Лекция 8. Perfect-forwarding. Контейнеры.

Илья Макаров

ИТМО JB

26 октября 2021

Санкт-Петербург

Преобразование ссылок в шаблонах

"Склейка" ссылок:

- `T& &` $\rightarrow$ `T&`
- `T& &&` $\rightarrow$ `T&`
- `T&& &` $\rightarrow$ `T&`
- `T&& &&` $\rightarrow$ `T&&`

Универсальная или пересылающая ссылка

```
template<typename T>
void foo(T && t) {}
```

- Если вызвать `foo` от lvalue типа A, то `T = A&`.
- Если вызвать `foo` от rvalue типа A, то `T = A`.

Как работает `std::move`?

Определение `std::move`:

```
template<class T>
typename remove_reference<T>::type&&
   move(T&& a)
{
   typedef typename remove_reference<T>::type&& RvalRef;
   return static_cast<RvalRef>(a);
}
```

Важно

`std::move` не выполняет никаких действий времени выполнения.

`std::move` для lvalue

Вызываем `std::move` для lvalue объекта.

```
X x;
x = std::move(x);
```

Тип T выводится как X&.

```
typename remove_reference<X&>::type&&
    move(X& && a)
{
    using RvalRef = typename remove_reference<X&>::type&&;
    return static_cast<RvalRef>(a);
}
```

После склейки ссылок получаем:

```
X&& move(X& a)
{
    return static_cast<X&&>(a);
}
```

`std::move` для rvalue

Вызываем `std::move` для временного объекта.

```
X x = std::move(X());
```

Тип T выводится как X.

```
typename remove_reference<X>::type&&
    move(X&& a)
{
    using RvalRef = typename remove_reference<X>::type&&;
    return static_cast<RvalRef>(a);
}
```

После склейки ссылок получаем:

```
X&& move(X&& a)
{
    return static_cast<X&&>(a);
}
```

Perfect forwarding

```
// для lvalue
template<typename T, typename Arg>
auto make_unique(Arg & arg) {
    return unique_ptr<T>(new T(arg));
}

// для rvalue
template<typename T, typename Arg>
auto make_unique(Arg && arg) {
    return unique_ptr<T>(new T(std::move(arg)));
}
```

`std::forward` позволяет записать это одной функцией.

```
template<typename T, typename Arg>
auto make_unique(Arg&& arg) {
    return unique_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

Как работает std::forward?

Определение std::forward:

```
template<class S>
S&& forward(typename remove_reference<S>::type& a)
{
    return static_cast<S&&>(a);
}
```

Важно

std::forward не выполняет никаких действий времени выполнения.

std::forward для lvalue

```
X x;
auto p = make_unique<A>(x);      // Arg = X&
```

```
unique_ptr<A> make_unique(X& && arg) {
  return unique_ptr<A>(new A(std::forward<X&>(arg)));
}

X& && forward(remove_reference<X&>::type& a) {
  return static_cast<X& &&>(a);
}
```

std::forward для lvalue

После склейки ссылок:

```
unique_ptr<A> make_unique(X& arg) {
  return unique_ptr<A>(new A(std::forward<X&>(arg)));
}

X& forward(X& a) {
  return static_cast<X&>(a);
}
```

std::forward для rvalue

```
auto p = make_unique<A>(X());    // Arg = X
```

```
unique_ptr<A> make_unique(X&& arg) {
  return unique_ptr<A>(new A(std::forward<X>(arg)));
}

X&& forward(remove_reference<X>::type& a) {
  return static_cast<X&&>(a);
}
```

std::forward для rvalue

После склейки ссылок:

```
unique_ptr<A> make_unique(X&& arg) {
  return unique_ptr<A>(new A(std::forward<X>(arg)));
}

X&& forward(X& a) {
  return static_cast<X&&>(a);
}
```

Variadic templates + perfect forwarding

Можно применить `std::forward` для списка параметров.

```
template<typename T, typename ...Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(
                new T(std::forward<Args>(args)...));
}
```

Теперь `make_unique` работает для произвольного числа аргументов.

```
auto p = make_unique<Array<string>>(10, string("Hello"));
```

Общие сведения о контейнерах

Контейнеры библиотеки STL можно разделить на две категории:

- последовательные,
- ассоциативные.

Общие вложенные типы

- `Container::value_type`
- `Container::iterator, Container::const_iterator`

Общие методы контейнеров

- `size, max_size, empty, clear`.
- `begin, end, cbegin, cend`.
- Операторы сравнения: `==, !=, >, >=, <, <=`.

Шаблон `array`

Класс-обёртка над статическим массивом.

- `operator[], at`,
- `back, front`.
- `fill`,
- `data`.

Позволяет работать с массивом как с контейнером.

