C++ 标准库学习

整理主要来源：OIwiki

[TOC]

标准库简介

C++标准

> C++ 自 1985 年诞生以来，一共由国际标准化组织（ISO）发布了 5 个正式的 C++ 标准，依次为 C++98、C++03、C++11（亦称 C++0x）、C++14（亦称 C++1y）、C++17（亦称 C++1z）、C++20（亦称 C++2a）。C++ 标准草案在 open-std 网站上，最新的标准 C++23（亦称 C++2b）仍在制定中。此外还有一些补充标准，例如 C++ TR1。

> 每一个版本的 C++ 标准不仅规定了 C++ 的*语法*、*语言特性*，还**规定了一套 C++ 内置库的实现规范**，这个库便是 **C++ 标准库**。C++ 标准库中包含大量常用代码的实现，如输入输出、基本数据结构、内存管理、多线程支持等。掌握 C++ 标准库是编写更现代的 C++ 代码必要的一步。C++ 标准库的详细文档在 cppreference 网站上，文档对标准库中的类型函数的用法、效率、注意事项等都有介绍，请善用。

标准模板库（STL）

标准模板库（Standard Template Library），是 C++ 标准库的一部分，里面包含了一些模板化的通用的数据结构和算法 由于其模板化的特点，它能够兼容自定义的数据类型，避免大量的造轮子（Reinventing_the_wheel）工作

STL的六大件包括容器、算法、迭代器、仿函数、适配器和空间配置器，其中几乎所有代码均使用了模板类和模板函数的概念[^1]

Boost库

乃除了标准库外，另一个久副盛名的**开源 C++ 工具库**，其代码具有可移植、高质量、高性能、高可靠性等特点。 其模块数量非常之大，功能全面，并且拥有完备的跨平台支持，因此被看作 C++ 的准标准库。 C++ 标准中的不少特性也都来自于 Boost，如智能指针、元编程、日期和时间等

STL容器

> 在C++ 中容器被定义为：**在数据存储上，有一种对象类型，它可以持有其它对象或指向其它对像的指针，这种对象类型就叫做容器。** 很简单，容器就是**保存其它对象的对象**，当然这是一个朴素的理解，这种“对象”还包含了一系列处理“其它对象”的方法，因为这些方法在程序的设计上会经常被用到，所以容器也体现了一个好处， 就是“容器类是一种**对特定代码重用问题的良好的解决方案**”。 容器还有另一个特点是容器可以自行扩展。在解决问题时我们常常不知道我们需要存储多少个对象，也就是说我们不知道应该创建多大的内存空间来保存我们的对象。 显然，数组在这一方面也力不从心。容器的优势就在这里，它不需要你预先告诉它你要存储多少对象，只要你创建一个容器对象，并合理的调用它所提供的方法，所 有的处理细节将由容器来自身完成。它可以为你申请内存或释放内存，并且用最优的算法来执行您的命令。 容器是随着面向对象语言的诞生而提出的，容器类在面向对象语言中特别重要，甚至它被认为是早期面向对象语言的基础。在现在几乎所有的面向对象的语言中也都伴随着一个容器集，在C++ 中，就是标准模板库（STL）。 和其它语言不一样，C++ 中处理容器是采用基于模板的方式。标准C++ 库中的

> 容器提供了多种数据结构，这些数据结构可以与标准算法一起很好的工作，这为我们的软件开发提供了良好的支持！

> kimiAI：在C++中，容器（Container）是标准模板库（Standard Template Library，STL）的一部分，它**提供了一种灵活的方式来存储数据集合。**容器**可以容纳各种类型的数据，并且提供了多种操作来管理这些数据。**

STL简介

分类

分类

序列式容器（顺序容器）

- **向量(vector)** 后端可高效增加元素的顺序表。
- **数组(array)** C++11，定长的顺序表，C 风格数组的简单包装。
- **双端队列(deque)** 双端都可高效增加元素的顺序表。
- **列表(list)** 可以沿双向遍历的链表。
- **单向列表(forward_list)** 只能沿一个方向遍历的链表。

关联式容器

- **集合(set)** 用以有序地存储 **互异** 元素的容器。其实现是由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种比较元素大小的谓词进行排列。
- **多重集合(multiset)** 用以有序地存储元素的容器。允许存在相等的元素。
- **映射(map)** 由 {键，值} 对组成的集合，以某种比较键大小关系的谓词进行排列。
- **多重映射(multimap)** 由 {键，值} 对组成的多重集合，亦即允许键有相等情况的映射。

