

04讲容器汇编I：比较简单的若干容器

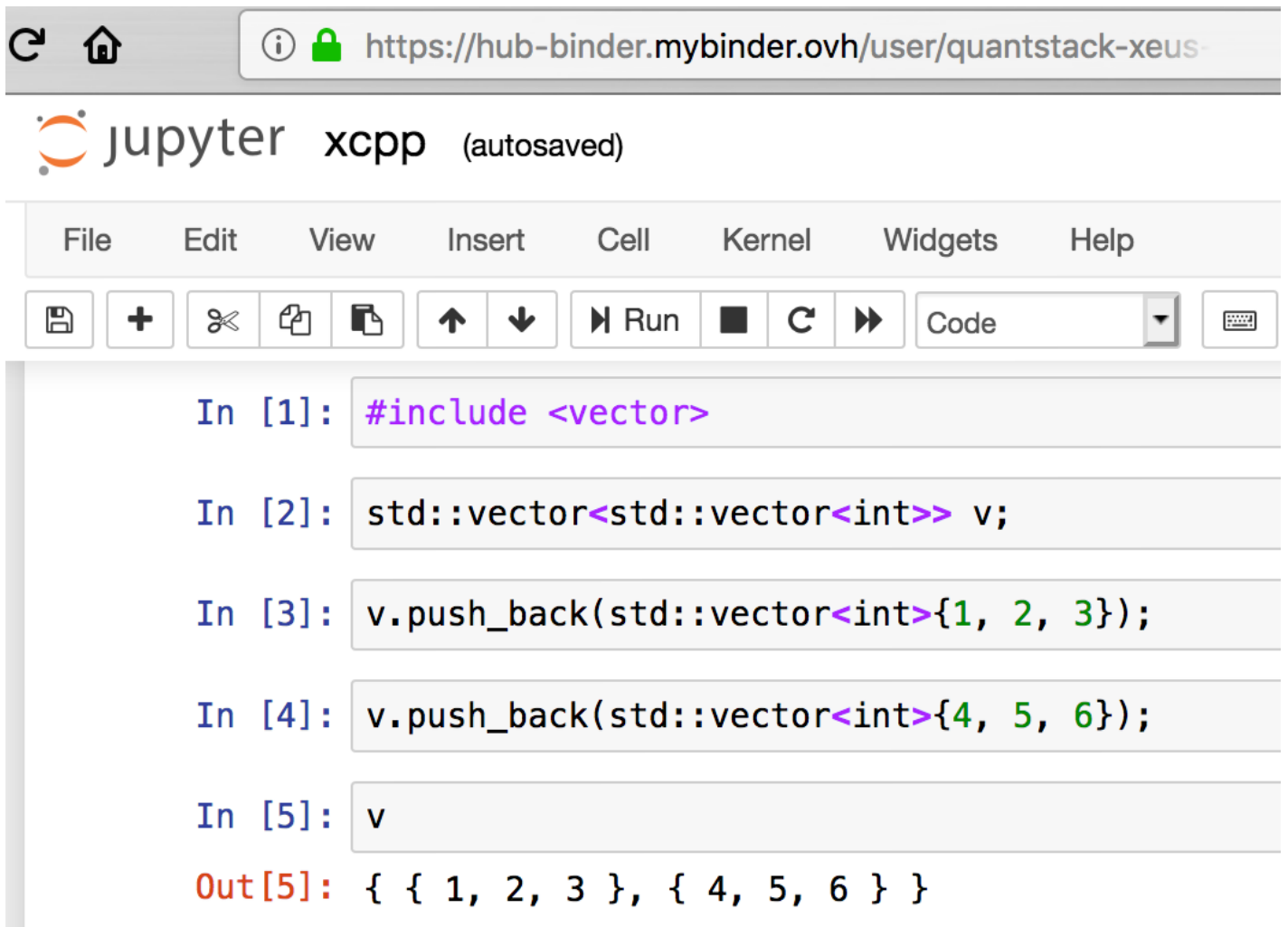


你好，我是吴咏炜。

上几讲我们学习了 C++ 的资源管理和值类别。今天我们换一个话题，来看一下 C++ 里的容器。

关于容器，已经存在不少的学习资料了。在 [cppreference](#) 上有很完备的参考资料 [\[1\]](#)。今天我们采取一种非正规的讲解方式，尽量不重复已有的参考资料，而是让你加深对于重要容器的理解。

对于容器，学习上的一个麻烦点是你无法直接输出容器的内容——如果你定义了一个 `vector<int> v`，你是没法简单输出 `v` 的内容的。有人也许会说用 `copy(v.begin(), v.end(), ostream_iterator(...))`，可那既啰嗦，又对像 `map` 或 `vector<vector<...>>` 这样的复杂类型无效。因此，我们需要一个更好用的工具。在此，我向你大力推荐 [xeus-cling](#) [\[2\]](#)。它的便利性无与伦比——你可以直接在浏览器里以交互的方式运行代码，不需要本机安装任何编译器（点击“Trying it online”下面的 `binder` 链接）。下面是在线运行的一个截图：



```
In [1]: #include <vector>

In [2]: std::vector<std::vector<int>> v;

In [3]: v.push_back(std::vector<int>{1, 2, 3});

In [4]: v.push_back(std::vector<int>{4, 5, 6});

In [5]: v

Out[5]: { { 1, 2, 3 }, { 4, 5, 6 } }
```

xeus-cling 也可以在本地安装。对于使用 Linux 的同学，安装应当是相当便捷的。有兴趣的话，使用其他平台的同学也可以尝试一下。

如果你既没有本地运行的条件，也不方便远程使用互联网来运行代码，我个人还为本专栏写了一个小小的工具 [3]。在你的代码中包含这个头文件，也可以方便地得到类似于上面的输出。示例代码如下所示：

```

#include <iostream>
#include <map>
#include <vector>
#include "output_container.h"

using namespace std;

int main()
{
    map<int, int> mp{
        {1, 1}, {2, 4}, {3, 9}};
    cout << mp << endl;
    vector<vector<int>> vv{
        {1, 1}, {2, 4}, {3, 9}};
    cout << vv << endl;
}

```

我们会得到下面的输出：

```

{ 1 => 1, 2 => 4, 3 => 9 }
{ { 1, 1 }, { 2, 4 }, { 3, 9 } }

```

这个代码中用到了很多我们目前专栏还没有讲的知识，所以你暂且不用关心它的实现原理。如果你能看得懂这个代码，那就太棒了。如果你看不懂，唔，不急，慢慢来，你会明白的。

工具在手，天下我有。下面我们正式开讲容器篇。

string

`string` 一般并不被认为是一个 C++ 的容器。但鉴于其和容器有很多共同点，我们先拿 `string` 类来说。

`string` 是模板 `basic_string` 对于 `char` 类型的特化，可以认为是一个只存放字符 `char` 类型数据的容器。“真正”的容器类与 `string` 的最大不同点是里面可以存放任意类型的对象。

跟其他大部分容器一样，`string` 具有下列成员函数：

- `begin` 可以得到对象起始点
- `end` 可以得到对象的结束点
- `empty` 可以得到容器是否为空
- `size` 可以得到容器的大小
- `swap` 可以和另外一个容器交换其内容