```
std::array<std::string, 3> a = {"One", "Two", "Three"};
std::cout << a.size() << std::endl;
std::cout << a[1] << std::endl;

// ошибка времени выполнения
std::cout << a.at(3) << std::endl;
```

Общие методы остальных последовательных контейнеров

- Конструктор от двух итераторов.
- Конструктор от `count` и `defVal`.
- Конструктор от `std::initializer_list<T>`.
- Методы `back, front`.
- Методы `push_back, emplace_back`
- Методы `assign`.
- Методы `insert`.
- Методы `emplace`.
- Методы `erase` от одного и двух итераторов.

Шаблон `vector`

Динамический массив с автоматическим изменением размера при добавлении элементов.

- `operator[]`, `at`,
- `resize`,
- `capacity`, `reserve`, `shrink_to_fit`,
- `pop_back`,
- `data`.

```
std::vector<std::string> v = {"One", "Two"};
v.reserve(100);
v.push_back("Three");
v.emplace_back("Four");
// Позволяет работать со старым кодом.
legacy_function(v.data(), v.size());
std::cout << v[2] << std::endl;
```

Шаблон deque

Контейнер с возможностью быстрой вставки и удаления элементов на обоих концах за $O(1)$. Реализован как список указателей на массивы фиксированного размера.

- `operator[]`, `at`,
- `resize`,
- `push_front`, `emplace_front`
- `pop_back`, `pop_front`,
- `shrink_to_fit`.

```
std::deque<std::string> d = {"One", "Two"};
d.emplace_back("Three");
d.emplace_front("Zero");
std::cout << d[1] << std::endl;
```

Шаблон `list`

Двусвязный список. В любом месте контейнера вставка и удаление производятся за $O(1)$.

- `push_front`, `emplace_front`,
- `pop_back`, `pop_front`,
- `splice`,
- `merge`, `remove`, `remove_if`, `reverse`, `sort`, `unique`.

```
std::list<std::string> l = {"One", "Two"};
l.emplace_back("Three");
l.emplace_front("Zero");
std::cout << l.front() << std::endl;
```

Итерация по списку

У списка нет методов для доступа к элементам по индексу.
Можно использовать range-based for:

```
using std::string;
std::list<string> l = {"One", "Two", "Three"};
for (string & s : l)
    std::cout << s << std::endl;
```

Итерация по списку

Для более сложных операций нужно использовать *итераторы*.

```
std::list<string>::iterator i = l.begin();
for ( ; i != l.end(); ++i)
    if (*i == "Two")
        break;
l.erase(i);
```

Итератор списка можно перемещать в обоих направлениях:

```
auto last = l.end();
--last; // последний элемент
```

Шаблон forward_list

Односвязный список. В любом месте контейнера вставка и удаление производятся за $O(1)$.

- `insert_after` и `emplace_after` вместо `insert` и `emplace`,
- `before_begin`, `cbefore_begin`,
- `push_front`, `emplace_front`, `pop_front`,
- `splice_after`,
- `merge`, `remove`, `remove_if`, `reverse`, `sort`, `unique`.

```
std::forward_list<std::string> fl = {"One", "Two"};
fl.emplace_front("Zero");
fl.push_front("Minus one");
std::cout << fl.front() << std::endl;
```

Шаблон `basic_string`

Контейнер для хранения символьных последовательностей.

```
typedef basic_string<char>      string;
typedef basic_string<wchar_t>   wstring;
typedef basic_string<char16_t>  u16string;
typedef basic_string<char32_t>  u32string;
```

- Метод c_str() для совместимости со старым кодом,
- поддержка неявных преобразований с C-строками,
- `operator[]`, `at`,
- `reserve`, `capacity`, `shrink_to_fit`,
- `append`, `operator+`, `operator+=`,
- `substr`, `replace`, `compare`,
- `find`, `rfind`, `find_first_of`, `find_first_not_of`, `find_last_of`, `find_last_not_of` (в терминах *индексов*)

Адаптеры и псевдоконтейнеры

Адаптеры:

- `stack` — реализация интерфейса стека.
- `queue` — реализация интерфейса очереди.
- `priority_queue` — очередь с приоритетом на куче.

Псевдо-контейнеры:

- `vector<bool>`
  - ненастоящий контейнер (не хранит `bool`-ы),
  - использует proxy-объекты.
- `bitset`
  Служит для хранения битовых масок.
  Похож на `vector<bool>`.
- `valarray`
  Шаблон служит для хранения числовых массивов и оптимизирован для достижения повышенной вычислительной производительности.

Ещё о `vector`

- Самый универсальный последовательный контейнер.
- Во многих случаях самый эффективный.
- Предпочитайте `vector` другим контейнерам.
- Интерфейс построен на итераторах, а не на индексах.
- Итераторы ведут себя как указатели.

Общие сведения

Ассоциативные контейнеры делятся на две группы:

- *упорядоченные* (требуют отношение порядка),
- *неупорядоченные* (требуют хеш-функцию).

Общие методы

1. `find` по ключу,
2. `count` по ключу,
3. `erase` по ключу.

Шаблоны `set` и `multiset`

`set` хранит упорядоченное множество (дерево поиска).
Операции добавления, удаления и поиска работают за $O(\log n)$.
Значения в `set` — неизменяемые.

- `lower_bound`, `upper_bound`, `equal_range`.

