C++

Лекция 13

Standard Library

Section 1

Intro

Motivation

Стандартная библиотека:

- Предоставляет качественный протестированный и документированный код
- Позволяет избегать переизобретения колеса
- Код, написанный с помощью стандартной библиотеки, проще поддерживать

Standard Library

- Является частью стандарта
- Основана на нескольких библиотеках, которые зарекомендовали себя на момент создания первого стандарта
- Вдохновлена идеями STL-фреймворка: контейнеры, итераторы, функциональные объекты, адаптеры
- Содержит стандартную библиотеку C
- Упрощает работу с вводом/выводом, памятью, строками, потоками, контейнерами и т.д.

Standard Library

Особенности

- `namespace std`[1]
- имена заголовочных файлов[2]
- аллокаторы

```
1) не помещайте свой код в пространство имен std
2) обратите внимание на имена заголовочных файлов из стандартной библиотеки C:
   используйте <cstring> вместо <string.h>
```

Implementations

- llvm `libc++`
- `libstdc++`
- Microsoft STL

Section 2

Containers

Containers

- Sequence Containers: `array`, `vector`, `deque`, `list`, `forward_list`
- Associative
  - Ordered: `set`, `map`, `multiset`, `multimap`
  - Unordered: `unordered_set`, `unordered_map`, ...
- Adaptors: `stack`, `queue`, `priority_queue`, `flat_set`, `flat_map`
- Views: `span`, `mdspan`

Containers

- Sequence Continers: `string`, ...*
- Views: `string_view`

```
*) template <CharT, Traits, Allocator> basic_string
```

std::vector

- имеет специализацию для `std::vector<bool>`

```
std::vector<bool> vs{true, true, false};
for (auto& item : vs) {} // compile-time error
```

- используем `reserve` если известна оценка размера сверху

```
std::vector<Box> boxes;
boxes.reserve(100);
assert(boxes.size() == 0);
assert(boxes.capacity() == 100);
```

- `emplace_back` vs `push_back`
- `operator[]` vs `at`

`std::unordered_*`

- `load_factor`, `max_load_factor`, rehashing
- если возможно оценить количество ключей, то можно задать количество bucket-ов, чтобы избежать лишнего рехеширования

iteration through maps

```
std::map<int, int> m{{1,2},{2,3},{3, 4}};
for (const auto& [key, value] : m) { // structured binding
    std::cout << key << " " << value << std::endl;
}
```

https://en.cppreference.com/w/cpp/language/structured_binding

span

```cpp
#include <vector>
#include <span>
#include <iostream>

void print_vec(std::span<int> s) {
    for (int& i : s) {
        std::cout << i << " ";
        i = 0;
    }
    std::cout << std::endl;
}

int main() {
  std::vector<int> vs{ 1, 2, 3, 4, 5, 6 };
  print_vec(vs);
  print_vec(vs);
}
```

mdspan (since c++23)

```cpp
#include <vector>
#include <mdspan> // (since c++23)
#include <iostream>

int main() {
  std::vector<int> vs{ 1, 2, 3, 4, 5, 6 };
  std::mdspan s{vs.data(), 2, 3}; // (since c++23)

  // multidim operator[] (since c++23)
  std::cout << s[0, 0] << " " << s[1, 2];
}
```

Section 3

Iterators

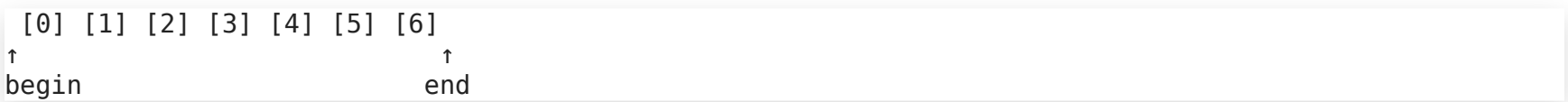
Iterator

- Все рассмотренные контейнеры имеют итераторы: `begin()`,`/end()`, `cbegin()/cend()`
- Итератор — объект, связанный с некоторой позицией в контейнере
- В зависимости от категории задает набор операций над собой
- С точки зрения интерфейса похож на указатель:

```
std::vector v{1,2,3,4,5}; // CTAD
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << " ";
}
```

Iterator

Используя итератор и операцию инкремента можно обойти все элементы контейнера.