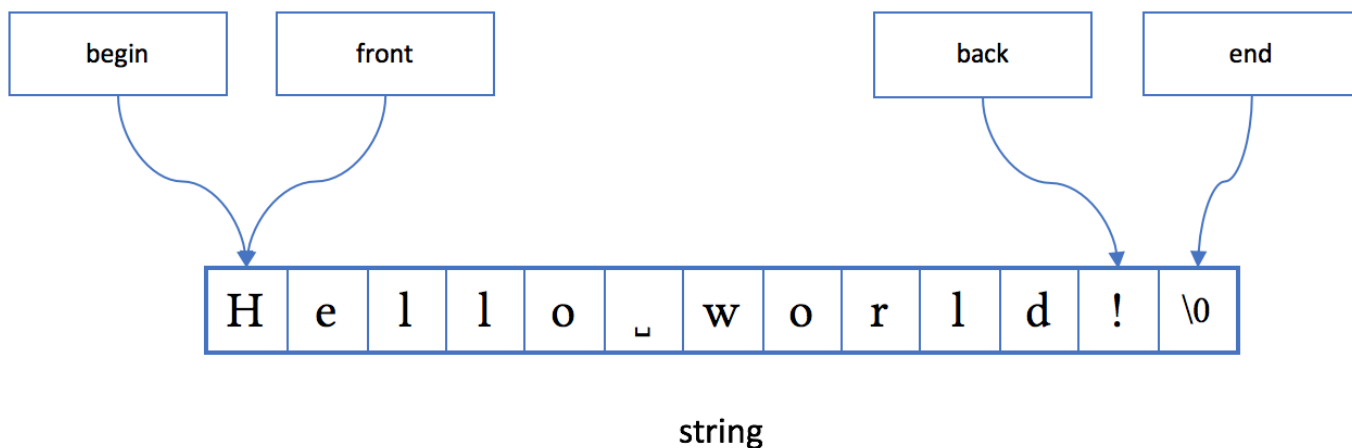
(对于不那么熟悉容器的人，需要知道 C++ 的 `begin` 和 `end` 是半开半闭区间：在容器非空时，`begin` 指向一个第一个元素，而 `end` 指向最后一个元素后面的位置；在容器为空时，`begin` 等于 `end`。在 `string` 的情况下，由于考虑到和 C 字符串的兼容，`end` 指向代表字符串结尾的 `\0` 字符。)

上面就几乎是所有容器的共同点了。也就是说：

- 容器都有开始和结束点
- 容器会记录其状态是否非空
- 容器有大小
- 容器支持交换

当然，这只是容器的“共同点”而已。每个容器都有其特殊的用途。

`string` 的内存布局大致如下图所示：



下面你会看到，不管是内存布局，还是成员函数，`string` 和 `vector` 是非常相似的。

`string` 当然是为了存放字符串。和简单的 C 字符串不同：

- `string` 负责自动维护字符串的生命周期
- `string` 支持字符串的拼接操作（如之前说过的 `+` 和 `+=`）
- `string` 支持字符串的查找操作（如 `find` 和 `rfind`）
- `string` 支持从 `istream` 安全地读入字符串（使用 `getline`）
- `string` 支持给期待 `const char*` 的接口传递字符串内容（使用 `c_str`）
- `string` 支持到数字的互转（`stoi` 系列函数和 `to_string`）
- 等等

推荐你在代码中尽量使用 `string` 来管理字符串。不过，对于对外暴露的接口，情况有一点复杂。我一般不建议在接口中使用 `const string&`，除非确知调用者已经持有 `string`：如果函数里不对字符串做复杂处理的话，使用 `const char*` 可以避免在调用者只有 C 字符串时编译器自动构造 `string`，这种额外的构造和析构代价并不低。反过来，如果实现较为复杂、希望使用 `string` 的成员函数的话，那就应该考虑下面的策略：

- 如果不修改字符串的内容，使用 `const string&` 或 C++17 的 `string_view` 作为参数类型。后者是最理想的情况，因为即使在只有 C 字符串的情况，也不会引发不必要的内存复制。
- 如果需要在函数内修改字符串内容、但不影响调用者的该字符串，使用 `string` 作为参数类型（自动拷贝）。
- 如果需要改变调用者的字符串内容，使用 `string&` 作为参数类型（通常不推荐）。

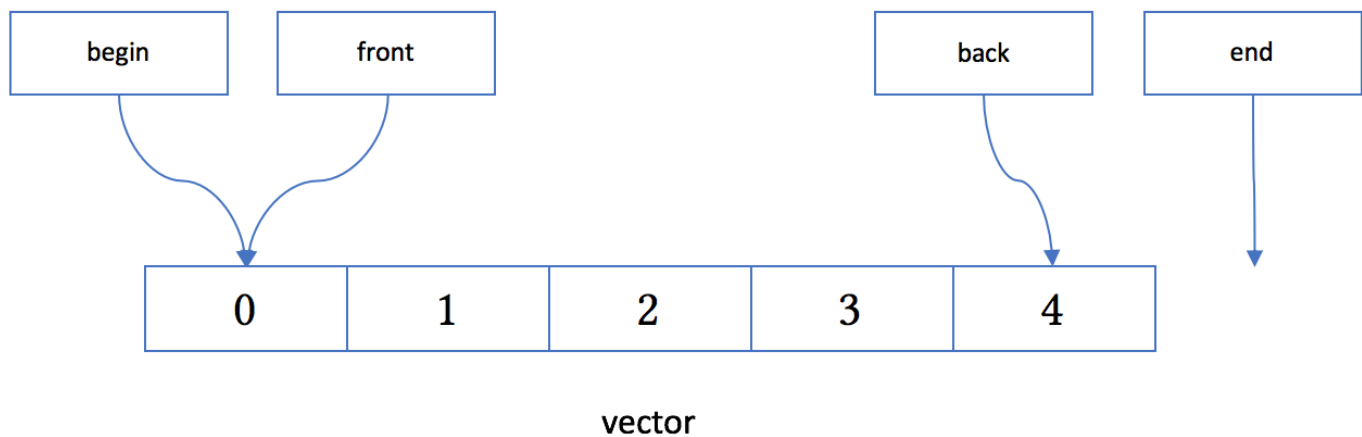
估计大部分同学对 `string` 已经很熟悉了。我们在此只给出一个非常简单的小例子：

```
string name;
cout << "What's your name? ";
getline(cin, name);
cout << "Nice to meet you, " << name
    << "!\n";
```

vector

vector 应该是最常用的容器了。它的名字“向量”来源于数学术语，但在实际应用中，我们把它当成动态数组更为合适。它基本相当于 Java 的 ArrayList 和 Python 的 list。

和 string 相似，vector 的成员在内存里连续存放，同时 begin、end、front、back 成员函数指向的位置也和 string 一样，大致如下图所示：



