Lecture 7

TOC

STL 容器

STL(Standard Template Library)是 C++ 标准库的一部分，提供了一系列的模板类和函数，用于实现常见的数据结构和算法。

- array: 固定大小数组
- vector: 动态数组, 支持随机访问, 动态扩容
- deque: 双端队列, 更低的首尾插入删除开销
- list: 双向链表
- forward_list: 单向链表
- stack: 栈
- queue: 队列
- priority_queue: 优先队列
- set: 有序集合
- map: 有序映射
- unordered_set: 无序集合
- unordered_map: 无序映射

std::array

定长数组，支持随机访问，不支持动态扩容。可以使用迭代器访问。

```cpp
std::array<int, 10> arr;
arr[0] = 1;
arr[1] = 2;
int a = arr.front();
int b = arr.back();
arr.fill(0);
```

Reference

vector

可动态调整大小的数组，支持插入等操作。

> 在已知元素个数的情况下，在构造函数中显示指定容器大小可以提高效率。

```
std::vector<int> v(10);
v.push_back(1);
v.insert(v.begin(), 2);
int a = v.front();
int b = v.back();
v.clear();
```

Reference

deque

双端队列，支持首尾插入删除，相比 vector 在首尾插入删除的开销更低。什么是队列(queue)

```
std::deque<int> d;
d.push_back(1);
d.push_front(2);
d.pop_back();
d.pop_front();
```

链表

关于链表的概念：Introduction

- `forward_list`：只能正向便利
- `list`：可以正向和反向遍历

任意位置插入为常数时间复杂度

什么是复杂度

Wrapper

- `queue` : https://oi-wiki.org/ds/queue/
- `stack` : https://oi-wiki.org/ds/stack/
- `priority_queue` : https://oi-wiki.org/ds/binary-heap/

这三类容器是对其他容器的封装，提供了特定的接口。构造时可以指定底层容器。

```
std::queue<int> q;
std::stack<int, std::vector<int>> s;
std::priority_queue<int, std::deque<int>, std::greater<int>> pq;
```

关联容器(associate container)

- `set` : https://oi-wiki.org/ds/rbtree/
- `map` : https://oi-wiki.org/ds/rbtree/

set

set 存储了一组已排序的元素，支持插入、删除、查找操作，时间复杂度 O(logn)。

```
std::set<int> a;
const auto comp = [](const int& lhs, const int& rhs) { return lhs < rhs; };
std::set<Color, decltype(comp)> b(comp);
```

如果要允许重复元素，可以使用 `multiset` 。

map

map 存储了一组键值对(key-value pair)，支持插入、删除、查找操作，时间复杂度 O(logn)。key 具有唯一性，每个key对应一个value。若要一个key对应多个value，可以使用 `multimap`。

```
std::map<int, int> a;
a.insert({1, 2});
a[3] = 4;
if(a.find(1) =/= a.end()) {
    a.erase(1);
} else {
  std::cout << "Not found" << std::endl;
}
```

无序关联系容器 Unordered associative container

- `unordered_set`: https://oi-wiki.org/ds/hash/
- `unordered_map`: https://oi-wiki.org/ds/hash/

二者一般都基于哈希表实现，插入，删除，搜索有均摊的常数时间复杂度。

迭代器（iterator）

在使用 STL 容器时，很多无序容器无法按顺序访问。迭代时常见的操作，遍历所有存储对象并执行某些操作。
所有STL容器都提供了迭代器。

Iterator

迭代器本质为满足一定函数的结构体，可以用来迭代容器对象。

STL 容器常用成员函数:

- `begin()`: 返回指向第一个成员的迭代器
- `rbegin()`： 反向迭代时的第一个成员的迭代器，仅适用于 sequence container
- `end()`：返回迭代最后一个元素后位置的迭代器，用于判断迭代结束
- `rend()`：同上类似
- `cbegin()`, `cend()`, `crbegin()`, `crend()`： const 迭代器，只能访问不能修改

C++20后提供了 Concept 来更加方便的实现迭代器行为

排序

- `std::sort`：常见实现为内省排序，保证为O($n \log n$)时间复杂度
- C++20后可以使用 `std::ranges::sort` 省去手动输入迭代范围

```cpp
std::array<int,5> a = {5,2,3,5,1};
std::sort(a.begin(),a,end(),[](auto &l, auto &r){return l < r;});
for(int atom:a){
  std::cout<<a<<' ';
}
std::endl(std::cout);
```

