

# Лекция 5. Основные коллекции данных и обработка ошибок

МИФИ, 2016

Роман Кузнецов

**Просьба отметить на  
портале!**

## План лекции

1. Основные коллекции данных в STL. Правильный выбор коллекции.
2. Основные алгоритмы над коллекциями данных в STL.
3. Обработка ошибок на этапе компиляции и в runtime.
4. Исключения в C++.

# Standard Template Library (STL)

STL базируется на следующих принципах:

- многократное использование и эффективность кода;
- модульность;
- расширяемость;
- удобство применения;
- взаимозаменяемость компонентов;
- унификация интерфейсов;
- гарантии вычислительной сложности операций.

С технической точки зрения, STL представляет собой набор шаблонов классов и алгоритмов (функций), предназначенных для совместного использования при решении широкого спектра задач.

## Состав STL

- **обобщенные контейнеры** (универсальные структуры данных) — векторы, списки, множества и т.д.;
- **обобщенные алгоритмы** решения типовых задач поиска, сортировки, вставки, удаления данных и т.д.;
- **итераторы** (абстрактные методы доступа к данным), являющиеся обобщением указателей и реализующие операции доступа алгоритмов к контейнерам;
- **функциональные объекты**, обобщающие понятие функции;
- **адаптеры**, модифицирующие интерфейсы контейнеров, итераторов, функций;
- **распределители** (аллокаторы) памяти.

## Big-O нотация

Оценка эффективности операции над контейнером, выраженная во времени выполнения этой операции, в зависимости от кол-ва элементов внутри этого контейнера (N).

$$T(N) = O(f(N))$$

Основные варианты:

- константное время выполнения:
- линейное время выполнения:
- квадратичное время выполнения:
- логарифмическое время выполнения:
- время выполнения «N логарифмов N»:

$$T(N) = O(1)$$

$$T(N) = O(N)$$

$$T(N) = O(N^2)$$

$$T(N) = O(\log N)$$

$$T(N) = O(N \log N)$$

# Основные контейнеры

## Последовательные контейнеры:

- массив (array);
- вектор (vector);
- дек (deque);
- список (list).

## Упорядоченные ассоциативные контейнеры:

- (мульти)множество (set, multiset);
- (мульти)отображение (map, multimap).

## Неупорядоченные ассоциативные контейнеры:

- неупорядоченное (мульти)множество (unordered\_set, unordered\_multiset);
- неупорядоченное (мульти)отображение (unordered\_map, unordered\_multimap).

## Адаптеры контейнеров:

- стек (stack);
- очередь (queue);
- очередь с приоритетом (priority\_queue).

## Сложность операций над последовательными контейнерами

Вид операции	Вектор	Дек	Список
Доступ к $i$ -му элементу	$O(1)$	-	$O(N)$
Добавление/удаление в начале	$O(N)$	Амортизированное $O(1)$	$O(1)$
Добавление/удаление в середине	$O(N)$	-	$O(1)$
Добавление/удаление в конце	Амортизированное $O(1)$	Амортизированное $O(1)$	$O(1)$
Поиск	$O(N)$	$O(N)$	$O(N)$



## Подключение контейнерных классов

Имена заголовочных файлов почти всегда совпадают с именами коллекции.

vector → **#include** <vector>

list → **#include** <list>

map → **#include** <map>

Подробнее смотрите в документации к STL.

## Вектор

Вектор — последовательный контейнер

- переменной длины;
- с произвольным доступом к элементам;
- с быстрой вставкой и удалением элементов в конце контейнера.

