# Инвалидация итератора

Инвалидацией итератора называется процесс, при котором итератор перестает указывать на данные, то есть итератор становится невалидным.
В разных контейнерах инвалидация происходит в разные моменты:
Для `std::list` при удалении элемента происходит инвалидация итераторов, указывающих на этот элемент.
`std::vector`: при удалении — инвалидация итератора удаляемого элемента и всех после него; при добавлении — инвалидация всех итераторов (т.к. может произойти переаллокация внутреннего буфера). Если при добавлении элемента в конец `capacity > size`, то инвалидации не происходит.
`std::deque`: при удалении первого и последнего элементов — инвалидация только соответствующих итераторов; при добавлении или удалении элементов в середине — инвалидация всех итераторов.

# Контейнер std::set

Множество — это структура данных, эквивалентная множествам в математике. Множество состоит из различных элементов заданного типа и поддерживает операции добавления элемента в множество, удаления элемента из множества, проверка принадлежности элемента множеству. Одно и то же значение хранится в множестве только один раз.

Для представления множеств в библиотеке STL имеется контейнер set, который реализован при помощи сбалансированного двоичного дерева поиска (красно-черного дерева), поэтому множества в STL хранятся в виде упорядоченной структуры, что позволяет перебирать элементы множества в порядке возрастания их значений. Для использования контейнера set нужно подключить заголовочный файл <set>.

В простейшем случае множество, например, данных типа int объявляется так:

```
std::set <int> some_set;
```

Для добавления элемента в множество используется метод `insert`:

```
some_set.insert(x);
```

Для проверки принадлежности элемента множеству используется метод `count(x)`. Этот метод возвращает количество вхождения передаваемого параметра в данный контейнер, но поскольку в множестве все элементы уникальные, то `count(x)` для типа `std::set` всегда возвращает 0 или 1. То есть для проверки принадлежности значения `x` множеству `some_set` можно использовать следующий код:

```
some_set.insert(10);
some_set.insert(10);
some_set.insert(20);
some_set.insert(30);

cout << some_set.count(10) << endl; // >> 1
```

Для удаления элемента используется метод `erase`. Ему можно передать значение элемента, итератор, указывающий на элемент или два итератора (в этом случае удаляется целый интервал элементов, содержащийся между заданными итераторами). Вот два способа удалить элемент x:

```
some_set.erase(x);
some_set.erase(some_set.find(x));
```

Метод `size()` возвращает количество элементов в множестве, метод `empty()`, возвращает логическое значение, равное true, если в множестве нет элементов, метод `clear()` удаляет все элементы из множества.

Контейнер `std::set` также имеет итератор, с помощью которого можно пройти по всем элементам коллекции. С итераторами контейнера `std::set` можно выполнять операции инкремента (что означает переход к следующему элементу) и декремента (переход к предыдущему элементу). Итераторы можно сравнивать на равенство и неравенство. Операции сравнения итераторов при помощи "<", "<=", ">", ">=" невозможны, также невозможно использовать операции прибавления к итератору числа. Разыменование итератора (применение унарного оператора `*`) возвращает значение элемента множества, на который указывает итератор. У множества есть метод `begin()`, который возвращает итератор на первый элемент множества, и метод `end()`, который возвращает фиктивный итератор на элемент, следующий за последним элементом в множестве. Таким образом, вывести все элементы множества можно так:

```
set <int> some_set;
set <int>::iterator it;
…
for (it = some_set.begin(); it != some_set.end(); ++it)
    cout << *it << " "
```

Благодаря тому, что множества хранятся в упорядоченном виде, все элементы будут выведены в порядке возрастания значений. В стандарте C++11 разрешается перебор всех элементом множества при помощи `range-based` цикла:

```
for (auto elem: S)
    cout << elem << " ";
```

Элементы также будут выведены в порядке возрастания.

