要函数对象(function object)作为参数。如果想用到常用的 C++函数时，可以用 ptr_fun 把普通函数转换成函数对象。比如 ptr_fun(strcmp)就把常用的串比较函数 strcmp 包装成一个函数对象，就可用在 STL 算法和容器中了。

4.5 分配算符和内存处理

可移植性的一个主要问题是能够把有关内存模型的信息封装起来。这些信息有：

火龙果•整理
uml.org.cn

· 指针类型
· 指针差异的类型(ptrdiff_t)
· 内存模型中对象尺寸的类型(size_t)
· 内存分配和回收原语

STL 中提供的分配算符(allocator)对象封装了以上信息。STL 中的容器(container)都有一个 alllocator 参数，这样容器就不用关心内存模型信息了。

STL 中提供了缺省的 allocator 对象，各家编译器也提供其产品支持的不同的内存模型 allocator。对每一种内存模型都要提供以下几个模板函数：

· allocate：分配缓冲区
· deallocate：回收缓冲区
· construct：通过调用合适的拷贝构造函数将结果直接放入没有初始化的内存中
· destroy：调用指定指针的析构函数

5 其余的 STL 部件

5.1 各部件如何协同工作

STL 的容器(contianer)用来存储任意类型的对象。容器需要分配算符(allocator)为参数。分配算符(allocator)是能够封装所使用内存模型信息的对象，它提供内存原语以对内存进行统一的存取。每种内存模型都有自己特定的 allocator。Container 使用 allocator 完成对内存的操作，内存模型的改变只影响 allocator，而不会(在代码一级)影响 contianer 对象。

算法(algorithm)是计算顺序。两个算法不同在其本身的计算上，而非读取输入数据和写出输出数据的方法上。STL 为算法提供了统一的数据存取机制 --- 迭代器(iterator)。不同的迭代器提供不同存取方式。

函数对象(function object)用在和算法的结合中，用以扩展算法的效用。

适应器(adaptor)是接口映射，它们在基本或已有的部件上实现新的对象，以提供不同或扩展的能力。

在设计 STL 时，不同部件间的接口都定义的尽可能少。STL 的目的在于：

· 简化应用程序的设计
· 减少要写的代码函数
· 增加可理解度和可维护性
· 提供基本的质量保证

5.2 向量(Vector)

除了上面提到的 vector 成员函数之外，它还有一个 reserve()成员函数，用来预订给定的向量尺寸。reserve 可以减少存取元素时的内存重分配。

火龙果•整理
uml.org.cn

5.3 线性表(List)

和向量(vector)不一样，线性表(list)不支持对元素的任意存取(即不通过下标[]存取)。list 中提供的成员函数和 vector 类似，有 begin、end、rbegin、rend、push_back、pop_back，list 也提供对表首元素的操作 push_front、pop_front。两个线性表可以用 splice、merge 合并在一起，也可翻转(reverse)和排序(sort)，也可让表中同值的元素唯一(unique)。

5.4 双向队列(Deque)

象向量 vector 一样，双向队列(Deque)支持任意存取。它提供的成员函数还有 push_front、pop_front、insert、push、earse、pop 等。

5.5 迭代标签(Iterator Tag)

每个迭代器(iterator)必须定义表达式 iterator_tag()，它返回该迭代器的类别标签(category tag)。

STL 中有 5 个迭代器标签：input_iterator_tag、output_iterator_tag、forward_iterator_tag、bidirectional_iterator_tag、random_access_iterator_tag。

迭代器标签被用在编译时选择效率最高的算法(因为几乎所有的算法都用到迭代器)。

5.6 关联容器(Container)

关联容器(Associative Container)提供了快速检索基于关键词(Key)的数据的能力。和序列容器(vector、list、deque)一样，关联容器用来存储数据，而且设计关联容器时考虑到了优化数据检索的意图 --- 通过关键词(Key)作为标识把单一的数据记录组织到特定的结构中(如 tree)。STL 提供了不同的关联容器：集合(set)、多元集合(multiset)、映射(map)、多元映射(multimap)。

set 和 map 支持唯一关键词(unique key)，就是对每个 KEY，最多只保存一个元素(数据记录)。multiset 和 multimap 则支持相同关键词(equal key)，这样可有很多个元素可以用同一个 KEY 进行存储。set(multiset)和 map(multimap)之间的区别在于 set(multiset)中的存储数据内含了 KEY 表达式；而 map(multimap)则将 Key 表达式和对应的数据分开存放。

我们假设现在要保存某公司里雇员的信息。雇员信息类定义如下：

```
class employee_data
{
public:
    employee_data() : name(“”), skill(0), salary(0) {}
    employee_data(string n, int s, long sa) : name(n), skill(s), salary(sa) {}

    stirng name; //  雇员名字
    int skill; //  雇员职称
```

火龙果•整理
uml.org.cn

```
    long salary; // 雇员薪水

    friend ostream& operator<<(ostream& os, const employee_data& e);
};

ostream& operator<<(ostream& os, const employee_data& e)
{
    os << "employee: " << e.name << " " << e.skill << " " << e.salary;
    return os;
}
```

现在想把雇员数据保存在集合 set(multiset)中，关键词 KEY 包含在被保存的对象中：

```
class employee
{
public:
    employee(int ii, const employee_data& e) : identification_code(i), description(e) {}

    int identification_code; // 标识雇员的关键词
    employee_data description;

    bool operator<(const employee& e) const
    {
        return identification_code < e.identification_code;
    }
};
```

