Modern cpp 30 lectures — simple container

Thursday, December 17, 2020 7:49 AM

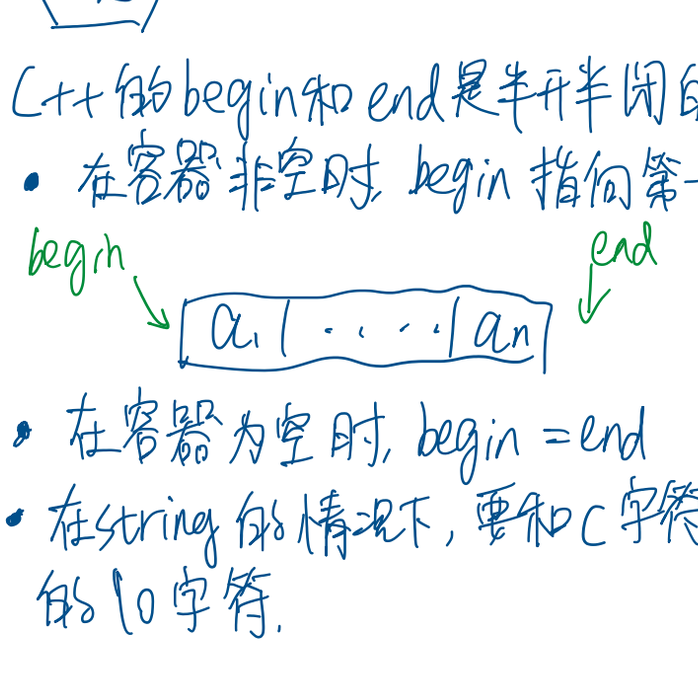
对容器,一个麻烦点是无法直接输出其内容.
Eg 定义一个 vector<int> v, 无法简单输出内容.
用 Copy(v.begin(), v.end(), ostream_iterator(...)), 对 map 或 vector<vector<...>> 这样的复杂类型无效,
推荐使用 xeus-cling, 可以交互显式容器内容.

String

String 是模板 basic_string 对于 char 类型的特化, 是一个只存放字符 char 类型数据的容器. 真正的容器类与 string 最大不同是可存放任意类型的对象.
String 具有下列成员函数
- begin 可得对象起始点.
- end 可得对象结束点.
- empty 可得对象是否为空
- size 可得容器大小
- Swap 可以和另外一个容器交换内容.

注意

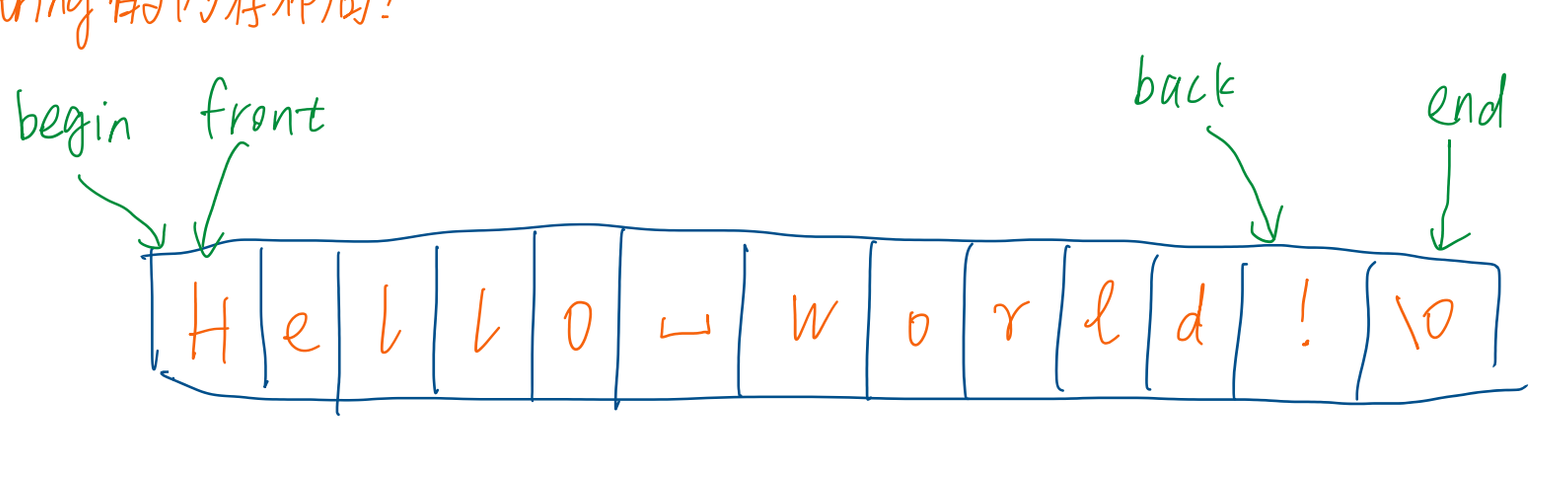
C++ 的 begin 和 end 是半开半闭的区间.
- 在容器非空时, begin 指向第一个元素, end 指向最后一个元素后面.