Для вывода элементов в порядке убывания можно использовать `reverse_iterator`:

```
for (auto it = some_set.rbegin(); it !=
some_set.rend(); ++it)
    cout << *it << " ";
```

Функции удаления элементов могут принимать итератор в качестве параметра. В этом случае удаляется элемент, на который указывает итератор. Например, чтобы удалить наименьший элемент:

```
some_set.erase(some_set.begin());
```

Но для удаления последнего (наибольшего) элемента в `std::set` нельзя использовать `reverse_iterator`, так как `end()` указывает на следующий элемент, после последнего. Нужно взять обычный итератор, указывающий на `end()`, уменьшить и удалить:

```
auto it = some_set.begin();
--it;
some_set.erase(it);
```

# Контейнер std::map

Контейнер `std::map` предназначенный для реализации абстракции отображения в виде упорядоченного ассоциативного контейнера (ассоциативный массив). По своей сути это обертка для красно-чёрного дерева. Элементами класса `map` являются пары из ключей и соответствующих им значений. Хранение элементов класса `std::map` реализовано в упорядоченном виде на основании критерия сортировки, который применяется по значениям ключей.

```
std::map<Key, Value, Compare=std::less<Key>,
Allocator=std::allocator>
```

По умолчанию критерий сортировки задаётся оператором `operator<` `( std::less() )`, однако можно написать компаратор, который будет располагать данные, так как необходимо. Для этого пользовательский компаратор можно указать третим типом в шаблоне `std::map.`

```
struct cmp{
   bool operator()(const string &a, const string &b) const{
      return a > b;
   }
};

std::map<string, int, cmp> some_map;
```

В отличие от контейнера `std::set` класс `std::map` предоставляет пользователю `mapped_type& operator[](const key_type& k)` с помощью которого по ключу можно получить значение. Если обращение будет произведено к не существующему элементу с помощью `operator[]`, то в результате будет выделена память под элемент и будет возвращена ссылка на эту память.

```
std::map<std::string, int> some_map;
some_map["10"] = 10;
```

Асимптотическая сложность операции связана с внутренней реализацией `std::map` — красно-чёрное дерево. Вставка элемента, обновление значения ключа, удаление по ключу, поиск по ключу, происходит за `O(logN)`. Поиск по значению происходит за `O(N)`, так как необходимы итерации, пока не будет найдено необходимое значение.

Для поиска используется функция `iterator find(const Key&)`, с помощью которой можно получить итератор на значение, соответсвующее ключу. Если такого значения нет, то будет возвращено значение итератора `end()`, то есть итератор, указывающий на элемент за последним. Для вставки также можно использовать метод `emplace`, который позволяет сэкономить на копирования значений аргументов.

Для вставки может быть использована функция `pair<iterator,bool> insert (const value_type& val)`, которая принимает в качестве аргумента пару ключ-значение и возвращает пару итератор-булин. Возвращенное значение итератора указывает на место, куда был вставлен элемент, а булин показывает, был ли он вставлен.

Функция `void erase (iterator position)` позволяет удалить элемент с помощью итератора или диапазон с помощью итератора на начало и конец удаляемого фрагмента. Также есть перегрузка удаления по ключу `size_type erase (const key_type& k)`.

Ассоциативный массив позволяет выполнять различные операции над парами ключ-значение. Пары реализуются с помощью контейнера `std::pair`. Например:

```
std::map<string, int> fruitMap = {{ "apple",
105 },{ "banana", 95},{ "grapefruit", 120 },
{ "orange", 130 }};

for (auto it = fruitMap.begin(); it !=
fruitMap.end(); ++it){
std::cout << it->first << ":" << it->second <<
std::endl;
}

fruitMap.insert ( pair<string,
int>("pineapple", 150) );
```

# Контейнер std::unorderd_set

Контейнер `std::unordered_set`, доступный в заголовке `<unordered_set>` в `STL`, является ассоциативным контейнером, который содержит не сортированные уникальные ключи. Контейнер `unordered_set` поддерживает большинство тех же операций, что и set, но его модель внутреннего хранения совершенно иная. Вместо того чтобы использовать компаратор для сортировки элементов в красно черное дерево, обычно реализуется как хеш-таблица.