无序（关联式）容器

- **无序（多重）集合(unordered_set/unordered_multiset)** C++11，与 `set`/`multiset` 的区别在于元素无序，只关心「元素是否存在」，使用哈希实现。
- **无序（多重）映射(unordered_map/unordered_multimap)** C++11，与 `map`/`multimap` 的区别在于键 (key) 无序，只关心 "键与值的对应关系"，使用哈希实现。

容器适配器

容器适配器其实**并不是容器。**它们不具有容器的某些特点（如：有迭代器、有 clear() 函数……）

> 「适配器是使一种事物的行为类似于另外一种事物行为的一种机制」，适配器对容器进行包装，使其表现出另外一种行为。

- **栈(stack)** 后进先出 (LIFO) 的容器，默认是对双端队列（deque）的包装。
- **队列(queue)** 先进先出 (FIFO) 的容器，默认是对双端队列（deque）的包装。
- **优先队列(priority_queue)** 元素的次序是**由作用于所存储的值对上的某种谓词决定的**的一种队列，默认是对向量（vector）的包装。

共同点

容器声明

都是 `containerName<typeName,...> name` 的形式，但模板参数（`<>` 内的参数）的个数、形式会根据具体容器而变。

本质原因：STL 就是「标准模板库」，所以容器都是模板类。

泛型编程是一种编程范式，旨在编写可以适用于多种数据类型的通用代码。通过泛型编程，我们可以编写一次代码，然后将其应用于不同的数据类型，从而避免重复编写相似的代码 编写与类型无关的通用代码，是代码复用的一种手段。**模板是泛型编程的基础。** 面向对象编程（OOP）和泛型编程都能处理在编写程序时不知类型的情况，不同之处在于：OOP能处理类型在程序运行之前都未知的情况：而在泛型编程中，在编译时就能获知类型了

迭代器

在 STL 中，迭代器（`Iterator`）用来**访问和检查** STL 容器中元素的对象，它的行为模式和*指针*类似，但是它封装了一些有效性检查，并且提供了统一的访问格式。类似的概念在其他很多高级语言中都存在，如 Python 的 `__iter__` 函数，C# 的 `IEnumerator`。

基础使用

其实迭代器本身可以看作一个数据指针。迭代器主要支持两个运算符：自增 `++` 和解引用（单目 `*` 运算符） 其中自增用来**移动迭代器**，解引用可以**获取或修改它指向的元素。** 指向某个 STL 容器 `container` 中元素的迭代器的类型一般为 `container::iterator`

迭代器可以用来遍历容器，例如，下面两个 for 循环的效果是一样的：

vector data(10);

for (int i = 0; i < data.size(); i++) cout << data[i] << endl; // 使用下标访问元素

for (vector::iterator iter = data.begin(); iter != data.end(); iter++) cout << *iter << endl; // 使用迭代器访问元素
// 在C++11后可以使用 auto iter = data.begin() 来简化上述代码： vector data(10);

for (int i = 0; i < data.size(); i++) cout << data[i] << endl; // 使用下标访问元素

for (vector::iterator iter = data.begin(); iter != data.end(); iter++) cout << *iter << endl; // 使用迭代器访问元素
// 在C++11后可以使用 auto iter = data.begin() 来简化上述代码

```
vector<int> data(10);

for (int i = 0; i < data.size(); i++)
  cout << data[i] << endl;  // 使用下标访问元素

for (vector<int>::iterator iter = data.begin(); iter != data.end(); iter++)
  cout << *iter << endl;  // 使用迭代器访问元素
// 在C++11后可以使用 auto iter = data.begin() 来简化上述代码
```

auto之于算法竞赛 大部分选手都喜欢使用 auto 来代替繁琐的迭代器声明。NOI 系列比赛（包括 CSP J/S）在评测时将使用 C++14，这个版本已经支持了 auto 关键字。