现在我们声明雇员集合 set(multiset)：

```
set<employee, less<employee>> employee_set;
multiset<employee, less<employee>> employee_multiset;
```

此时，employee 既是 Key type 又是 Value type。

如果我们想把雇员信息保存在映射 map(multimap)中，则如下声明：

```
map<int, employee_data, less<int>> employee_map;
multimap<int, employee_data, less<int>> employee_multimap;
```

此时 Key type 是 int，而 Value type 是 employee_data。

所有的关联容器都有以下成员函数：begin, end, rbegin, rend, empty, size, max_size, swap、insert、erase 等，其意义同顺序容器一样。下面我们用插入函数 insert 想关联容器中加入元素：

```
employee_data ed1("john", 1, 5000); // 雇员 John 的信息
employee_data ed2("tom", 5, 2000); // 雇员 tom 的信息
employee_data ed3("mary", 2, 3000); // 雇员 mary 的信息

employee e1(1010, ed1); // 雇员 John，证件号 1010(KEY)
employee e2(2020, ed2); // 雇员 tom，证件号 2020(KEY)
employee e3(3030, ed3); // 雇员 mary，证件号 3030(KEY)

employee_set.insert(e1); // 第一次，成功加入雇员 John 的信息
```

火龙果•整理
uml.org.cn

```
employee_set.insert(e1); // 第二次，不成功加入雇员 John 的信息。因为已经存在了
```

假定 John 和 Tom 在同一部门(部门代号：101)工作，而 Mary 在另一部门工作(部门代号：102)，现在把这三位用 multimap 来保存：

```
employee_multimap.insert(make_pair(101, ed1)); // 101 部门的 John
employee_multimap.insert(make_pair(101, ed2)); // 101 部门的 Tom
employee_multimap.insert(make_pair(102, ed3)); // 102 部门的 Mary
```

那么现在在 101 部门工作的雇员就有 2 人了：

```
multimap<int, employee_data, less<int>>::size_type count = employee_multimap.count(101);
```

有关关联容器成员函数的详细定义请参考 STL 源码。

6 版权信息

STL 可以自由的使用，只需要加上 HP 的版权说明即可：

```
/*
*
* Copyright (c) 1994
* Hewlett-Packard Company
*
* Permission to use, copy, modify, distribute and sell this
* software and its documentation for any purpose is hereby
* granted without fee, provided that the above copyright notice
* appear in all copies and that both that copyright notice and
* this permission notice appear in supporting documentation.
* Hewlett-Packard Company makes no representations about the
* suitability of this software for any purpose. It is provided
* "as is" without express or implied warranty.
*/
```

看微软的 STL 实现头文件，比如#include <vector>中，都有以上的版权提示。

7 文献

[1] Stroustrup, Bjarne：The C++ Programming Language – 2nd ed.
June, 1993

[2] Stepanov, Alex；Lee, Meng：The Standard Template Library
HP Labaratories, 1501 Page Mill Road, Palo Alto, CA 94304
October 31, 1995

[3] STL++
The Enhanced Standard Template Library, Tutorial & Reference Manual
Modena Software Inc., 236 N. Santa Cruz Ave, Suite 213, Los Gatos CA 95030
1994

火龙果•整理
uml.org.cn

[4] Standard Template Library Reference
Rensselaer Polytechnic Institute, 1994
includes as chapter 6
The STL Online Algorithm Reference
Cook, Robert Jr.；Musser, David R.；Zalewski, Kenneth J.
online at http://www.cs.rpi.edu/~musser/stl.html

[5] 附件一
The Standard Template Library Tutorial, Johannes Weidl, 1996-04-26
Stl-tut.zip

[6] 附件二
The Standard Template Libray 以及源代码, Alexander Stepanov & Meng Lee, 1995-10-31
Stl.zip

8 后话

本文我基本上是翻译 Johannes Weidl 的<<The Standard Template Library Tutorial>>一书，当然有节选，如果你想深入了解 STL，那么这本书(共 56 页)是个不错的开始。其中的大部分例子我都试过。

从网上看到的消息：VC++ 4.2 中就号称有 STL 的支持了，但有很多错误；到 VC++5.0 就可以了。我在 4.2 上试过确实不行，然后就直接安装 VC6.0 使用 STL 了。

在 MSDN Libray (April 1999)中有 STL 的详细文档说明以及 118 个 STL 的例子，可以帮助你理解和使用 STL 库。(MSDN 安装盘的位置在 748-domain 中，\\Soft_library\MSDN1\Library\199904\Setup，注意安装时选择“VC++ Document”，STL 文档就在其中。

实际上 STL 中有的功能在 MFC 中都有替代品。例如在 MFC 中有 CArray 等。不过，在你只能使用标准 C/C++如在 Unix 或 Linux 环境下要使用这些数据结构，使用 STL 是一个很好的选择。**STL 最大的特点是它使用的算法非常可靠且效率很高**，例如其中的二叉树算法就很经典。不过看 STL 源代码还是有点困难的，需要下点功夫。不过如果只是要使用可以不看源代码，使用起来很简单的。

我个人的建议是：目前我们程序都是用微软的工具开发的，还用它提供的开发工具生成程序框架，但自己的代码尽量少使用类库中的东东，转向使用 STL，因为 STL 现在是 C++标准的一部分而且效率很好。