Хеш-функция (`hasher`) — это функция, которая принимает ключ и возвращает уникальное значение `size_t`, называемое хеш кодом. `unordered_set` организует свои элементы в хеш-таблицу, которая связывает хеш-код с коллекцией из одного или не скольких элементов, называемой сегментом. Чтобы найти элемент, `unordered_set` вычисляет свой хеш-код, а затем просматривает соответствующее поле в хеш-таблице. Пока хеш-функция работает быстро и в каждом сегменте не слишком много элементов, `unordered_set` имеет даже более высокую производительность, чем его упорядоченные аналоги: количество содержащихся элементов не увеличивает время вставки, поиска и удаления. Когда два разных ключа имеют одинаковый хеш код, это называется коллизией хеша. Если существует коллизия хеша, это означает, что два ключа будут находиться в одном и том же сегменте. Чем больше коллизий хеша, тем больше будет блоков и тем больше времени будет тратиться на поиск в блоке правильного элемента.

Хеш-функция имеет несколько требований:

1) принимает ключ и возвращает хеш-код `size_t`;
2) не генерирует исключений;
3) равные ключи дают равные хеш-коды;
4) неравные ключи дают неравные хеш-коды с высокой вероятностью. Существует небольшая вероятность коллизии хеша.

`STL` предоставляет шаблон класса хеш-функции `std::hash<T>` в заголовке `<functional>`, который содержит специализации для фундаментальных типов, типов перечислений, типов указателей, умных указателей.

```
std::hash<std::string> hasher;

auto hash_sum = hasher("42");
std::cout << hash_sum << std::endl;//>>9852803422550112 80

auto hash_sum = hasher("43");
std::cout << hash_sum << std::endl;//>>4882636553280179057
```

Для пользовательских типов можно специализировать шаблон `hash`:

```
class Foo{
    int integer_0;
    int integer_1;
    std::string string_0;
public:
    Foo(int a_integer_0, int a_integer_1, std::string a_string_0):
integer_0(a_integer_0), integer_1(a_integer_1), string_0(a_string_0){}

    friend std::hash<Foo>;
};

namespace std {
    template <>
    struct hash<Foo>{
        size_t operator()(const Foo & x) const{
            static std::hash<std::string> hasher;
            return x.integer_0 + x.integer_1 + hasher(x.string_0);
        }
    };
};
…

Foo foo(45, 0, "hello");
std::hash<Foo> hasher;
auto hash = hasher(foo);
std::cout << hash << std::endl;
```

Шаблон класса `std::unordered_set<T, Hash, KeyEqual, Allocator>` принимает четыре параметра шаблона:

1) тип ключа `T`
2) тип хеш-функции `Hash`, который по умолчанию равен `std::hash<T>`
3) тип функции равенства `KeyEqual`, который по умолчанию равен `std::equal_to<T>`
4) тип распределителя `Allocator`, по умолчанию равен `std::allocator<T>`

`std::unordered_set` поддерживает конструкторы, эквивалентные конструкторам `std::set`, с корректировками для различных параметров шаблона (для `std::set` требуется `Comparator`, тогда как для `std::unordered_set` требуются `Hash` и `KeyEqual`).

# Контейнер std::unorderd_map

`std::unorderd_map` — неупорядоченный ассоциативный контейнер, построенный на основе хеш таблицы.
`std::unorderd_map` шаблонный класс, в который ходят:

```
std::unorderd_map <
    Key,
    Value,
    Hash=std::hash<Key>,
    EqualTo=std::equal_to<Key>
>
```

Соотвественно так же как и для `std::map` существуют обязательные параметры ключ и значение. Кроме того есть два необязательных параметра — хеш-функция, по умолчанию `std::hash<Key>`, и компаратор для установления эквивалентности двух ключей `std::equal_to<Key>`.