Технически вектор STL реализован как шаблон с параметрами вида:

```
template <typename T, // тип данных  
          typename Allocator = allocator<T>>  
vector;
```

## Конструирование вектора

```
vector<T> vector1;           // за время O(1)
vector<T> vector2(N, value); // за время O(N) с вызовом T::T(T&)
vector<T> vector3(N);        // за время O(N) с вызовом T::T()
vector<T> vector4(vector3);   // за время O(N)
vector<T> vector5(first, last); // за время O(N)
// за время O(N) с помощью initializer_list<T>
vector<T> vector6 { T(1), T(2), T(3) };
```

## Вектор: пример использования

```
std::vector<int> v = { 7, 5, 16, 8 };  
v.push_back(25);  
v.push_back(13);
```

```
std::vector<Bullet> bullets;  
bullets.reserve(3);  
bullets.emplace_back(Bullet(55));  
bullets.push_back(Bullet(15));  
bullets.push_front(Bullet(10));
```

```
auto it = bullets.begin();  
bullets.erase(it);
```

```
auto itEnd = bullets.end();  
bullets.erase(itEnd); // Ошибка!
```

```
bullets.clear();
```

## Дек

Дек — последовательный контейнер

- переменной длины;
- с произвольным доступом к элементам;
- с быстрой вставкой и удалением элементов в начале и конце контейнера;
- без гарантии сохранения корректности итераторов после вставки и удаления.

Технически дек реализован как шаблон с параметрами вида:

```
template <typename T, // тип данных  
         typename Allocator = allocator<T>>  
deque;
```

**Примечание:** конструирование и многие функции аналогичны `vector<T>`.

## Дек: пример использования

```
class Message {};  
void foo()  
{  
    std::deque<Message> messageQueue;  
    messageQueue.push_back(Message(MessageType::GameStarted));  
    messageQueue.push_back(Message(MessageType::GamePaused));  
    messageQueue.push_back(Message(MessageType::GameFinished));  
    while (!messageQueue.empty())  
    {  
        auto const & message = messageQueue.front();  
        // process message...  
        messageQueue.pop_front();  
    }  
}
```

## Список

Список — последовательный контейнер

- переменной длины;
- с двунаправленными итераторами для доступа к элементам;
- с быстрой вставкой и удалением элементов в любой позиции;
- со строгой гарантией сохранения корректности итераторов после вставки и удаления.

Технически список реализован как шаблон с параметрами вида:

```
template <typename T, // тип данных  
         typename Allocator = allocator<T>>
```

```
list;
```

**Примечание:** конструирование аналогично `vector<T>`.

## Список: пример использования

```
class Alien {};  
  
int main()  
{  
    std::list<Alien> aliens;  
    aliens.push_back(Alien());  
    aliens.insert(aliens.begin(), Alien());  
  
    for (auto it = aliens.begin(); it != aliens.end(); ++it)  
        std::cout << *it << std::endl;  
  
    return 0;  
}
```



## Упорядоченные ассоциативные контейнеры

**Множество** — контейнер типа `set<T>` с поддержкой уникальности ключей и быстрым доступом к ним.

**Отображение** — контейнер типа `map<Key, T>` с поддержкой уникальных ключей типа `Key` и быстрым доступом по ключам к значениям типа `T`.

**Мультимножество** — аналогичный множеству контейнер типа `multiset<T>` с возможностью размещения в нем неуникальных ключей.

**Мультиотображение** — аналогичный отображению контейнер типа `multimap<Key, T>` с возможностью размещения в нем неуникальных ключей.

## Множество и мультимножество

Множество, мультимножество — упорядоченный ассоциативный контейнер

- переменной длины;
- с двунаправленными итераторами для доступа к элементам;
- с логарифмическим временем доступа.