- 在容器为空时, begin = end
- 在 string 的情况下, 要和 C 字符串兼容, end 指向代表字符串结尾的 \0 字符.

容器的共同点:
- 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

String 的内存布局:



String 和 C 字符串的不同点:
- 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, eg +, +=
- String 支持字符串的查找, eg. find, rfind.
- String 支持从 istream 安全地读入字符串, e.g. 使用 getline
- String 支持给期待 const char* 的接口传递字符串内容 (用 c_str)
- String 支持到数字的互转, Eg. stoi, to_string.

Eg.

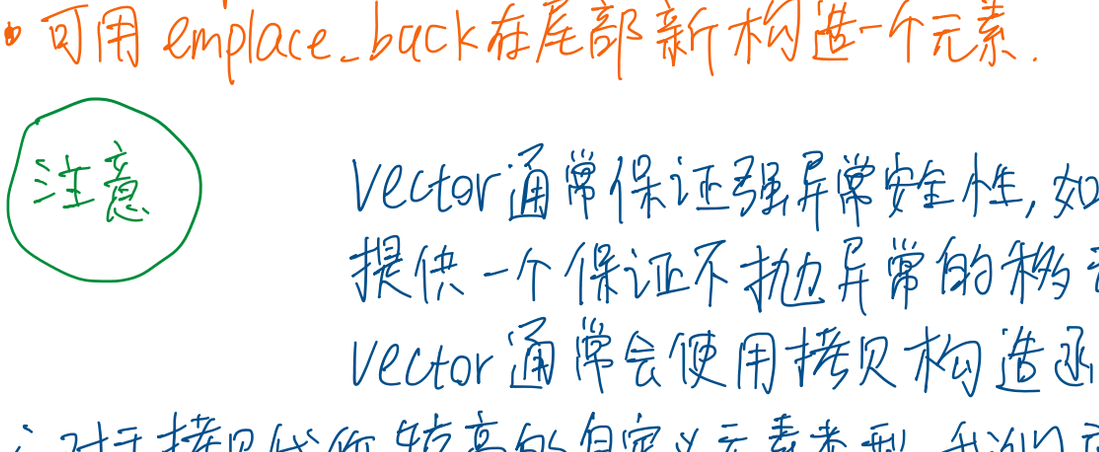
```
String name
cout << "What's your name?";
getline (cin, name);
cout << "Nice to meet you, " << name << "!\n";
```

注意.

- 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. 使用 const char* 可避免在调用者只有 C 字符串时, 编译器自动构造 string.
- 如果不需要修改字符串的内容, 使用 const string & 或 C++17 的 string_view 作为参数. String_view 是最理想的, 因为即使只有 C 字符串, 也不会有不必要的内存复制.

Vector

动态数组, 相当于 Java 的 ArrayList 和 Python 的 list.



除了容器的共同点, vector 允许下面的操作
- 可以使用中括号的下标访问成员, 跟 String 一样.
- 可以使用 data 获取指向其内容的裸指针, 同 string
- 可以用 capacity 获得当前分配的存储空间大小, 以元素数量计, 同 string.
- 可以用 reserve 改变存储空间大小, 成功后 capacity() 会改变, 同 string.
- 可以用 resize 改变大小, 成功后 size() 会改变, 同 string
- 可用 pop_back 删除最后一个元素, 同 string.
- 可用 push_back 在尾部插入元素, 同 string.
- 可用 insert 在指定位置插入元素, 同 string.
- 可用 erase 在指定位置删除元素, 同 string.
- 可用 emplace 在指定位置构造元素.
- 可用 emplace_back 在尾部新构造一个元素.