Reverse Iterator

```
    std::vector ints{1,2,3};
    std::ostream_iterator<int> os_it(std::cout, " ");
    for (auto it = ints.rbegin(); it != ints.rend(); ++it) {
        *os_it++ = *it;
    }

    assert(ints.rbegin().base() == ints.end());
```


```
    std::vector vs{1,2,3};
    auto it = vs.begin() + 2;
    auto rit = std::reverse_iterator(it);

    assert(rit == vs.rbegin() + 1);
    assert(rit.base() == it);
    assert(vs.rbegin().base() == vs.end());
```

Iterator adapters

- `move_iterator`
- `back_inserter_iterator`
- `front_inserter_iterator`
- `inserter_iterator`
- `istream_iterator`
- `ostream_iterator`

move_iterator

```
    std::vector<std::vector<int>> matrix {{1,2,3},{4,5,6}};
    std::vector<std::vector<int>> m2(
        std::make_move_iterator(matrix.begin()),
        std::make_move_iterator(matrix.end()));
    assert(matrix[0].empty());
```

back_inserter_iterator

```
std::vector<int> v;
std::fill_n(std::back_inserter(v), 10, 2);
assert(v.size() == 10);
assert(v[0] == 2);
```

istream_iterator & ostream_iterator

```
    std::istringstream str("1 2 3 4 5");
    std::partial_sum(std::istream_iterator<int>(str),
                     std::istream_iterator<int>(),
                     std::ostream_iterator<int>(std::cout, ","));
```

Invalidation

Иногда возникает необходимость сохранять сами итераторы, но в некоторых случая они могут инвалидироваться при модификации контейнера:

https://en.cppreference.com/w/cpp/container#Iterator_invalidation

Section 4

Algorithms

<algorithm>

Алгоритмы работают с контейнерами используя итераторы

```
std::vector vs{1,2,3};
std::copy(vs.rbegin(), vs.rend(),
          std::ostream_iterator<int>(std::cout, "_"));
```

Алгоритмы используют категорию итератора, чтобы выбрать наиболее эффективную реализацию

Если контейнер предлагает аналог алгоритма в качестве метода, то лучше использовать его (`std::lower_bound` vs `std::set<T>::lower_bound`)

remove-erase idiom

```
std::vector vs{1,2,3,4,5};

auto it = std::remove(vs.begin(), vs.end(), 3);
std::cout << "vs.size: " << vs.size() << "\n";
vs.erase(it, vs.end());
std::cout << "vs.size: " << vs.size()<< "\n";
```

```
std::vector vs{1,2,3,4,5};
vs.erase(std::remove(vs.begin(), vs.end(), 3), vs.end());
```

Section 5

<memory>

std::shared_ptr

```
class Person {
public:
    Person() = default;

    friend void makeFriendship(
        const std::shared_ptr<Person>& p1,
        const std::shared_ptr<Person>& p2)
    {
        p1->friends.push_back(p2);
        p2->friends.push_back(p1);
    }

    ~Person() { std::cout << "dtor\n"; }
private:
    std::vector<std::shared_ptr<Person>> friends;
};

int main() {
    auto p1 = std::make_shared<Person>();
    auto p2 = std::make_shared<Person>();
    makeFriendship(p1, p2);
}
```

Есть ли проблемы в этом коде?