除了容器类的共同点，vector 允许下面的操作（不完全列表）：

- 可以使用中括号的下标来访问其成员（同 string）
- 可以使用 data 来获得指向其内容的裸指针（同 string）
- 可以使用 capacity 来获得当前分配的存储空间的大小，以元素数量计（同 string）
- 可以使用 reserve 来改变所需的存储空间的大小，成功后 capacity() 会改变（同 string）
- 可以使用 resize 来改变其大小，成功后 size() 会改变（同 string）
- 可以使用 pop_back 来删除最后一个元素（同 string）
- 可以使用 push_back 在尾部插入一个元素（同 string）
- 可以使用 insert 在指定位置前插入一个元素（同 string）
- 可以使用 erase 在指定位置删除一个元素（同 string）
- 可以使用 emplace 在指定位置构造一个元素
- 可以使用 emplace_back 在尾部新构造一个元素

大家可以留意一下 push_... 和 pop_... 成员函数。它们存在时，说明容器对指定位置的删除和插入性能较高。vector 适合在尾部操作，这是它的内存布局决定的。只有在尾部插入和删除时，其他元素才会不需要移动，除非内存空间不足导致需要重新分配内存空间。

当 push_back、insert、reserve、resize 等函数导致内存重分配时，或当 insert、erase 导致元素位置移动时，vector 会试图把元素“移动”到新的内存区域。vector 通常保证强异常安全性，如果元素类型没有提供一个**保证不抛异**

常的移动构造函数，`vector` 通常会使用拷贝构造函数。因此，对于拷贝代价较高的自定义元素类型，我们应当定义移动构造函数，并标其为 `noexcept`，或只在容器中放置对象的智能指针。这就是为什么我之前需要在 `smart_ptr` 的实现中标上 `noexcept` 的原因。

下面的代码可以演示这一行为：

```
#include <iostream>
#include <vector>

using namespace std;

class Obj1 {
public:
    Obj1()
    {
        cout << "Obj1()\n";
    }
    Obj1(const Obj1&)
    {
        cout << "Obj1(const Obj1&)\n";
    }
    Obj1(Obj1&&)
    {
        cout << "Obj1(Obj1&&)\n";
    }
};

class Obj2 {
public:
    Obj2()
    {
        cout << "Obj2()\n";
    }
    Obj2(const Obj2&)
    {
        cout << "Obj2(const Obj2&)\n";
    }
    Obj2(Obj2&&) noexcept
    {
        cout << "Obj2(Obj2&&)\n";
    }
};

int main()
```

```

{
    vector<Obj1> v1;
    v1.reserve(2);
    v1.emplace_back();
    v1.emplace_back();
    v1.emplace_back();

    vector<Obj2> v2;
    v2.reserve(2);
    v2.emplace_back();
    v2.emplace_back();
    v2.emplace_back();
}

```

我们可以立即得到下面的输出：

```

Obj1()
Obj1()
Obj1()
Obj1(const Obj1&)
Obj1(const Obj1&)
Obj2()
Obj2()
Obj2()
Obj2(Obj2&&)
Obj2(Obj2&&)

```

Obj1 和 Obj2 的定义只差了一个 `noexcept`，但这个小小的差异就导致了 `vector` 是否会移动对象。这点非常重要。

C++11 开始提供的 `emplace...` 系列函数是为了提升容器的性能而设计的。你可以试试把 `v1.emplace_back()` 改成 `v1.push_back(Obj1())`。对于 `vector` 里的内容，结果是一样的；但使用 `push_back` 会额外生成临时对象，多一次（移动或拷贝）构造和析构。如果是移动的情况，那会有小幅性能损失；如果对象没有实现移动的话，那性能差异就可能比较大了。

现代处理器的体系架构使得对连续内存访问的速度比不连续的内存要快得多。因而，`vector` 的连续内存使用是它的一大优势所在。当你不知道该用什么容器时，缺省就使用 `vector` 吧。

`vector` 的一个主要缺陷是大小增长时导致的元素移动。如果可能，尽早使用 `reserve` 函数为 `vector` 保留所需的内存，这在 `vector` 预期会增长很大时能带来很大的性能提升。

deque

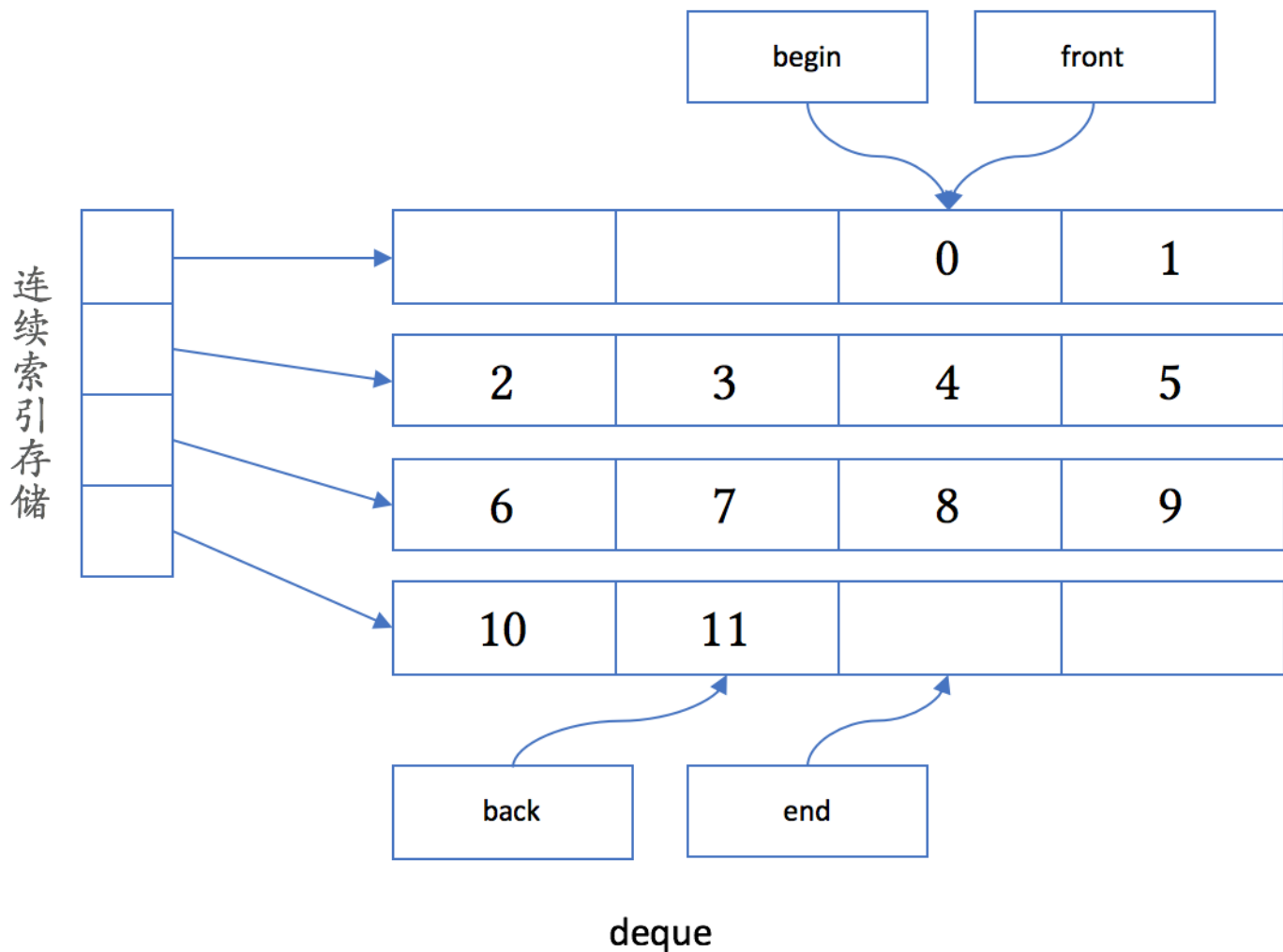
`deque` 的意思是 `double-ended queue`，双端队列。它主要是用来满足下面这个需求：