注意 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,
∴ 对于拷贝代价较高的自定义元素类型, 我们应定义移动构造函数, 并标为 noexcept, 或只在容器中放置对象的智能指针. 这就是上一章 smart_ptr 的实现中标上 noexcept 的原因. Eg.

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

using namespace std;

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

  Obj1(const obj&)
  {
    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();
}
```

Output:

```
Obj1()
Obj1()
Obj1()
Obj1(Const obj1&)
Obj1(Const obj1&)
Obj2()
Obj2()
Obj2()
Obj2(Obj2&&)
Obj2(Obj2&&)
```

解释:
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

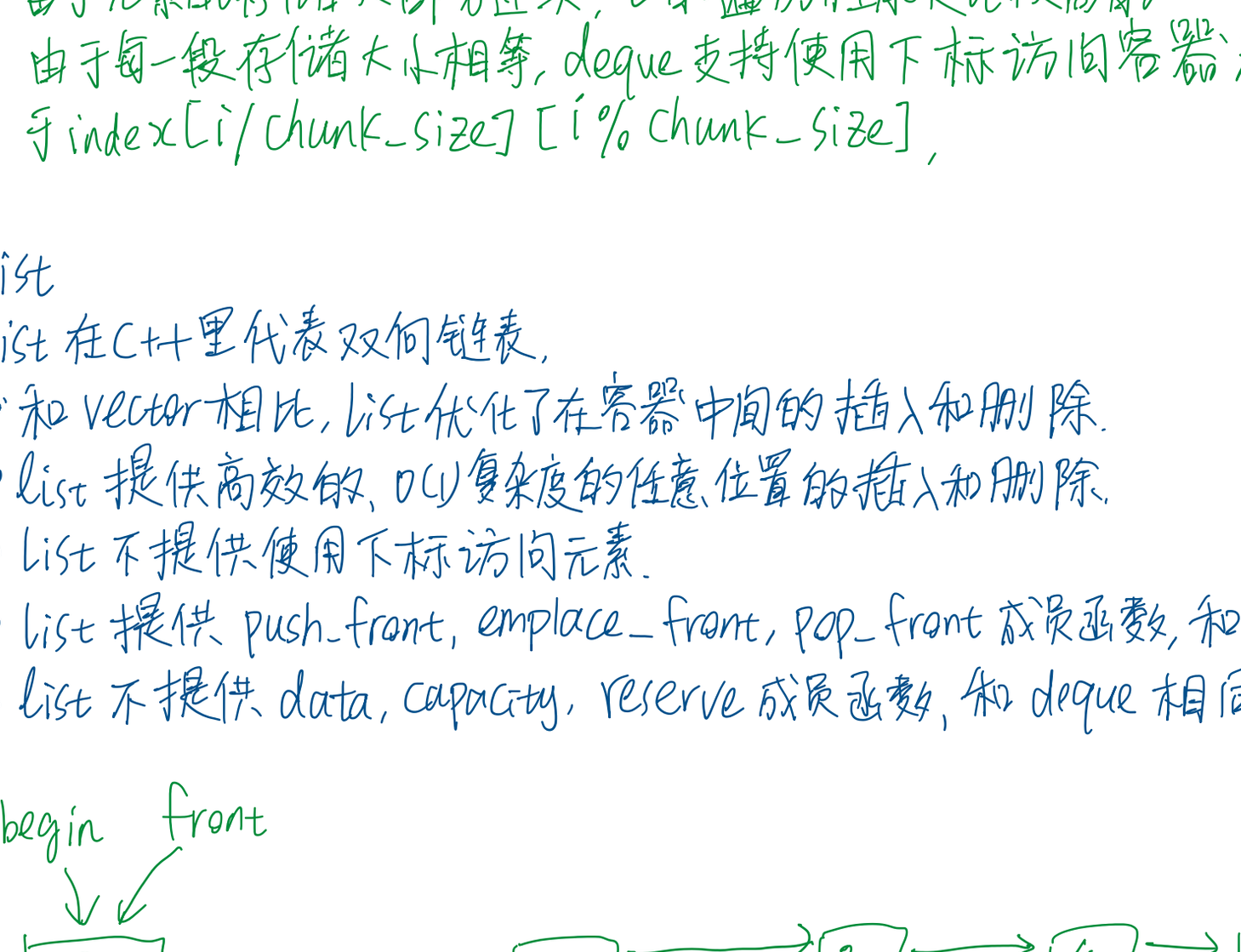
Obj1 和 Obj2 的定义只差了一个 noexcept, 但这个小小的差异就导致了 vector 是否会移动对象.

C++ 提供 emplace 系列函数是为了提升容器性能. 如果把 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, 多一次拷贝构造和一次析构.

Tips:
- 对于现代处理器, 访问连续内存比访问不连续内存快的多, 所以 vector 的连续内存使用是它的一大优势. 如果不知道用什么容器时, 缺省可用 vector.
- vector 的一个主要缺陷是大小增长时导致的元素移动, 应尽早使用 reserve 函数, 为 vector 保留所需内存.

deque

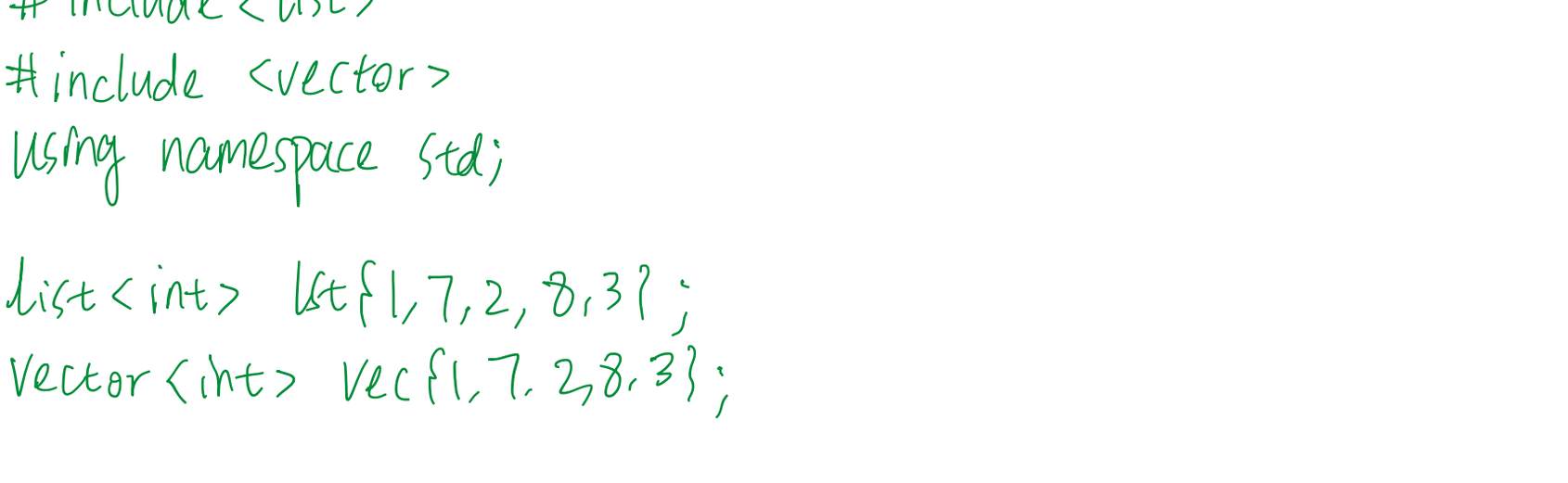
双端队列 double-ended queue, 主要用来满足: 容器不仅可以从尾部自由的添加和删除元素, 也可以从头部自由地添加和删除.
deque 的接口和 vector 相比, 有如下区别:
- deque 提供 push_front, emplace_front, pop_front 成员函数.
- deque 不提供 data, capacity, reserve 成员函数.



- 如果只从头, 尾两个位置对 deque 进行增删操作, 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],

list

list 在 C++ 里代表双向链表.
- 和 vector 相比, list 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.
- list 不提供使用下标访问元素.
- list 提供 push_front, emplace_front, pop_front 成员函数, 和 deque 相同.
- list 不提供 data, capacity, reserve 成员函数, 和 deque 相同.



- list 的一些成员函数:
  - merge
  - remove
  - remove_if
  - reverse
  - sort
  - unique

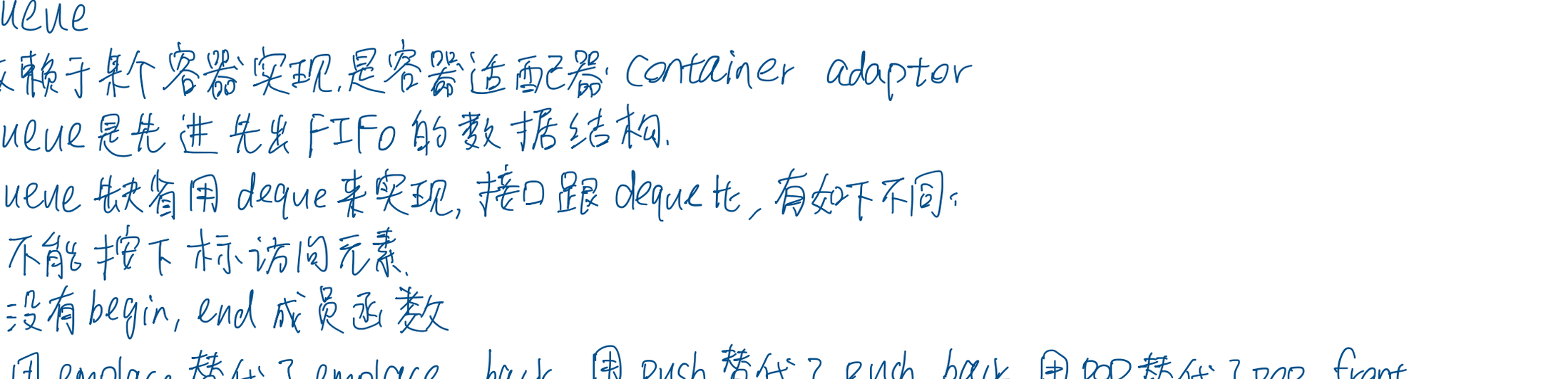
Eg.