分类

迭代器根据其支持的操作依次分为以下几类：

- **InputIterator（输入迭代器）**：只要求支持拷贝、自增和解引访问。
- **OutputIterator（输出迭代器）**：只要求支持拷贝、自增和解引赋值。
- **ForwardIterator（向前迭代器）**：同时满足 InputIterator 和 OutputIterator 的要求。
- **BidirectionalIterator（双向迭代器）**：在 ForwardIterator 的基础上支持自减（即反向访问）。
- **RandomAccessIterator（随机访问迭代器）**：在 BidirectionalIterator 的基础上支持加减运算和比较运算（即随机访问）。
- **ContiguousIterator（连续迭代器）**：在 RandomAccessIterator 的基础上要求对可解引用的迭代器 `a + n` 满足 `(a + n)` 与 `(std::address_of(*a) + n)` 等价（即连续存储，其中 a 为连续迭代器、n 为整型值）。

（ContiguousIterator 于 C++17 中正式引入。） （「输入」指的是「可以从迭代器中获取输入」，而「输出」指的是「可以输出到迭代器」。「输入」和「输出」的施动者是程序的其它部分，而不是迭代器自身。） （指针满足随机访问迭代器的所有要求，可以当作随机访问迭代器使用。）

迭代器并不互斥——一个「类别」的迭代器是可以包含另一个「类别」的迭代器的。例如，在要求使用向前迭代器的地方，同样可以使用双向迭代器。

相关函数

很多 STL 函数 都使用迭代器作为参数。

- 可以使用 `std::advance(it, n)` 将迭代器 `it` 向后移动 `n` 步；若 `n` 为负数，则对应向前移动。迭代器必须满足双向迭代器，否则行为未定义。
- 可以使用 `std::next(it)` 获得向前迭代器 `it` 的后继（此时迭代器 `it` 不变），`std::next(it, n)` 获得向前迭代器 `it` 的第 `n` 个后继。（C++11）
- 可以使用 `std::prev(it)` 获得双向迭代器 `it` 的前驱（此时迭代器 `it` 不变），`std::prev(it, n)` 获得双向迭代器 `it` 的第 `n` 个前驱。（C++11）

STL 容器 一般支持从一端或两端开始的访问，以及对 const 修饰符 的支持。 例如容器的 `begin()` 函数可以获得指向容器第一个元素的迭代器，`rbegin()` 函数可以获得指向容器最后一个元素的反向迭代器，`cbegin()` 函数可以获得指向容器第一个元素的 const 迭代器，`end()` 函数可以获得指向容器尾端（「尾端」并不是最后一个元素，可以看作是最后一个元素的后继；「尾端」的前驱是容器里的最后一个元素，其本身不指向任何一个元素）的迭代器。

共有函数

- `=`：有赋值运算符以及复制构造函数。
- `begin()`：返回指向开头元素的迭代器。
- `end()`：返回指向末尾的下一个元素的迭代器。end() 不指向某个元素，但它是末尾元素的后继。
- `size()`：返回容器内的元素个数。
- `max_size()`：返回容器**理论上**能存储的最大元素个数。依容器类型和所存储变量的类型而变。

- `empty()`：返回容器是否为空。
- `swap()`：交换两个容器。
- `clear()`：清空容器。
- `==`/`!=`/`<`/`>`/`<=`/`>\=`：按 字典序 比较两个容器的大小。（比较元素大小时 map 的每个元素相当于 set<pair<key, value> >，无序容器不支持 </>/<=/>=。）

序列式容器（即顺序容器）

vector

`std::vector` 是 STL 提供的 内存连续的、可变长度 的数组（亦称列表）数据结构。能够提供线性复杂度的插入和删除，以及常数复杂度的随机访问。

为何使用vector（OIwiki）

作为 OIer，对程序效率的追求远比对工程级别的稳定性要高得多，而 vector 由于其对内存的动态处理，**时间效率在部分情况下低于静态数组**，并且在 OJ 服务器不一定开全优化的情况下更加糟糕。所以在正常存储数据的时候，通常不选择 vector。下面给出几个 vector 优秀的特性，**在需要用到这些特性的情况下，vector 能给我们带来很大的帮助。**

vector 可以动态分配内存

很多时候我们**不能提前开好那么大的空间**（eg：预处理 1~n 中所有数的约数）。尽管我们能知道数据总量在空间允许的级别，但是单份数据还可能非常大，这种时候我们就**需要 vector 来把内存占用量控制在合适的范围内。**vector 还支持**动态扩容**，在内存非常紧张的时候这个特性就能派上用场了。