`shared_ptr` & `weak_ptr`

- internals
- make_shared
- deleter

`shared_ptr` & `weak_ptr`

internals

```
shared_ptr
----------
| ptr ---|-------> [ managed object ]
|        |         ____________________
| refcnt-|-------> | used_cnt         |
----------         | weak_cnt         |
                   --------------------
```

```
    std::shared_ptr<A> sp{new A};
    std::weak_ptr<A> wp = sp;
    std::cout << sizeof(sp) << " " << sizeof(wp);
```

```
(clang-16|gcc-13) output:
16 16
```

libstdc++

libc++

`make_shared`

- only one allocation
- the memory occupied by T persists until weak owners are alive
- no custom deleter

shared_ptr deleter

```
#define LOG() \
    do { std::cout << __PRETTY_FUNCTION__ << std::endl; } while(0)
struct A {
    ~A() { LOG(); }
};

struct MyDeleter {
    void operator()(const A*) { LOG(); } //memleak
};

int main() {
  std::shared_ptr<A> sp{new A, MyDeleter{}};
}
```

```
Output:
void MyDeleter::operator()(const A *)
```

custom deleter

```
struct A {
    ~A() { LOG(); }
};
struct B : A {
    ~B() { LOG(); }
};

int main() {
  std::shared_ptr<A> sp{new B};
}
```

```
Output:
B::~B()
A::~A()
```

weak_ptr

```
std::shared_ptr<A> sp{new B};
std::weak_ptr<A> wp = sp;

if (auto p = wp.lock()) {
  // use p - it's shared_ptr;
}
```

shared_ptr & unique_ptr

```
auto p = std::make_unique<int>(13);
std::shared_ptr<int> sp(std::move(p));
```

Section 6

Utilities

Useful template classes

- `std::any`
- `std::optional`
- `std::variant`
- `std::tuple` / `std::pair`

Section 7

<chrono>

Main concepts

- `clock`
- `time point`
- `time duration`

Use case

```
auto start = std::chrono::steady_clock::now();
payload();
auto end = std::chrono::steady_clock::now();
std::cout << "Elapsed time: " << (end - start).count();
```

Section 8

<regex>

Use cases

- проверить, что текст удовлетворяет паттерну (`std::regex_match`)
- заменить паттерн (`std::regex_replace`)
- найти паттерн в тексте (`std::regex_search`)
- итерация по паттернам в тексте (`std::regex_iterator`)

std::regex_match

```
    std::string good = "dblkjdfkl";
    std::string bad = "ab!bc";

    {
        std::regex rgx("\\w+");
        assert(std::regex_match(good, rgx));
        assert(!std::regex_match(bad, rgx));
    }
```

std::regex_replace

```
std::string target = "a 10.11 b 20.22 c 30.33 d";
std::regex rgx(R"(((\d+)\.(\d+)))");
std::smatch sm;

std::cout << std::regex_replace(target, rgx, "XXXX");
```

std::regex_search

```
    std::string target = "a 10.11 b 20.22 c 30.33 d";
    std::regex rgx(R"(((\d+)\.(\d+)))");
    std::smatch sm;

    if (std::regex_search(target, sm, rgx)) {
        std::cout << "prefix: " << sm.prefix() << std::endl;

        for (size_t i = 0; i < sm.size(); ++i) {
            std::cout << sm[i] << std::endl;
        }

        std::cout << "suffix: " << sm.suffix() << std::endl;
    }
```

std::sregex_iterator

```
std::string target = "a 10.11 b 20.22 c 30.33 d";
std::regex rgx(R"(((\d+)\.(\d+)))");

auto number_begin =
    std::sregex_iterator(target.begin(), target.end(), rgx);
auto number_end = std::sregex_iterator();

for (auto it = number_begin; it != number_end; ++it) {
    std::smatch match = *it;
    std::cout << match.str(0) << " ";
}
```

Section 9

<filesystem>

filesystem library

```
    std::cout << "cur path: " << fs::current_path() << std::endl;

    std::string folder1 = "playground/folder1";
    fs::create_directories(folder1);

    fs::path symlinkPath = fs::current_path() /= "sym_folder1";
    fs::create_symlink(folder1, symlinkPath);

    assert(fs::is_directory(folder1));
    assert(fs::exists(symlinkPath));
    assert(fs::is_symlink(symlinkPath));
```