```
#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());
sort(lst.begin(), lst.end()); ← 错误
lst.sort();
```

Output:
lst = {1, 2, 3, 7, 8}.
vec = {1, 2, 3, 7, 8}.

forward_list

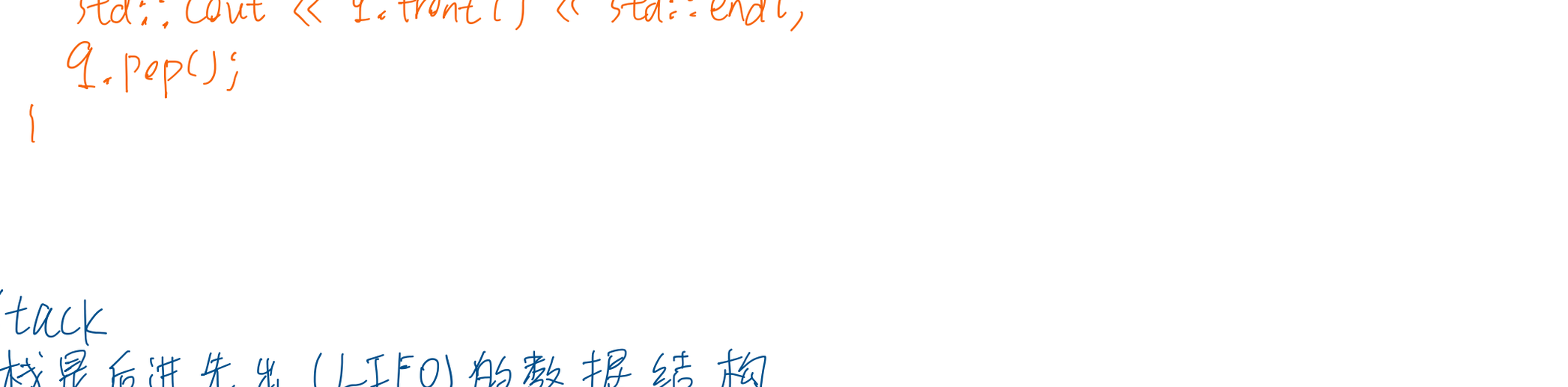
从 C++11 开始的单向链表.



- 大部分 C++ 容器都支持 insert 成员函数 (从指定位置之前插入一个元素), 对于 forward_list, 这不容易做到, 标准库提供了 insert_after 替代.
- forward_list 跟 list 比, 还缺少:
  - back
  - size
  - push_back
  - emplace_back
  - pop_back
- forward_list 的优势: 节约内存, 在单个元素大小较小 (sizeof), 节约的内存很可观.

queue

依赖于某个容器实现, 是容器适配器 container adaptor
queue 是先进先出 FIFO 的数据结构.
queue 缺省用 deque 来实现, 接口跟 deque 比, 有如下不同:
- 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 push_xxx, pop_xxx, emplace_xxx, insert, erase 函数