Хеш таблица, которая используется в контейнере `std::unorderd_map` реализована методом цепочек. Для этого способа реализации характерно хранение объектов в так называемых «корзинах», которые сами по себе являются самостоятельными структурами данных например вектор или дерево. В начале имеется область памяти, массив содержащий набор «корзин». Хеш функция позволяет определить, какую в какую корзину будет отправлен объект. Хеш функция может давать определенное количество коллизий, которые не будут причиной поломки `std::unorderd_map`. В какой то момент, может случиться ситуация, когда «корзины» будут неравномерно переполняться. Для определения этого события существует параметр `load_factor`, который есть `size()/ bucket_cout()`, по сути относительная загруженность хеш таблицы. Если параметр `load_factor=0.5`, то считается, что коллекция достаточно заполнена, те поиск будет досрочно долгим. В таком случае происходит увеличение массива «корзин» (`bucket_cout`) и пересчёт (`rehash`) всей коллекции, в результате чего таблица становится более разреженной. Параметр `max_load_factor` можно менять вручную.

Соответственно поиск и удаление в лучшем случае будет происходить за `O(1)`, в худшем случае, когда хеш функция для ключей будет давать слишком много коллизий, эффективность `std::unorderd_map` будет равна эффективности внутреннего устройства «корзины». Начало «корзины» связано с концом другой «корзины», таки образом итератор `std::unorderd_map` позволяет двигаться по «корзинам», так, как будто это список. Кроме всего прочего элементы внутри `std::unorderd_map` хранятся не в виде переданного объекта, а в специальной структуре, которая содержит дополнительную информацию, например хеш ключа, для того, чтобы не пересчитывать его.

# Функторы

Функторы (или функциональные объекты) — это объект, использование которого возможно подобно вызову функции. Функциональный объект является экземпляром класса C++, в котором определён `operator()`. С одной стороны это обычные объекты, но с другой стороны, они похожи на функции, которые могут сохранять своё предыдущее состояние.

```
class SimpleFunctor {
    std::string name_;
public:
    SimpleFunctor(const char *name) : name_(name) {}
    void operator()() { std::cout << "Hello, " <<
name_ << endl; }
};

…

SimpleFunctor sf("simple functor");
sf(); // выводит "simple functor"
```

Перегрузка операторов в основном заключается в переопределении типа параметров, но не их количества. Например, оператор == всегда принимает два параметра, тогда как оператор `!` всегда принимает один параметр. Оператор `()` позволяет изменять как тип параметров, так и их количество.

```
class Matrix{
    double data[5][5];
public:
    Matrix(){
        for (int row=0; row < 5; ++row)
            for (int col=0; col < 5; ++col)
                data[row][col] = 0.0;
    }
    double& operator()(int row, int col);
    const double& operator()(int row, int col) const; // для константных объектов
    void operator()();
};

double& Matrix::operator()(int row, int col){
    assert(col >= 0 && col < 5);
    assert(row >= 0 && row < 5);
    return data[row][col];
}

const double& Matrix::operator()(int row, int col) const{
    assert(col >= 0 && col < 5);
    assert(row >= 0 && row < 5);
    return data[row][col];
}

void Matrix::operator()(){
    for (int row=0; row < 5; ++row)
        for (int col=0; col < 5; ++col)
            data[row][col] = 0.0;
}
…

Matrix matrix;
matrix(2, 3) = 3.6;
std::cout << matrix(2, 3);
```

Чаще всего функторы в C++ используются в качестве предикатов, функций в алгоритмах `STL`. Например, алгоритм `for_each` библиотеки `STL` выполняет действие над элементами контейнера, но для этого ему необходимо передать три параметра: итераторы, настроенные на первый и последний элементы над которыми будут выполняться действие, и указатель на функцию или функтор, которые реализуют само это действие. Например:

```
class EvenOddFunctor {
    int even_;
    int odd_;
public:
    EvenOddFunctor() : even_(0), odd_(0) {}
    void operator()(int x) {
        if (x%2 == 0) even_ += x;
        else odd_ += x;
    }
    int even_sum() const { return even_; }
    int odd_sum() const { return odd_; }
};

EvenOddFunctor evenodd;
std::vector<int> my_list = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
evenodd = std::for_each(my_list.begin(),
                        my_list.end(),
                        evenodd);
```

# Лямбда-выражения

В языке C++ есть ещё один способ реализации действия над элементами контейнера – это лямбда выражения. Лямбда-выражение это фактически краткая запись неименнованного функтора. Лямбда-выражения записываются с использованием нового синтаксиса, однако в момент компиляции они превращаются в реальный объект класса.