- 容器不仅可以从尾部自由地添加和删除元素，也可以从头部自由地添加和删除。

`deque` 的接口和 `vector` 相比，有如下的区别：

- deque 提供 `push_front`、`emplace_front` 和 `pop_front` 成员函数。
- deque 不提供 `data`、`capacity` 和 `reserve` 成员函数。

deque 的内存布局一般是这样的：



可以看到：

- 如果只从头、尾两个位置对 deque 进行增删操作的话，容器里的对象永远不需要移动。
- 容器里的元素只是部分连续的（因而没法提供 `data` 成员函数）。
- 由于元素的存储大部分仍然连续，它的遍历性能是比较高的。
- 由于每一段存储大小相等，deque 支持使用下标访问容器元素，大致相当于 `index[i / chunk_size][i % chunk_size]`，也保持高效。

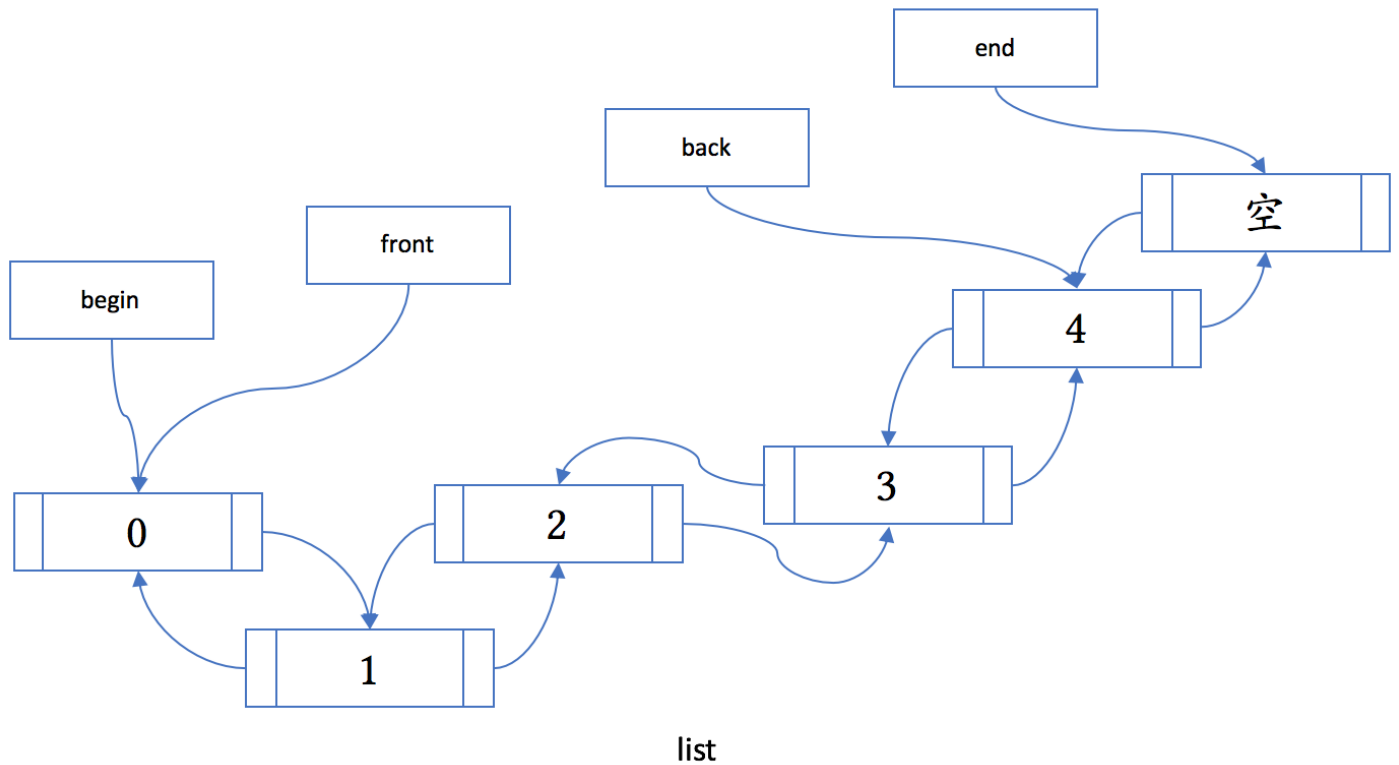
如果你需要一个经常在头尾增删元素的容器，那 deque 会是个合适的选择。

list

list 在 C++ 里代表双向链表。和 vector 相比，它优化了在容器中间的插入和删除：

- list 提供高效的、 $O(1)$ 复杂度的任意位置的插入和删除操作。
- list 不提供使用下标访问其元素。
- list 提供 `push_front`、`emplace_front` 和 `pop_front` 成员函数（和 deque 相同）。
- list 不提供 `data`、`capacity` 和 `reserve` 成员函数（和 deque 相同）。

它的内存布局一般是下图这个样子：



需要指出的是，虽然 `list` 提供了任意位置插入新元素的灵活性，但由于每个元素的内存空间都是单独分配、不连续，它的遍历性能比 `vector` 和 `deque` 都要低。这在很大程度上抵消了它在插入和删除操作时不需要移动元素的理论性能优势。如果你不太需要遍历容器、又需要在中间频繁插入或删除元素，可以考虑使用 `list`。

另外一个需要注意的地方是，因为某些标准算法在 `list` 上会导致问题，`list` 提供了成员函数作为替代，包括下面几个：

- `merge`
- `remove`
- `remove_if`
- `reverse`
- `sort`
- `unique`

下面是一个示例（以 `xeus-cling` 的交互为例）：

```
#include <algorithm>
#include <list>
#include <vector>
using namespace std;
```

```
list<int> lst{1, 7, 2, 8, 3};
vector<int> vec{1, 7, 2, 8, 3};
```

```
sort(vec.begin(), vec.end());    // 正常
// sort(lst.begin(), lst.end()); // 会出错
lst.sort();                      // 正常
```

```
lst // 输出 { 1, 2, 3, 7, 8 }
```

```
vec // 输出 { 1, 2, 3, 7, 8 }
```

如果不用 `xeus-cling` 的话，我们需要做点转换：

- 把 `using namespace std;` 后面的部分放到 `main` 函数里。
- 文件开头加上 `#include "output_container.h"` 和 `#include <iostream>`。
- 把输出语句改写成 `cout << ... << endl;`。