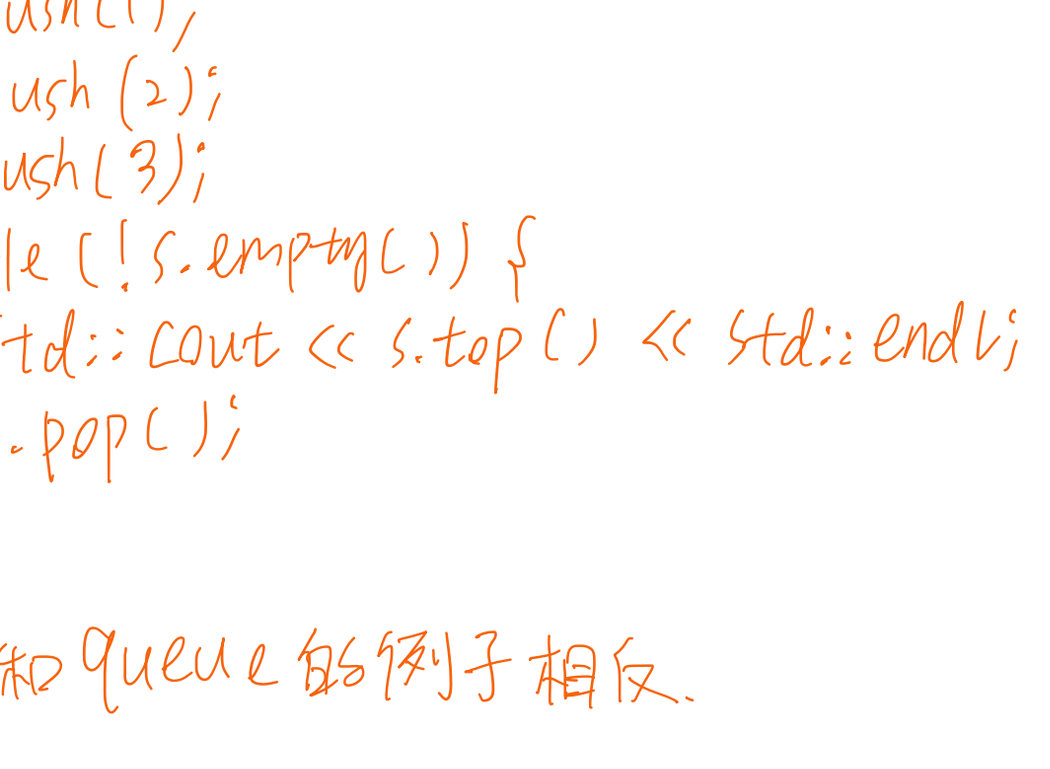


Eg.

```
#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();
  }
}
```

Stack

栈是后进先出 (LIFO) 的数据结构.
Stack 缺省也是用 deque 来实现, 但概念上跟 vector 更相似. 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 变成了 top, 没有 front
- 用 emplace 替代了 emplace_back, push 替代了 push_back, pop 替代了 pop_back,
- 没有 push_xxx, pop_xxx, emplace_xxx, insert, erase 函数



Eg.

```
#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();
  }
}
```

输出和 queue 的例子相反.

Tips
- stack/queue 的 pop 函数返回类型为 void, 而不是直接返回 top/front 成员, 是为了确保异常安全.
- 强制移动构造函数不抛异常, Eg. 要把 vector 里的两个对象移到一个新的 vector, 移第1个成功, 第2个时有异常, 现在两个 vector 都废掉了. 拷贝不影响旧的容器. 即使发生异常, 旧的那个还是好的, 这就是异常安全.
- 大部分容器都是在堆上分配空间的.
- 只有内存分配成功, 新对象构造成功, 才移动/拷贝旧对象.