Синтаксис лямбда-выражения имеют вид:

[захват] (параметры) -> возвращаемый_тип
{
    инструкции;
}

Захват и параметры могут быть пустыми, если они не нужны. Тип возвращаемого значения является необязательным, и если он опущен, будет использоваться значение `auto` то есть использование вывода типа для определения типа возвращаемого значения.

```
[]() {}; // лямбда-выражение без захватов, без параметров и без возвращаемого типа
…
auto myLambda = [](){
  cout << "Hello World!" << endl;
};
myLambda();
```

Программный код не скомпилируется, если в лямда-функции будет присутствовать два и более оператора return, которые будут возвращать объекты разных типов, не связанных между собой иерархией наследования и не способные быть приведены к типу базового класса. И даже, если эти объекты имеют базовый класс, необходимо будет прописать тип возвращаемого значения, им как раз будет указатель на объект базового класса (в общем случае).

```
struct A
{
    A(){}
};
struct B : public A
{
    B(){}
};
struct C : public A
{
    C(){}
};

…

auto Lambda = [](int type) -> A*{
    if (type == 0){
        return new B();
    } else {
        return new C();
    }
};
Lambda(0);
```

Замыкание (closure) представляет собой объект времени выполнения, создаваемый лямбда-выражением. В зависимости от режима захвата замыкания хранят копии ссылок на захваченные данные. В приведенном ниже вызове `std::find_if` замыкание представляет собой объект, передаваемый во время выполнения в `std::find_if` в качестве третьего аргумента.
Класс замыкания (closure class) представляет собой класс, из которого инстанцируется замыкание. Каждое лямбда-выражение заставляет компиляторы генерировать уникальный класс замыкания. Инструкции внутри лямбда-выражения становятся выполнимыми инструкциями функций-членов их класса замыкания.

```
std::vector<int> vec = {5, 10, 20, 30};
auto res = std::find_if(vec.begin(), vec.end(),
[]( int val ) { return 0 < val && val < 10; });
```

В отличие от вложенных блоков кода, где любой идентификатор, определенный во внешнем блоке, доступен и во внутреннем, лямбды в языке C++ могут получить доступ только к определенным видам идентификаторов: глобальные идентификаторы, объекты, известные во время компиляции и со статической продолжительностью жизни.

```
array<string, 4> arr{ "apple", "banana", "walnut", "lemon" };
string search = "nut";
auto found = find_if(arr.begin(), arr.end(), [](string str) {
    return (str.find(search) != string::npos); // ошибка
});
```

Данный код не скомпилируется, так как `search` не соответствует ни одному из этих требований, поэтому лямбда не может её увидеть.

Поле capture clause используется для того, чтобы предоставить лямбда-выражению доступ к переменным из окружающей области видимости, к которым она обычно не имеет доступ. Всё, что нужно для этого сделать это перечислить в поле capture clause объекты, к которым необходимо получить доступ внутри лямбда-выражения.

```
array<string, 4> arr{ "apple", "banana", "walnut", "lemon" };
string search = "nut";
auto found = find_if(arr.begin(), arr.end(), [search](string
str) {
    return (str.find(search) != string::npos); // ошибка
});
```

Переменные, захваченные лямбда-выражениями, являются клонами переменных из внешней области видимости, а не фактическими «внешними» переменными. Таким образом, в примере, приведенном выше, при создании объекта лямбда-выражения, она получает свою собственную переменную-клон с именем `search`. По умолчанию переменные захватываются как константные значения, значит `search` не получится изменить внутри лямбда-выражения. Чтобы разрешить изменения значения переменных, которые были захвачены по значению, можно пометить лямбду как `mutable`, в данном контексте, ключевое слово `mutable` удаляет спецификатор `const` со всех переменных, захваченных по значению.

```
int ammo = 10;
auto shoot{
    [ammo]() mutable {
      --ammo;
      std::cout << "Shoot! " << ammo << " shot(s) left.\n";
    }
};
shoot();
cout << "Shoot! " << ammo << endl; // >>10
```

При вызове лямбда захватила копию переменной `ammo`. Затем, когда лямбда уменьшает значение переменной `ammo` с `10` до `9` и до `8`, то, на самом деле, она уменьшает значение копии, а не исходной переменной. Значение переменной `ammo` сохраняется, несмотря на вызовы лямбда-выражения.