这次我会给一下改造的示例（下次就请你自行改写了）：

```
#include "output_container.h"
#include <iostream>
#include <algorithm>
#include <list>
#include <vector>
using namespace std;

int main()
{
    list<int> lst{1, 7, 2, 8, 3};
    vector<int> vec{1, 7, 2, 8, 3};

    sort(vec.begin(), vec.end());    // 正常
    // sort(lst.begin(), lst.end()); // 会出错
    lst.sort();                      // 正常

    cout << lst << endl;
    // 输出 { 1, 2, 3, 7, 8 }

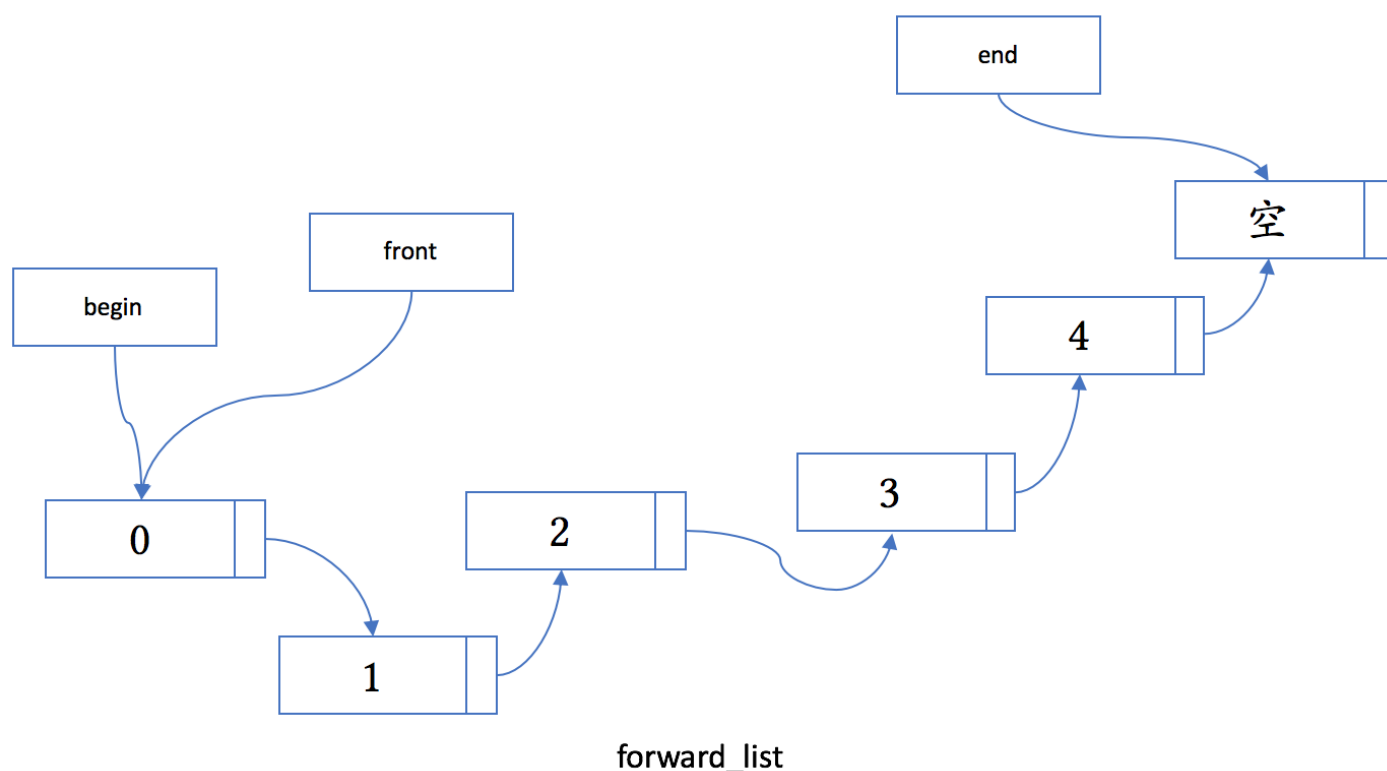
    cout << vec << endl;
    // 输出 { 1, 2, 3, 7, 8 }
}
```

forward_list

既然 `list` 是双向链表，那么 C++ 里有没有单向链表呢？答案是肯定的。从 C++11 开始，前向列表 `forward_list` 成了标

准的一部分。

我们先看一下它的内存布局：



大部分 C++ 容器都支持 `insert` 成员函数，语义是从指定的位置之前插入一个元素。对于 `forward_list`，这不是一件容易做到的事情（想一想，为什么？）。标准库提供了一个 `insert_after` 作为替代。此外，它跟 `list` 相比还缺了下面这些成员函数：

- `back`
- `size`
- `push_back`
- `emplace_back`
- `pop_back`

为什么会需要这么一个阉割版的 `list` 呢？原因是，在元素大小较小的情况下，`forward_list` 能节约的内存是非常可观的；在列表不长的情况下，不能反向查找也不是个大问题。提高内存利用率，往往就能提高程序性能，更不用说在内存可能不足时的情况了。

目前你只需要知道这个东西的存在就可以了。如果你觉得不需要用到它的话，也许你真的不需要它。

queue

在结束本讲之前，我们再快速讲两个类容器。它们的特别点在于它们都不是完整的实现，而是依赖于某个现有的容器，因而被称为容器适配器（container adaptor）。

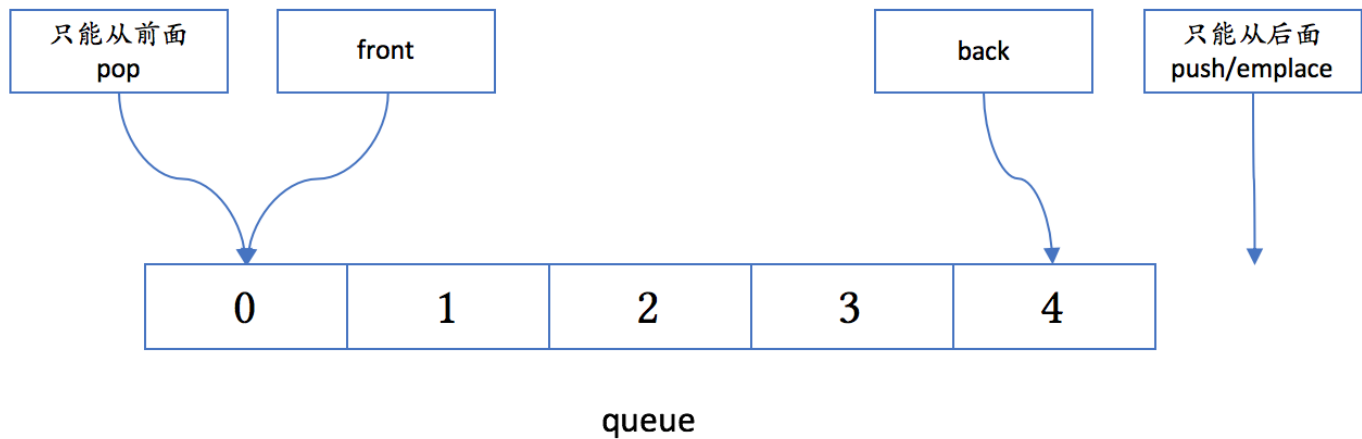
我们先看一下队列 `queue`，先进先出（FIFO）的数据结构。

`queue` 缺省用 `deque` 来实现。它的接口跟 `deque` 比，有如下改变：

- 不能按下标访问元素

- 没有 `begin`、`end` 成员函数
- 用 `emplace` 替代了 `emplace_back`，用 `push` 替代了 `push_back`，用 `pop` 替代了 `pop_front`；没有其他的 `push_...`、`pop_...`、`emplace...`、`insert`、`erase` 函数