```
std::set<int> primes = {2, 3, 5, 7, 11};
// дальнейшее заполнение
if (primes.find(173) != primes.end())
    std::cout << 173 << " is prime\n";

// std::pair<iterator, bool>
auto [it, inserted] = primes.insert(3);
```

Шаблоны set и multiset

В multiset хранится упорядоченное мультимножество.

```
std::multiset<int> fib = {0, 1, 1, 2, 3, 5, 8};
// iterator
auto it = fib.insert(13);
// pair<iterator, iterator>
auto [it_begin, it_end] = fib.equal_range(1);
```

Шаблоны map и multimap

Упорядоченное отображение (дерево поиска по ключу).
Операции добавления, удаления и поиска работают за $O(\log n)$.

```
    using value_type = std::pair<const Key, T>;
```

- lower_bound, upper_bound, equal_range,
- operator[], at.

```
std::map<std::string, int> phonebook;
phonebook.emplace("Marge", 2128506);
phonebook.emplace("Lisa",  2128507);
phonebook.emplace("Bart",  2128507);
// std::map<string,int>::iterator
auto it = phonebook.find("Maggie");
if (it != phonebook.end())
    std::cout << "Maggie: " << it->second << "\n";
```

Особые методы map: operator[] и at

```
if (auto it = phonebook.find("Marge");
        it != phonebook.end())
    it->second = 5550123;
else
    phonebook.emplace("Marge", 5550123);
// или
phonebook["Marge"] = 5550123;
```

Особые методы `map`: `operator[]` и `at`

Метод `operator[]`:

1. работает только с неконстантным map,
2. требует наличие у `T` конструктора по умолчанию,
3. работает за $O(\log n)$ (не стоит использовать map как массив).

Метод `at`:

1. генерирует ошибку времени выполнения,
   если такой ключ отсутствует,
2. работает за $O(\log n)$.

Использование собственного компаратора

```
struct P { string name; string surname; };

bool operator<(P const& a, P const& b) {
    return a.name < b.name ||
           (a.name == b.name && a.surname < b.surname);
}
// уникальны по сочетанию имя + фамилия
std::set<P> s1;

struct PComp {
    bool operator()(P const& a, P const& b) const {
        return a.surname < b.surname;
    }
};
// уникальны по фамилии
std::set<P, PComp> s2;
```

Шаблоны unordered_set и unordered_multiset

unordered_set хранит множество как хеш-таблицу.
Операции добавления, удаления и поиска за $O(1)$ в среднем.
Значения в unordered_set — неизменяемые.

- equal_range, reserve,
- методы для работы с хеш-таблицей.

```
unordered_set<int> primes = {2, 3, 5, 7, 11};
// дальнейшее заполнение
if (primes.find(173) != primes.end())
    std::cout << 173 << " is prime\n";

// std::pair<iterator, bool>
auto [it, inserted] = primes.insert(3);
```

В unordered_multiset хранится мультимножество.

Шаблоны `unordered_map` и `unordered_multimap`

Хранит отображение как хеш-таблицу.
Операции добавления, удаления и поиска за $O(1)$ в среднем.

- `equal_range`, `reserve`, `operator[]`, `at`,
- методы для работы с хеш-таблицей.

```
unordered_map<std::string, int> phonebook;
phonebook.emplace("Marge", 2128506);
phonebook.emplace("Lisa",  2128507);
phonebook.emplace("Bart",  2128507);

// unordered_map<string,int>::iterator
auto it = phonebook.find("Maggie");
if (it != phonebook.end())
    std::cout << "Maggie: " << it->second << "\n";
```

Использование собственной хеш-функции

```
struct P { string name; string surname; };
bool operator==(P const& a, P const& b) {
    return a.name == b.name
        && a.surname == b.surname;
}
namespace std {
    template <> struct hash<P> {
        size_t operator()(P const& p) const {
            hash<string> h;
            return h(p.name) ^ h(p.surname);
        }
    };
}
// уникальны по сочетанию имя + фамилия
unordered_set<P> s;
```

Использование собственной хеш-функции

Общие рекомендации:

- По возможности лучше не писать свои хеш-функции.
- Для комбинирования лучше использовать специальные функции.
- Пример реализации boost::hash_combine:

```
template <typename T>
inline void hash_combine(
    size_t seed,
    const T & v)
{
    std::hash<T> hasher;
    seed ^= hasher(v) + 0x9e3779b9
            + (seed<<6) + (seed>>2);
}
```