Подобно тому, как функции могут изменять значения аргументов, передаваемых им по ссылке, также возможно захватывать переменные по ссылке, чтобы позволить лямбда-выражению влиять на значения аргументов. Чтобы захватить переменную по ссылке, необходимо добавить знак амперсанда к имени переменной, которую желательно захватить. В отличие от переменных, которые захватываются по значению, переменные, которые захватываются по ссылке, не являются константными (если только переменная, которую они захватывают, не является изначально `const`).

```
int ammo = 10;

auto shoot =
    [&ammo]() {
        --ammo;
        cout << "Shoot! " << ammo << endl;
};

shoot();
cout << "Shoot! " << ammo << endl; // >> 9
```

Возможно захватить несколько переменных. Для этого в блоке захвата необходимо разделить их запятыми. Можно выполнить как несколько захватов по значению, так и несколько захватов по ссылке.

```
int health{ 33 };
int armor{ 100 };
std::vector<CEnemy> enemies{};

[health, armor, &enemies](){};
```

Необходимость явно перечислять переменные для захвата иногда может быть несколько обременительной. Есть возможность с помощью компилятора автоматически сгенерировать списки переменных, которые нужно захватить. Чтобы захватить все задействованные переменные по значению, необходимо использовать `=` в качестве значения для захвата, чтобы захватить все задействованные переменные по ссылке, необходимо использовать `&` в качестве значения для захвата.

```
int ammo = 10;
int damage = 0;

auto shoot{
    [=]() mutable {  // или &
        —ammo;
        ++damage;
        std::cout << "Shoot! " << ammo << " shot(s) left.\n";
    }
};

shoot();
shoot();

std::cout << "Shoot " << ammo << std::endl;
std::cout << "Damage " << damage << std::endl;
```

Захваты по умолчанию могут быть смешаны с обычными захватами. Вполне допускается захватить некоторые переменные по значению, а другие — по ссылке, но при этом каждая переменная может быть захвачена только один раз.

```
int health{ 33 };
int armor{ 100 };
std::vector<CEnemy> enemies{};

// Захват переменных health и armor по значению, а enemies – по ссылке
[health, armor, &enemies](){};

// Захват enemies по ссылке, а все остальные – по значению
[=, &enemies](){};

// Захват переменных armor по значению, а все остальные – по ссылке
[&, armor](){};

// Запрещено, так как уже определен захват по ссылке для всех переменных
[&, &armor](){};

// Запрещено, так как уже определен захват по значению для всех переменных
[=, armor](){};

// Запрещено, так как переменная armor используется дважды
[armor, &health, &armor](){};

// Запрещено, так как захват по умолчанию должен быть первым элементом в списке захвата
[armor, &](){};
```

Иногда нужно захватить переменную с небольшой модификацией или объявить новую переменную, которая видна только в области видимости лямбда-функции. Это возможно сделать, определив переменную в лямбда-захвате без указания её типа.

```
std::array areas{ 100, 25, 121, 40, 56 };

int width = 5;
int height = 5;

auto found{ std::find_if(areas.begin(), areas.end(), [userArea{
width * height }](int knownArea) {
    return (userArea == knownArea); })
};
```

Переменная userArea будет рассчитана только один раз во время определения лямбда-выражения. Вычисляемая площадь хранится в объекте лямбда-выражения и одинакова для каждого вызова. Если лямбда-выражения имеет модификатор mutable и изменяет переменную, которая определена в захвате, то исходное значение переменной будет переопределено.