它的实际内存布局当然是随底层的容器而定的。从概念上讲，它的结构可如下所示：



鉴于 `queue` 不提供 `begin` 和 `end` 方法，无法无损遍历，我们只能用下面的代码约略展示一下其接口：

```
#include <iostream>
#include <queue>

int main()
{
    std::queue<int> q;
    q.push(1);
    q.push(2);
    q.push(3);
    while (!q.empty()) {
        std::cout << q.front()
                  << std::endl;
        q.pop();
    }
}
```

这个代码的输出就不用解释了吧。哈哈。

stack

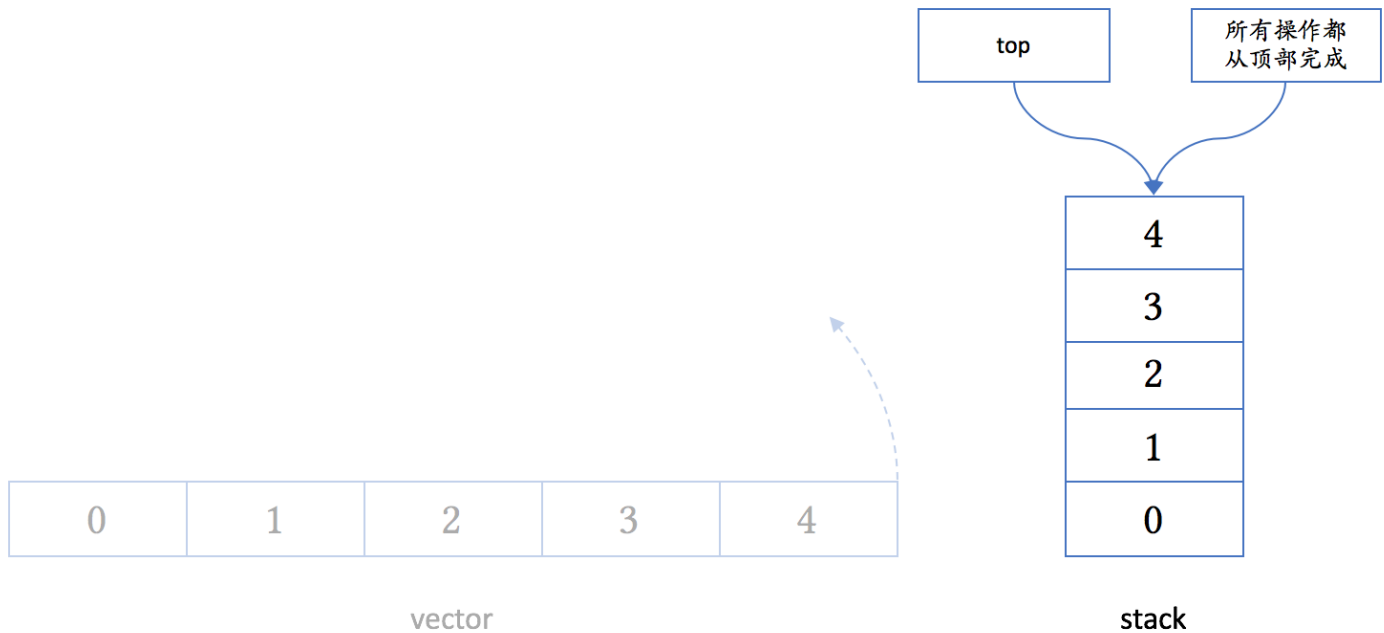
类似地，栈 `stack` 是后进先出（LIFO）的数据结构。

`stack` 缺省也是用 `deque` 来实现，但它的概念和 `vector` 更相似。它的接口跟 `vector` 比，有如下改变：

- 不能按下标访问元素
- 没有 `begin`、`end` 成员函数
- `back` 成了 `top`，没有 `front`

- 用 `emplace` 替代了 `emplace_back`，用 `push` 替代了 `push_back`，用 `pop` 替代了 `pop_back`；没有其他的 `push_...`、`pop_...`、`emplace...`、`insert`、`erase` 函数

一般图形表示法会把 `stack` 表示成一个竖起的 `vector`：



这里有一个小细节需要注意。`stack` 跟我们前面讨论内存管理时的栈有一个区别：在这里下面是低地址，向上则地址增大；而我们讨论内存管理时，高地址在下面，向上则地址减小，方向正好相反。提这一点，是希望你在有需要检查栈结构时不会因此而发生混淆；在使用 `stack` 时，这个区别通常无关紧要。

示例代码和上面的 `stack` 相似，但输出正好相反：

```
#include <iostream>
#include <stack>

int main()
{
    std::stack<int> s;
    s.push(1);
    s.push(2);
    s.push(3);
    while (!s.empty()) {
        std::cout << s.top()
                  << std::endl;
        s.pop();
    }
}
```

内容小结

本讲我们介绍了 C++ 里面的序列容器和两个容器适配器。通过本讲的介绍，你应该已经对容器有了一定的理解和认识。下一

讲我们会讲完剩余的标准容器。

课后思考

留几个问题请你思考一下：

1. 今天讲的容器有哪些共同的特点？
2. 为什么 C++ 有这么多不同的序列容器类型？
3. 为什么 `stack`（或 `queue`）的 `pop` 函数返回类型为 `void`，而不是直接返回容器的 `top`（或 `front`）成员？

欢迎留言和我交流你的看法。

参考资料

[1] cppreference.com, “Containers library”. <https://en.cppreference.com/w/cpp/container>

[1a] cppreference.com, “容器库”. <https://zh.cppreference.com/w/cpp/container>

[2] QuantStack, xeus-cling. <https://github.com/QuantStack/xeus-cling>

[3] 吴咏炜, output_container. https://github.com/adah1972/output_container/blob/master/output_container.h

精选留言



中年男子

我发现老师的问题基本都可以在文章中找到答案，

1、2就不说了，3说一下我的理解:引用老师在vector那段的话 `stack(queue)`为保证强异常安全性，如果元素类型没有提供一个保证不抛异常的移动构造函数，通常会使用拷贝构造函数，而`pop`作用是释放元素，c++98还没有移动构造的概念，所以如果返回成员，必须要调用拷贝构造函数，这时分配空间可能出错，导致构造失败，要抛出异常，所以没必要返回成员。