vector 重写了比较运算符及赋值运算符

vector **重载了六个比较运算符**，以**字典序**实现，这使得我们可以**方便地判断两个容器是否相等**（*复杂度与容器大小成线性关系*）。例如可以利用 `vector<char>` 实现字符串比较（当然，还是用 `std::string` 会更快更方便）。另外 vector 也重载了赋值运算符，使得数组拷贝更加方便。

vector 便利的初始化

由于 `vector` 重载了 `=` 运算符，所以我们**可以方便地初始化。**此外从 C++11 起 vector 还支持 **列表初始化**，例如 `vector<int> data {1, 2, 3};`

vector 之使用方法

以下为常用用法，详细内容 请参见 C++ 文档。

构造函数

用例参见如下代码：

```
using namespace std;
```

```
// 1. 创建空vector；常数复杂度
vector<int> v0;
// 1+. 这句代码可以使得向vector中插入前3个元素时，保证常数时间复杂度
v0.reserve(3);
// 2. 创建一个初始空间为3的vector，其元素的默认值是0；线性复杂度
vector<int> v1(3);
// 3. 创建一个初始空间为3的vector，其元素的默认值是2；线性复杂度
vector<int> v2(3, 2);
// 4. 创建一个初始空间为3的vector，其元素的默认值是1，
// 并且使用v2的空间配置器；线性复杂度
vector<int> v3(3, 1, v2.get_allocator());
// 5. 创建一个v2的拷贝vector v4， 其内容元素和v2一样；线性复杂度
vector<int> v4(v2);
// 6. 创建一个v4的拷贝vector v5，其内容是{v4[1], v4[2]}；线性复杂度
vector<int> v5(v4.begin() + 1, v4.begin() + 3);
// 7. 移动v2到新创建的vector v6，不发生拷贝；常数复杂度；需要 C++11
vector<int> v6(std::move(v2));  // 或者 v6 = std::move(v2);
```

```
// 以下是测试代码，有兴趣的同学可以自己编译运行一下本代码。
cout << "v1 = ";
copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v2 = ";
copy(v2.begin(), v2.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v3 = ";
copy(v3.begin(), v3.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v4 = ";
copy(v4.begin(), v4.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v5 = ";
copy(v5.begin(), v5.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v6 = ";
copy(v6.begin(), v6.end(), ostream_iterator<int>(cout, " "));
cout << endl;
```

元素访问

有如下方法

方法	解释
`at()`	v.at(pos) 返回容器中下标为 `pos` 的引用。如果数组越界抛出 `std::out_of_range` 类型的异常。
`operator[]`	`v[pos]` 返回容器中下标为 `pos` 的引用。不执行越界检查。
`front()`	`v.front()` 返回首元素的引用。

方法	解释
`back()`	`v.back()` 返回末尾元素的引用。
`data()`	`v.data()` 返回指向数组第一个元素的指针。

vector 迭代器

vector 提供了如下几种 迭代器：

迭代器	解释
`begin()`/`cbegin()`	返回指向首元素的迭代器，其中 `*begin = front`。
`end()`/`cend()`	返回指向数组尾端占位符的迭代器，注意是没有元素的。
`rbegin()`/`crbegin()`	返回指向逆向数组的首元素的逆向迭代器，可以理解为正向容器的末元素。
`rend()`/`crend()`	返回指向逆向数组末元素后一位置的迭代器，对应容器首的前一个位置，没有元素。

上述的迭代器中，含有字符 `c` 的为只读迭代器，你不能通过只读迭代器去修改 vector 中的元素的值。如果一个 vector 本身就是只读的，那么它的一般迭代器和只读迭代器完全等价。只读迭代器自 C++11 开始支持。

长度和容量

vector 有以下几个与容器长度和容量相关的函数。注意，vector 的**长度（size）指有效元素数量**，而**容量（capacity）指其实际分配的内存长度**，相关细节请参见后文的实现细节介绍。

长度相关：

函数	解释
`empty()`	返回一个 bool 值，即 `v.begin() == v.end()`，true 为空，false 为非空。
`size()`	返回容器长度（元素数量），即 `std::distance(v.begin(), v.end())`。
`resize()`	改变 vector 的长度，多退少补。补充元素可以由参数指定。
`max_size()`	返回容器的最大可能长度。