Переменные захватываются в точке определения лямбды. Если переменная, захваченная по ссылке, прекращает свое существование до прекращения существования лямбды, то лямбда остается с висячей ссылкой.

```
auto makeLambda(const string& name)
{
  // Захват переменной name по ссылке и возврат лямбды
   return [&]() {
      cout << "Name is " << name << endl; //
неопределенное поведение
   };
}

…

auto sayName{ makeLambda("Lambda") };
sayName();
```

В качестве возвращаемого типа функции `makeLambda` используется ключевое слово `auto`, которое раскрывается в тип объектных ссылок на функции `std::function<void()>`. Вызов функции `makeLambda()` создает временный объект `std::string` из строкового литерала `"Lambda"`. Лямбда-выражение в функции `makeLambda()` захватывает временную строку по ссылке. Данная строка уничтожается при выполнении возврата `makeLambda()`, но при этом лямбда-выражение все еще ссылается на нее. Когда вызывается `sayName()`, происходит попытка доступа к висячей ссылке, что приводит неопределенному поведению программы. Следует помнить, что захваченные переменные должны существовать дольше, чем само лямбда-выражение.

Лямбда-выражения являются объектами, соответсвенно их можно копировать. В некоторых случаях это может вызвать проблемы.

```
int i{ 0 };

auto count{ [i]() mutable {
    std::cout << ++i << '\n';
}};

count(); // >> 1
auto otherCount{ count }; // создание копии count

count(); // >> 2
otherCount(); // >> 2
```

При копировании объекта count, так же копируется его текущее состояние.

Пример демонстрирует возникновение той же проблемы. Когда с помощью лямбда-выражения создается объект `std::function`, то он внутри себя создает копию лямбда-выражения. Таким образом, вызов `fn()` фактически выполняется при использовании копии лямбда-выражения.

```
void invoke(const std::function<void(void)>& fn)
{ fn(); }

…
```

```
int i{ 0 };

auto count{ [i]() mutable {
    std::cout << ++i << '\n';
}};

invoke(count); // >>1
invoke(count); // >>1
invoke(count); // >>1
```

Если необходимо передать изменяемое лямбда-выражение и при этом избежать непреднамеренного копирования, то есть два варианта решения данной проблемы. Один из них — использовать вместо этого лямбда-выражение, не содержащую захватов. Копирования не будет если не использовать захват и отслеживать состояние, используя статическую локальную переменную. Но статические локальные переменные могут быть трудны для отслеживания и делают код менее читабельным. Второй вариант — это с самого начала не допустить возможности копирования лямбда-выражения. В C++ предоставляет тип `std::ref`, который позволяет передавать обычный тип, как если бы это была ссылка. Обёртывая лямбда-выражение в `std::ref` позволяет получить копирование не объекта, а ссылки на объект.

```
invoke(std::ref(count)); // >>1
invoke(std::ref(count)); // >>2
invoke(std::ref(count)); // >>3
```

# std::function

Класс `std::function` является высокоуровневой оберткой над функциями и функциональными объектами. Объекты класса `std::function` могут хранить, копировать и вызывать произвольные вызываемые объекты — функции, лямбда-выражения, выражения связывания и другие функциональные объекты.. `std::function` входит в стандарт языка C++11, для его использования необходимо добавить заголовочный файл `<functional>`.

```
struct A {
    A(int num) : num_(num){}
    void printNumberLetter(char c) const {std::cout << "Number: " << num_  << " Letter:
" << c << std::endl;}
    int num_;
};

void printLetter(char c) { std::cout << c << std::endl; }

struct B { void operator() () {std::cout << "B()" << std::endl;} };

int main()
{
    // Содержит функцию
    std::function<void(char)> f_print_Letter = printLetter;
    f_print_Letter('Q');

    // Содержит лямбда-выражение
    std::function<void()> f_print_Hello = [] () {std::cout << "Hello world!" <<
std::endl;};
    f_print_Hello();

    // Содержит связыватель
    std::function<void()> f_print_Z = std::bind(printLetter, 'Z');
    f_print_Z();

    // Содержит вызов метода класса
    std::function<void(const A&, char)> f_printA = &A::printNumberLetter;
    A a(10);
    f_printA(a, 'A');

    // Содержит функциональный объект
    B b;
    std::function<void()> f_B = b;
    f_B();
}
```