2019-12-04 23:48

作者回复

很棒。

异常安全是关键。

2019-12-05 09:29



YouCompleteMe

1.都是线性容器

2.不同容器功能，效率不一样

3.实现`pop`时返回元素时，满足强异常安全，代码实现复杂，可读性差。

2019-12-04 12:45

作者回复

3的正解终于出现，有人说到“异常安全”了。

再说两句，这是C++98时设计的接口，没有移动就只能那样。有了移动，在多线程的环境里，移动返回加弹出实际上就变得有用了。我对复杂和可读性部分不那么同意。

2019-12-04 21:03



禾桃

请教一个问题，

#1

为什么一定强制移动构造函数不要抛出异常？

移动构造函数抛出异常后，`catch`处理不可以吗？

#2

为什么拷贝构造函数被允许抛出异常？

能麻烦给些代码说明一下吗？

非常感谢！

2019-12-05 11:13

作者回复

catch住也没有用了。仔细想一下，我现在要把vector里的两个对象移到一个新的vector，移第一个成功，第二个时有异常，然后vector该怎么办？现在两个vector都废掉了。

拷贝不影响旧的容器。即使发生异常，至少老的那个还是好的。这就是异常安全。

2019-12-05 18:00



徐凯

第一个问题 今天讲的大多是线性结构的容器，也可以说大多是非关联容器

第二个问题 应该不只是c++ 所有语言都提供了，之所以对其封装是便于使用，不需要用户自己去造轮子。同时有些容器内部有迭代器 与stl算法相结合可以便于实现泛型编程。c++委员会想让c++成为一个多元化的语言支持 面向对象 面向过程 泛型编程

第三个问题 将对容器的操作与获取值的操作分离开，用途会更明确。同时pop由于已经从容器中剔除了那个元素，那么返回的只能是个拷贝不允许返回已销毁元素的引用。这意味着需要一次拷贝构造操作。而top只需要返回指定元素的引用，高效简洁。将两次操作分开使得操作更明确同时没有额外开销。

个人见解 请老师赐教

2019-12-04 09:52

作者回复

挺好。三比其他回答已经进一步了，但还是没有触及到某个关键字。

2019-12-04 12:57



Alice

老师 您好 我是一个c++的初学者，这一讲的容器的概念原理都理解了，就是vector那一段的演示代码推不出老师的结果来，能不能麻烦老师再解释一下那段代码，辛苦老师了！

2019-12-06 20:11

作者回复

关于几次拷贝/移动的问题？参考 hello world 的评论下的廖熊猫的回答：

在插入的时候，你会发现空间不够了，然后开辟新的空间，在新空间先把最后插入的元素放好，然后再依次把以前的元素一个一个挪过来。空间不够的话最后一个元素是没法插入进去的啊，所以没办法移动三次的。

还有我自己的回答：

两者都是要构造第3个对象时空间不足，需要这样：

1. 分配一个足够大的新内存区域。
2. 在上面构造第3个对象。
3. 如果成功（没有异常），再移动/拷贝旧的对象。
4. 全部成功，则析构旧对象，释放旧对象的内存。
5. 如果1出现异常，直接抛出即可；如果2-3出现异常，则析构已成功构造的对象，释放新内存空间，继续抛出异常。

如果不是这个问题。请把问题阐释得更详细些。可以重新开一个新的评论。

2019-12-07 10:56



Encoded



Star

《现代C++实战31讲》第一天

容器汇编1：比较简单的若干容器

一、容器的输出：

1.简单容器(如：vector)输出就是遍历 (v.begin, v.end)

2.复杂容器(如：vector<vector>)就需要工具 xeus-cling

二、string

1.接口中不建议使用const string&，除非确实知道调用者使用的是string，如果函数不对字符串做特殊处理的话用const char* 可以避免在调用字符串的时候构造string

三、vector

1.vector主要缺陷是大小的增长导致的元素移动，如果可能，尽早使用reserve函数为vector保留所需要的内存，在vector预期会增长很大时带来很大的性能提升

四、deque

1.如果需要经常在头尾增删元素内容，deque会合适

五、list

1.list 是双向链表

2.forward_list是单向链表

六、stack

1.后进先出，底层由deque实现

课后思考：

1.容器有哪些共同点

答：都是线性容器，非关联容器

2.为什么C++有那么多不同的序列容器类型

答：不同容器对应实现不同需求，效率不同

3.为什么stack(或者queue) pop函数返回的是void而不是直接返回内容

答：为了保证异常安全，如果返回的成员构造失败就会抛出异常。

2019-12-19 10:00

作者回复

学习很认真，回答也基本抓住要点了，尤其问题 2 和 3。

2019-12-19 12:55



Alice

吴老师您好，我是那个那天问您vector演示代码的学生，还需要接着请教这段代码的一些问题。因为我刚接触c++不久，可能有些基本语法理解的不是很到位还没有那么深，就需要再问几个基础的问题了。就先请教obj1部分的函数吧，先用reserve (2) 预留了两个存储空间，然后接着用emplace_back()在最后面构造新元素，所以说因为有两个新开的空间那么前两次用emplace_back()构造元素成功就调用构造函数抛出两个obj1()不知道理解的对不对？那第三个obj1()是怎么来的呢？后面两个obj1(const obj1&)怎么来的也不是很理解？这里的obj1&为什么要定义成const类型呢？

还有就是我现阶段对构造函数的理解还停留在初始化的意思上理解地还是太浅吧，不知道该怎么再往深理解一下？

麻烦老师再帮我解答一下问题，辛苦老师了

2019-12-11 18:58

作者回复

头两个在已有空间上成功构造。第三个时发现空间不足，系统会请求更大的空间，大小由实现决定（比如两倍）。有了足够的空间后，就会在新空间的第三个的位置构造（第三个obj1），成功之后再把头两个拷贝或移动过来。

2019-12-12 07:16



花晨少年

如果不修改字符串的内容，使用 const string& 或 C++17 的 string_view 作为参数类型。后者是最理想的情况，因为即使在只有 C 字符串的情况，也不会引发不必要的内存复制

没有理解，“只有 C 字符串的情况，也不会引发不必要的内存复制”，对于string_view相对于const string&，能否简单举个例子

2019-12-07 22:09

作者回复

只有const char*，目标是const string&，会引发一个临时string的构造，会导致内存复制。

用string_view当然也会产生临时对象，但string_view不会复制字符串的内容。

2019-12-08 12:23



robonix

老师，文中提到使用v1.push_back会额外生成临时对象，多一次拷贝构造和一次析构。是不是应该改为多一次移动构造和析构呢？

2019-12-16 09:43

作者回复

这个疑问提得好。在目前这个例子里，确实是移动构造而不是拷贝构造。

2019-12-16 13:33



凌云

```
int main()
{
    vector<Obj1> v1;
    //v1.reserve(2);
    v1.emplace_back();
    v1.emplace_back();
    v1.emplace_back();
}
```

输出：

```
Obj1()
Obj1()
Obj1(const Obj1&)
Obj1()
Obj1(const Obj1&)
Obj1(const Obj1&)
```