Sort Algorithm

std algorithm

Utility

Variant(C++17)

C++ 17 加入了 `std::variant` 作为类型安全的 union。提供了读取时的类型检查，访问失败时抛出 `std::bad_variant_access` 异常。

```
std::variant<int, double, std::string> v;
v = 42;
v = 3.14;
v = "Hello, World!";
std::cout << std::get<std::string>(v) << std::endl;

auto visitor = [](const auto& value) {
  std::cout << "Value: " << value << std::endl;
};

std::visit(visitor, v);
```

Reference

std::any(C++17)

更加安全的void*指针,且不需要手动管理内存。

- std::any能存储任意类型的变量，但访问时需要使用`std::any_cast`进行显式类型转换。转换失败抛出std::bad_any_cast异常。

```
std::any a;
a = 42;
a = 3.14;
a = std::string("Hello, World!");

if (a.has_value()) {
  try {
    std::cout << std::any_cast<std::string>(a) << std::endl;
  } catch (const std::bad_any_cast& e) {
    std::cout << "bad cast: " << e.what() << std::endl;
  }
}
```

Reference

std::optional

表示可能不存在的值，一般用做函数的返回值

```
std::optional<int> foo() {
  if (...) {
    return 0;
  } else {
    return {}; // return an empty std::optional<int>
  }
  return std::nullopt; // return an empty std::optional<int>
}
std::optional opt = foo(1, 2);
if (opt.has_value()) {}

if (opt){ // equal to `if(opt.has_value())`
  std::cout<<"No value"<<std::endl;
}
```

Reference

std::tuple

`tuple` 是一个辅助类，用于实现其他语言中的元组效果，快速组合任意变量为一个结合体，省去了定义结构体的操作。

```
std::tuple<int, float, double> a;
/* equal to
struct {
  int;
  float;
  double;
} */

a.get<0>() = 1;
std::get<1>(a) = 2.0;
auto [x, y, z] = a;// structured binding
```

`std::pair` 是 `std::tuple` 的特例，只有两个元素，支持使用 `first` 和 `second` 访问元素。

Reference

std::function

- std::function 是一个通用的函数封装类，可以用来封装任何可以调用的目标，如普通函数、lambda表达式、函数指针、成员函数指针等。
- std::function 提过一个 `operator()` 来调用封装的函数对象。

> 只有没有捕获的lambda表达式才能转换为函数指针，但是std::function可以封装任何lambda表达式。

```
std::function<int(int, int)> add = [](int a, int b) {
    return a + b;
};

int result = add(3, 4);
std::cout << "Result: " << result << std::endl;
```

Reference

std::chrono

- timestamp： 自1970年1月1日以来的时间间隔，单位为秒，unix epoch 为 00:00:00 UTC on 1 January 1970。为负数时表示1970年1月1日之前的时间。
- time_t: 用于表示时间的数据类型，通常为整数，单位为秒, 除了非常老的系统，绝大多数系统上该类型为64位整数。

```
auto start = std::chrono::steady_clock::now();
std::this
auto end = std::chrono::steady_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);

std::cout << "elapse: " << duration.count() << " ms" << std::endl;
```

- `std::chrono::system_clock`：系统时钟。now()函数返回系统当前时间戳
- `std::chrono::steady_clock`：递增时钟。now()函数返回的时间戳从一个不确定的时间点开始。
- `std::chrono::high_resolution_clock`：高精度时钟。linux中的实现等同于`steady_clock`。

随机数

C++提供一系列用来生成随机数的函数

- 随机种子：提供固定顺序的随机数生成方式，以保证多次运行的结果一致。
- Example

```cpp
std::random_device rd;
std::mt19937 gen(rd());  // 随机数引擎

// □□□□□□
// unsigned int seed = 42;
// std::mt19937 gen(seed);

std::uniform_int_distribution<> intDist(1, 100); // 均匀分布随机
int randomInt = intDist(gen);

std::normal_distribution<> realDist(0.0, 1.0); // 正态分布随机
double randomReal = realDist(gen);
```

命令行编译

https://blog.vollate.top/2023/12/08/compile-knowledge/#gcc-clang-%E4%BD%BF%E7%94%A8

CMake

https://blog.vollate.top/2024/10/12/cmake-introduction/

Q&A