Технически множества и мультимножества реализованы как шаблоны с параметрами вида:

```
template <typename Key, // тип ключа
          typename Compare = less<Key>, // ф-ия сравнения
          typename Allocator = allocator<Key>>
set; // или multiset
```

## Конструирование множества

```
set(Compare const & comp = Compare());
```

```
template <typename InputIterator>  
set(InputIterator first, InputIterator last, Compare const & comp =  
Compare());
```

```
set(set<Key, Compare, Allocator> const & rhs);
```

## Множество: пример использования

```
std::set<GameEntity *> damagedEntities;
```

```
void DamageEntity(GameEntity * entity)
{
    // ...
    damagedEntities.insert(entity);
}
```

```
void ProcessDamage()
{
    for (GameEntity * entity : damagedEntities)
    {
        // Process damage ...
    }
}
```

## Отображения и мультиотображения

Отображение, мультиотображение — упорядоченный ассоциативный контейнер

- переменной длины;
- с двунаправленными итераторами для доступа к элементам;
- с логарифмическим временем доступа.

Технически отображения и мультиотображения реализованы как шаблоны с параметрами вида:

```
template <typename Key, // тип ключа
          typename T,    // тип значения
          typename Compare = less<Key>, // ф-ия сравнения
          typename Allocator = allocator<pair<Key const, Value>>>
map; // или multimap
```

## Отображение: пример использования

```
class Texture
{
public:
    Texture(std::string const & name) {}
};

int main()
{
    std::map<std::string, std::shared_ptr<Texture> > textures;
    textures.insert(std::make_pair("gun", std::make_shared<Texture>("gun.png")));
    textures.insert(std::make_pair("alien", std::make_shared<Texture>("alien.png")));

    auto it = textures.find("gun");
    if (it != textures.end())
    {
        std::shared_ptr<Texture> texture = it->second;
        // ...
    }
    return 0;
}
```

## Неупорядоченные ассоциативные контейнеры

Неупорядоченное множество — `unordered_set<T>`.

Неупорядоченное отображение — `unordered_map<Key, T>`.

Неупорядоченное мультимножество — `unordered_multiset<T>`.

Неупорядоченное мультиотображение — `unordered_multimap<Key, T>`.

Структура и функции схожи с упорядоченными ассоциативными контейнерами.

Реализуется при помощи хэш-таблицы.

# Неупорядоченные ассоциативные контейнеры: пример

```
enum FactoryType { AlienFactoryType = 0, BulletFactoryType = 1 };

class Factory {};
class AlienFactory : public Factory {};
class BulletFactory : public Factory {};

int main()
{
    std::unordered_map<int, std::unique_ptr<Factory> > factories;
    factories[AlienFactoryType] = std::unique_ptr<Factory>(new AlienFactory());
    factories[BulletFactoryType] = std::unique_ptr<Factory>(new BulletFactory());

    // ...
    Factory * factory = factories[BulletFactoryType].get();
    // ...

    return 0;
}
```



## Итераторы

Итераторы (обобщенные указатели) — объекты, предназначенные для обхода последовательности объектов в обобщенном контейнере.

Типы итераторов:

- входные;
- выходные;
- однонаправленные;
- двунаправленные;
- произвольного доступа.

## Итераторы: разрешенные операции

*i (чтение)	== !=	++i	*i (запись)	--i	+ - < >
Входные			Запрещено		
Запрещено		Выходные		Запрещено	
Однонаправленные				Запрещено	
Двунаправленные					Запрещено
Произвольного доступа					

Обход контейнера итератором осуществляется в пределах диапазона, определяемого парой итераторов (first и last). При этом итератор last никогда не разыменовывается: [first; last).

## Основные итераторы в STL

Шаблоны классов контейнеров содержат определения следующих типов итераторов:

- изменяемый итератор прямого обхода (допускает преобразование к константному итератору; *\*i* — ссылка):

`Container<T>::iterator`

- константный итератор прямого обхода (*\*i* — константная ссылка):

`Container<T>::const_iterator`

- изменяемый итератор обратного обхода:

`Container<T>::reverse_iterator`

- константный итератор обратного обхода:

`Container<T>::const_reverse_iterator`

## Основные итераторы: пример

```
void foo(std::list<GameEntity *> const & entities)
{
    for (std::list<GameEntity *>::iterator it = entities.begin(); it != entities.end(); ++it)
    {
        it->DoSomething();
    }

    for (auto it = entities.begin(); it != entities.end(); ++it)
    {
        it->DoSomethingElse();
    }

    for (auto it = entities.rbegin(); it != entities.rend(); ++it)
    {
        it->DoSomethingElseAgain();
    }
}
```

## Использование using

```
using TEntityList = std::list<GameEntity *>;
```

```
using TEntityListConstIt = std::list<GameEntity *>::const_iterator;
```

```
void foo(TEntityList const & entities)
{
    for (TEntityListConstIt it = entities.begin(); it != entities.end(); ++it)
    {
        it->DoSomething();
    }
}
```

## Итераторы вставки

Итераторы вставки работают совместно с обобщенными алгоритмами, разыменованное итератора `*i` влечет за собой добавление элемента при помощи одного из предоставляемых контейнером методов вставки.

В STL итераторы вставки являются шаблонами классов, параметром которых является контейнерный тип `Container`:

`back_insert_iterator<Container>` — использует метод класса `Container::push_back`;

`front_insert_iterator<Container>` — использует метод класса `Container::push_front`;

`insert_iterator<Container>` — использует метод класса `Container::insert`.

## Итераторы вставки: пример

```
int main()
{
    std::list<int> l;
    std::vector<int> v = { 3, 6, 7, 2 };
    std::copy(v.begin(), v.end(), std::back_inserter<std::list<int>>(l));
    std::copy(v.begin(), v.end(), back_inserter(l));

    return 0;
}
```

## Обобщенные алгоритмы

Обобщенные алгоритмы STL предназначены для эффективной обработки обобщенных контейнеров.

Делятся на четыре основных группы:

- немодифицирующие последовательные алгоритмы;
- модифицирующие последовательные алгоритмы;
- алгоритмы упорядочения;
- алгоритмы на числах.

`#include <algorithm>`



## Немодифицирующие последовательные алгоритмы

Не изменяют содержимое контейнера-параметра и решают задачи поиска перебором, подсчета элементов и установления равенства двух контейнеров.

Например: `find()`, `equal()`, `count()`.

# Немодифицирующие последовательные алгоритмы: пример

```
int main()
{
    std::vector<int> v { 0, 1, 1, 2, 3, 4 };

    auto it = std::find(v.begin(), v.end(), 3);
    if (it != v.end())
    {
        // Do something...
    }

    size_t cnt = std::count(v.begin(), v.end(), 1);

    return 0;
}
```

## Модифицирующие последовательные алгоритмы

Изменяют содержимое контейнера-параметра, решая задачи копирования, замены, удаления, размешивания, перестановки значений и пр.

Например: `copy()`, `random_shuffle()`, `replace()`.

## Модифицирующие последовательные алгоритмы: пример

```
int main()
{
    std::array<int, 10> s { 5, 7, 4, 2, 8, 6, 1, 9, 0, 3 };
    std::replace(s.begin(), s.end(), 8, 88);

    return 0;
}
```

## Алгоритмы упорядочения

алгоритмы STL, работа которых опирается на наличие или установление отношения порядка на элементах. К данной категории относятся алгоритмы сортировки и слияния последовательностей, бинарного поиска, а также теоретико-множественные операции на упорядоченных структурах.

Например: `sort()`, `binary_search()`, `set_union()`.

## Алгоритмы упорядочения: пример

```
int main()
{
    std::array<int, 10> s = { 5, 7, 4, 2, 8, 6, 1, 9, 0, 3 };

    // sort using the default operator <
    std::sort(s.begin(), s.end());

    // sort using a standard library compare function object
    std::sort(s.begin(), s.end(), std::greater<int>());

    // sort using a custom function object
    struct
    {
        bool operator()(int a, int b) { return a < b; }
    } customLess;
    std::sort(s.begin(), s.end(), customLess);

    return 0;
}
```

## Алгоритмы на числах

Алгоритмы обобщенного накопления, вычисления нарастающего итога, попарных разностей и скалярных произведений.

Например: `accumulate()`, `partial_sum()`, `inner_product()`.

## Алгоритмы на числах: пример

```
#include <numeric>
```

```
int main()
```

```
{
```

```
    std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
    int sum = std::accumulate(v.begin(), v.end(), 0);
```

```
    int product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>());
```

```
    return 0;
```

```
}
```



## Еще пара полезных алгоритмов

- Удаление элементов `std::remove`

```
void foo()
{
    std::string str = "Текст с несколькими пробелами";
    str.erase(std::remove(str.begin(), str.end(), ' '), str.end());
}
```

## Еще пара полезных алгоритмов

- Получение списка уникальных значений `std::unique`

```
void foo()
{
    // remove duplicate elements
    std::vector<int> v { 1, 2, 3, 1, 2, 3, 3, 4, 5, 4, 5, 6, 7 };
    std::sort(v.begin(), v.end()); // 1 1 2 2 3 3 3 4 4 5 5 6 7
    auto last = std::unique(v.begin(), v.end());
    // v now holds {1 2 3 4 5 6 7 x x x x x x}, where 'x' is indeterminate
    v.erase(last, v.end());
}
```

## Способы обработки ошибок

1. Игнорировать;
2. Выводить сообщение;
3. Возвращать код ошибки (возвращать `bool` - есть ошибка или нет ошибки);
4. `assert` и `static_assert`;
5. `try/catch`.

## Вывод сообщений об ошибках

```
int GetSegmentsCount(float fullLength, float segmentLength)
{
    if (fabs(segmentLength) < 1e-5)
    {
        std::cout << "Segment length is zero" << std::endl;
        return 0;
    }
    return static_cast<int>(fullLength / segmentLength);
}
```

## Использование кодов ошибок

```
enum class ErrorCode
```

```
{  
    Ok,  
    ZeroLength  
};
```

```
ErrorCode GetSegmentsCount(float fullLength, float segmentLength, int & result)
```

```
{  
    if (fabs(segmentLength) < 1e-5)  
        return ErrorCode::ZeroLength;  
    result = static_cast<int>(fullLength / segmentLength);  
    return ErrorCode::Ok;  
}
```

## Использование assert

**Assertion checking** — проверка на условия, которые никогда не должно произойти в программе. Однако, если такие условия наступают, программа аварийно завершается. Как правило, такие проверки работают только в DEBUG-режиме.

```
#include <cassert>
```

```
int GetSegmentsCount(float fullLength, float segmentLength)
{
    assert(fabs(segmentLength) >= 1e-5 && "Segment length is zero");
    return static_cast<int>(fullLength / segmentLength);
}
```

## Использование `static_assert`

Проверка корректности написанного кода на этапе компиляции. Возможно только для тех условий, которые компилятор может проверить на этапе компиляции.

```
template <typename T, int Size>
class Tuple
{
    static_assert(Size > 1, "Tuple must contain at least 2 elements");
    T m_elements[Size];
};
```

```
Tuple<int, 1> tuple; // Ошибка!
Tuple<int, 2> tuple2; // Нет ошибок!
```

## constexpr

Константное выражение, которое может быть вычислено на этапе компиляции.

```
constexpr int GetTupleSize(int baseSize)
{
    return baseSize + 1;
}
```

```
template <typename T, int Size>
class Tuple
{
    static_assert(Size > 1, "Tuple must contain at least 2 elements");
    T m_elements[Size];
};
```

```
Tuple<int, GetTupleSize(0)> tuple; // Ошибка!
Tuple<int, GetTupleSize(1)> tuple2; // Нет ошибок!
```



# Мощь constexpr


```
#include <iostream>
```

```
template<int Index>
constexpr unsigned long long Factorial()
{
    static_assert(Index > 0, "Factorial for negative numbers does not exist");
    return Index * Factorial<Index - 1>();
}
```

```
template<>
constexpr unsigned long long Factorial<0>()
{
    return 1;
}
```

```
int main()
{
    constexpr unsigned long long v = Factorial<66>();
    std::cout << v << std::endl;
    return 0;
}
```

Возможны даже рекурсивные функции, которые будут выполнены на этапе компиляции.



## Мощь constexpr

```
#include <iostream>
```

```
constexpr unsigned long long Factorial(unsigned int const n)
{
    return n == 0 ? 1 : n * Factorial(n - 1);
}
```

Тернарный `if` можно использовать  
в `constexpr` функциях!

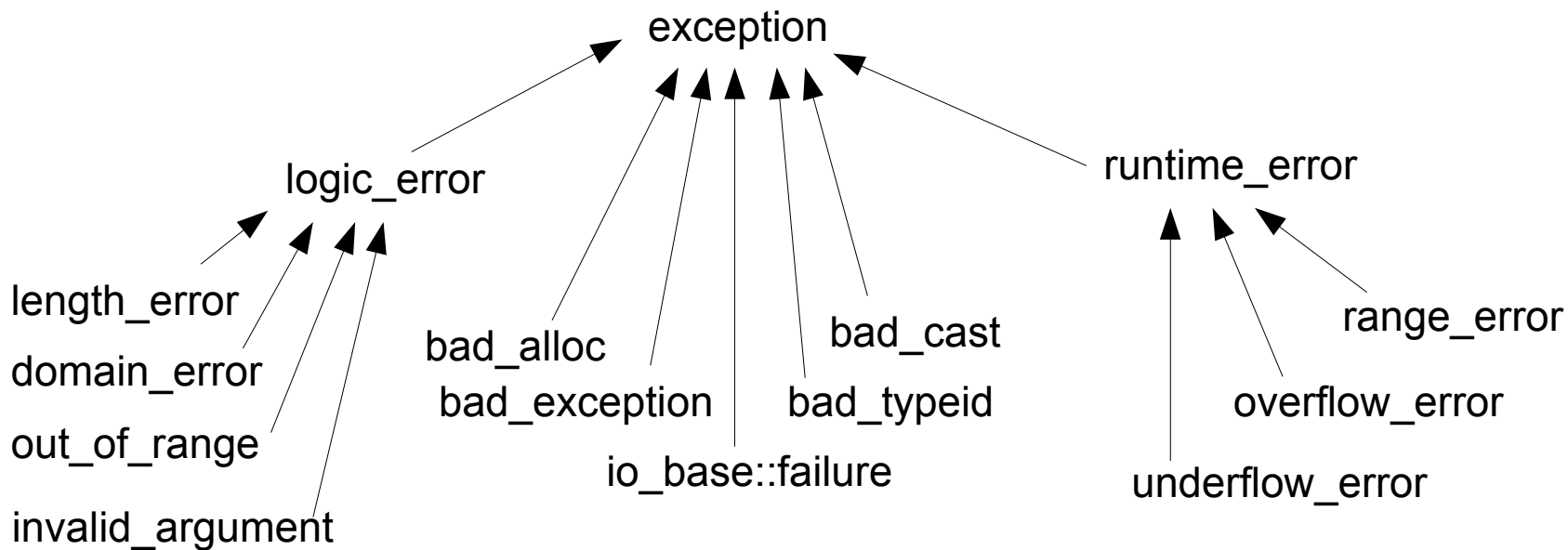
```
int main()
{
    constexpr unsigned long long v = Factorial(66);
    std::cout << v << std::endl;
    return 0;
}
```

# Система обработки исключительных ситуаций

Предназначена для обработки стандартных и пользовательских исключений, позволяет обрабатывать ошибки не в том месте, где они возникли.

```
try
{
    // Код, где возможно исключение.
}
catch (std::exception const & e)
{
    // Обработчик исключения.
}
```

# Иерархия исключений стандартной библиотеки



## Выбрасывание исключений (плохой пример)

```
try
{
    if (y == 0) throw std::string("Division by zero");
    z = x / y;
}
catch (std::string const & ex)
{
    std::cerr << ex;
}
```

## Выбрасывание исключений (хороший пример)

```
try
{
    if (y == 0) throw std::invalid_argument("Division by zero!");
    z = x / y;
}
catch (std::exception const & ex)
{
    std::cerr << ex.what();
}
```

## Выбрасывание исключений (еще один хороший пример)

```
class ExceptionDivisionByZero : public std::exception {};
```

```
try
{
    if (y == 0) throw ExceptionDivisionByZero();
    z = x / y;
}
catch (ExceptionDivisionByZero const & ex)
{
    std::cerr << "Division by zero!";
}
```

## Исключение в конструкторе

```
class Gun
{
public:
    Gun(unsigned int health)
    {
        if (health > 100)
            throw std::invalid_argument("Health must be in range 0..100");
    }
};
```



Если исключение прерывает выполнение конструктора, то объект не считается сконструированным!  
Деструктор для него вызван не будет.

## Исключение в деструкторе

```
class Gun
{
public:
    ~Gun()
    {
        try
        {
            // Do something...
        }
        catch(std::exception const & ex)
        {
            // Process an exception.
        }
    }
};
```

Нельзя выпускать исключения за пределы деструктора!

Иначе Undefined Behaviour.

## Catch all

```
int main()
{
    try
    {
        // Execute program...
    }
    catch(...)
    {
        std::cerr << "Unknown error!";
    }
    return 0;
}
```

Хорошая практика использовать такой отлов исключений однажды на всю программу, в главной функции!

## Цепочка исключений

```
try
{
    // Do something.
}
catch(std::bad_alloc const & ex)
{
    // Process lack of memory.
}
catch(std::bad_cast const & ex)
{
    // Process incorrect cast.
}
```

Исключение выбирает первый наиболее подходящий блок!

## Наследники в цепочке исключений

```
try
{
    // Do something.
}
catch(std::bad_alloc const & ex)
{
    // Process lack of memory.
}
catch(std::exception const & ex)
{
    // Process all other exceptions inherited from std::exception.
}
```

Сначала ищется точное соответствие, затем соответствие по базовым классам вверх по цепочке.

## Ошибки построения цепочки исключений

```
try
{ }
catch(std::exception const & ex)
{ }
catch(...) // Ошибка!
{ }
catch(std::bad_alloc const & ex)
{ }
```

Правильно располагать блоки **catch** в порядке от частного к общему!

## Правильный порядок

```
try
{ }
catch(std::bad_alloc const & ex)
{ }
catch(std::exception const & ex)
{ }
catch(...)
{ }
```

## Проброс исключения

```
try
{
    try
    {
        // Здесь произошла ошибка выделения памяти.
    }
    catch (std::bad_alloc const & ex)
    {
        // Выдаем сообщение, и перевыбрасываем.
        throw;
    }
}
catch (std::exception const & ex)
{
    // Заносим в лог.
}
```



## noexcept

- ключевое слово, позволяющее пометить функцию (метод) как невыбрасывающую исключения.

Необходим, например, для определения возможности использовать семантику перемещения.

```
void foo() noexcept  
{  
}
```

Можно использовать как оператор `noexcept(expression)`, возвращает `bool`.

## Домашнее задание

- 1) Добавить в классы, созданные в предыдущих ДЗ, проверки на ошибки при помощи исключений. Доработайте тесты.
- 2) Пересмотрите использование контейнерных классов в предыдущих ДЗ. Выберите правильные коллекции.

**Срок сдачи: 20.10.2016 23:59:59**

**Просьба оставить отзыв о  
данном занятии на портале!**

# Спасибо за внимание!

Роман Кузнецов

[r.kuznetsov@mapswithme.com](mailto:r.kuznetsov@mapswithme.com)