"vector 适合在尾部操作，这是它的内存布局决定的。只有在尾部插入和删除时，其他元素才会不需要移动，除非内存空间不足导致需要重新分配内存空间"，讲义中的例子，如果未调用reserve，那么是构造，构造，拷贝构造，构造，然后是2次拷贝构造，那么重新分配内存空间的意思是，当指定的空间不足以放下所有元素时，会将前面的元素拷贝一遍？

2019-12-15 22:09

作者回复

会将前面的元素拷贝或移动一遍。

移动的条件文中提到了，元素类型需要“提供一个保证不抛异常的移动构造函数”。

2019-12-16 09:15



神秘的火柴人

老师，文中：

stack

类似地，栈 stack 是后进先出（LIFO）的数据结构。

queue 缺省也是用 deque 来实现，但它的概念和 vector 更相似。它的接口跟 vector 比，有如下改变：

这里queue缺省是不是笔误了，应该是stack吧

2019-12-13 11:11

作者回复

多谢多谢。已经修复。

2019-12-14 16:32



Alice

谢谢老师的回复，老师您说“扩充空间是一个编译器自发进行的操作，没有用户控制。一般会类似于reserve(size() * 2)”，我之前没有遇到过这个知识点，那老师我应该看点什么稍微补充一下这块知识？

2019-12-12 16:58

作者回复

<https://www.cnblogs.com/skyfsm/p/7488053.html>

这篇中文文章说得还比较清楚，可以看一下。

2019-12-12 19:31



Alice

老师 vector obj1演示代码里，在构造第三个元素发现空间不够之后，v1会按我们之前设定好的自动调用v1.reserve (2) 来扩充空间，是这个意思嘛？

2019-12-12 13:16

作者回复

不是。执行 v1.reserve(2) 之后，空间大小就是 2 了。扩充空间是一个编译器自发进行的操作，没有用户控制。一般会类似于 reserve(size() * 2)。

2019-12-12 16:52



robonix

老师，假如移动构造函数被声明为noexcept了，诱导编译器调用移动构造，而此时却又抛异常了，程序也会直接停止吗？

2019-12-11 08:30

作者回复

对，缺省情况下，std::terminate 会被调用。

2019-12-11 18:21



hdongdong123

老师给的工具xeus-cling为啥第二次运行的时候就报错error: redefinition of 'str'

2019-12-09 19:58

作者回复

就像你在代码里对 str 定义两次一样啊。这仍然是 C++，不是 Python。

Restart kernel 可以重新来。

2019-12-09 20:42



皓首不倦

老师您好 第三个问题能不能这样理解 pop作用是弹出最后一个对象 弹出后该对象内存已经脱离容器当前所管理的有效的范围 虽然该内存存在后续有push操作时候还会被重复使用到 但是pop执行完后 该内存逻辑层面看是暂时脱离了容器的管理范围的 显然pop不能将该内存以引用方式传给外面 否则外部会持有一片目前脱离管理的无效内存 外部再对该内存不论读还是写都是不合适的 所以pop如果要返回对象 只能选择拷贝方式返回 会触发拷贝构造 对于内存占用大或者是需要进行深拷贝的对象而言 这个操作开销太大了 所以选择用top 返回可以安全访问的对象引用 而pop就单纯作为退栈操作不返回对象 我个人理解这样设计api 接口是为了避免不安全地访问内存 对比Java的 stack的 pop接口 Java的pop接口就返回了栈顶对象 因为这个对象内存托管给了jvm管理 调用端拿到了这个出栈的对象的引用也不会有访问内存的问题 但是c++如果把对象内存通过引用带给调用端 那调用端就可能直接读写容器内部的私有内存了 这片内存地址随时可能因为容器的扩容行为而变成野地址 对其访问其实并不安全 不知道我这样理解是否正确

2019-12-09 18:55

作者回复

Java是引用语义，返回对象就是返回个指针，没有任何问题。

C++是值语义，以前返回对象只能是拷贝，可能发生异常。一旦发生异常，对象已经被弹出，那它就彻底“丢失”了。

2019-12-09 21:01



花晨少年

老师请问 "在元素大小较小的情况下，forward_list 能节约的内存是非常可观的"

这句话怎么理解呢，元素大小较小的情况下，是指的元素数量小，还是元素本身的值偏小呢。

2019-12-08 00:34

作者回复

指单个元素的大小，sizeof。

2019-12-08 11:35



总统老唐

吴老师，关于 vector 的 emplace_back 有2点疑问：

1, 我的理解, v1 的内存空间是在栈上分配的, 当 v1 的capacity达到最大值时, 需要给 v1 重新分配空间才能存放新的对象, 因为栈帧是连续内存空间, 那么不管是从高到低还是从低到高, 已经分配的地址, 比如v1[0],应该不变, 但是查看v1[0]的地址, 发现有变化, 看起来第三次 `emplace_back` 导致 v1 的地址变化了, 这样一来, 原来存放v1[0] 和 v1[1] 的栈空间不就空了吗, 那是不是导致了栈的内存空间不连续了?

2, 第三次 `emplace_back` 的打印信息显示, 先调用了构造函数, 再调用拷贝函数, 是不是表示, 当 v1 获得新地址后, 是先在新的 v1[2] 上构造新对象, 再把原来 v1[0] 和 v1[1] 中的对象拷贝过来?

2019-12-06 18:07

作者回复

上来就错了。大部分容器都是在堆上分配空间的.....

2 正确。内存分配成功, 新对象构造成功, 才移动/拷贝旧对象。

2019-12-06 19:42



虫二

1.本章节大部分都是非关联容器

2.各容器效率不同, 为了方便使用应用在不同的场景之中

3.在某些特定情况下会引发异常问题

2019-12-04 21:51

作者回复

可以。够言简意赅的。

2019-12-05 09:42

糖

1.线性容器

2.由于不同场景下需要有合适的数据结构与之对应, 比如既然有了deque为何会有queue和stack呢, queue和stack的功能deque也能实现, 而且甚至比queue和stack具有更大的自由, 这是由于在很多情况下接口的自由使得犯错误的几率也就变大, 因此将大多数接口都封装起来减小出错的可能性。以及链表、数组都存在也是因为他们都具有不可代替的一面。

3.JAVA中确实是当pop时返回被pop的值, 而C++中并没有返回该值, 我认为很大程度上是由于C++更注重效率, 毕竟这样做可以减小一次拷贝或者移动, 当容器中存储的对象拷贝或移动很费劲或者多次pop时, 这将大大降低C++的pop速度。另外我认为可能和异常的避免有关, 由于如果需要返回被pop的值, 需要提前将其拷贝到其他地方或者是移动到其他地方, 这两者都可能需要内存的分配, 所以可能会出现异常。第二点纯属脑洞, 不清楚自己考虑的对不对, 希望老师指正。。。

2019-12-04 20:39

作者回复

还是重点谈3。对于C++的情况, 基本没问题。对于Java, 则错了。Java的情况最接近于你返回一个智能指针——这个操作本身性能是没问题的。主要约束是必须在堆上放置对象。

2 你对防犯错的考虑非常好。其他人似乎没提到。

2019-12-05 09:39