容量相关：

函数	解释
`reserve()`	使得 vector 预留一定的内存空间，避免不必要的内存拷贝。
`capacity()`	返回容器的容量，即不发生拷贝的情况下容器的长度上限。
`shrink_to_fit()`	使得 vector 的容量与长度一致，多退但不会少。

元素增删及修改

操作	含义
clear()`	清除所有元素
`insert()`	支持在某个迭代器位置插入元素、可以插入多个。复杂度与 `pos` 距离末尾长度成线性而非常数的
`erase()`	删除某个迭代器或者区间的元素，返回最后被删除的迭代器。复杂度与 `insert` 一致。
`push_back()`	在末尾插入一个元素，均摊复杂度为 常数，最坏为线性复杂度。
`pop_back()`	删除末尾元素，常数复杂度。
`swap()`	与另一个容器进行交换，此操作是 常数复杂度 而非线性的。

vector 的实现细节

vector 的**底层仍是定长数组**，它能够实现动态扩容的原因是**增加了避免数量溢出**的操作。 首先需要指明的是 vector 中元素的**数量（长度）n** 与它已**分配内存最多能包含元素的数量（容量）N** 是不一致的，vector 会分开存储这两个量。 当向 vector 中添加元素时，如发现 **n>N**，那么容器会分配一个**尺寸为 2N 的**数组，然后将旧数据从原本的位置**拷贝到新的数组中**，再将原来的内存**释放**。尽管这个操作的渐进复杂度是 O(n)，但是可以证明其**均摊复杂度为 O(1)**。 而在末尾删除元素和访问元素则都仍然是 O(1) 的开销。 因此，只要对 vector 的尺寸估计得当并善用 `resize()` 和 `reserve()`，就能使得 vector 的效率与定长数组不会有太大差距。

`vector<bool>`

标准库特别提供了对 bool 的 vector **特化**，每个「bool」只占 1 bit，且支持动态增长。但是其 `operator[]` 的返回值的类型不是 `bool&` 而是 `vector<bool>::reference`。因此，使用 `vector<bool>` 使需谨慎，可以考虑使用 `deque<bool>` 或 `vector<char>` 替代。而如果你需要节省空间，请直接使用 `bitset`（见之后）。

array(C++11)

`std::array` 是 STL 提供的 **内存连续的、固定长度** 的数组数据结构。其本质是对原生数组的直接封装。

为何用array

array 实际上是 **STL 对数组的封装**。它相比 vector **牺牲了动态扩容的特性**，但是换来了**与原生数组几乎一致的性能**（在开满优化的前提下）。因此能够使用原生数组的地方几乎都可以直接把定长数组都换成 array，而动态分配的数组可以替换为 vector

成员函数

隐藏定义的成员函数

函数	作用
`operator=`	以来自另一 array 的每个元素重写 array 的对应元素

元素访问

函数	作用
at	访问指定的元素，同时进行越界检查
operator[]	访问指定的元素，不进行越界检查
front	访问第一个元素
back	访问最后一个元素
data	返回指向内存中数组第一个元素的指针

> at 若遇 pos >= size() 的情况会抛出 std::out_of_range。

容量

函数	作用
empty	检查容器是否为空
size	返回容纳的元素数
max_size	返回可容纳的最大元素数

操作

函数	作用
fill	以指定值填充容器
swap	交换内容

注意，交换两个 array 是 $\Theta(\text{size})$ 的，而非与常规 STL 容器一样为 $O(1)$。

非成员函数

函数	作用
operator== 等	按照字典序比较 array 中的值
std::get	访问 array 的一个元素
std::swap	特化的 std::swap 算法

下面是一个 array 的使用示例：

```
// 1. 创建空array，长度为3；常数复杂度
std::array<int, 3> v0;
// 2. 用指定常数创建array；常数复杂度
std::array<int, 3> v1{1, 2, 3};

v0.fill(1);  // 填充数组
```

```
// 访问数组
for (int i = 0; i != arr.size(); ++i) cout << arr[i] << " ";
```

deque

`std::deque` 是 STL 提供的 **双端队列** 数据结构。能够提供线性复杂度的插入和删除，以及常数复杂度的随机访问。

注： [^1]: 链接 [^2]: