Модуль 13

- Лямбды
- Лямбда-захват
- Исключения
- Обработка исключений
- Классы исключений
- Недостатки использования исключений

Лямбды

В C++, лямбда-выражение (или просто лямбда) представляет собой анонимную функцию, которая может быть определена прямо внутри другой функции или выражения. Лямбда-выражения позволяют создавать краткие функции без явного объявления их имени или типа, что делает их удобными для использования в качестве аргументов функций или для определения функциональных объектов в местах, где требуется локальная функциональность.

Лямбды

```
[ captureClause ] ( параметры ) -> возвращаемыйТип
{
стейтменты;
}
```

Лямбды

1. `[] (параметры) { действия }` - структура лямбды

   `[] (параметры) -> type { действия }`

2. `[]() { std::cout << "Hello" << std::endl; }` - пример

3.
```
class Lambda
{
public:
    auto operator()() const { std::cout << "Hello" << std::endl; }
};
```
- “такой” класс генерирует компилятор

4. `[]{ std::cout << "Hello" << std::endl; }` - лямбда без параметров

5. `[](){std::cout << "Hello" << std::endl;} ();` - Запуск лямбды

Лямбды

6. `auto hello { [](){std::cout << "Hello" << std::endl;} };` - присвоение лямбда-выражению переменной.

7. `auto print { [](const std::string& text){std::cout << text << std::endl;} };` - лямбда с параметрами

8. `auto sum { [](int a, int b) -> double {return a + b;} };` - возврат значения из лямбды

9.
```
void function (int a, int b, int (*op)(int, int))
{
    std::cout << op(a, b) << std::endl;
}
```
- функция принимающая лямбду как параметр

   `auto sum { [](int a, int b) -> int return a + b;} };` - лямбда

10. `auto hello = [](const auto& value) {std::cout << value << std::endl; };` - (generic lambda) - auto параметр

Лямбда-захват

1. **все внешние переменные из области видимости**
   a. `[=](int x) { std::cout << x + n << std::endl; };` - по значению. Внешние переменные сохраняется в приватную константную переменную
      i. `[=](int x) mutable { std::cout << x + (++n) << std::endl; };` - (mutable) редактирование переменной, но результат локальный
   b. `[&](int x) { std::cout << x + n << std::endl; };` - по ссылке, редактируется внешняя переменная
2. **Определенные переменные**
   a. `[n](){ std::cout << "n: " << n << std::endl;};` - по значению
   b. `[&n](){ std::cout << "n: " << n << std::endl;};` - по ссылке
   c. `[=, &m, &n]` - все перменные по значению, m, n - по ссылке
3. **Захват членов класса**

```
[this](){ std::cout << text << std::endl;}
```

Исключения

Оператор throw используется для сигнализирования о возникновении исключения или ошибки. Сигнализирование о том, что произошло исключение, называется **генерацией исключения** (или ***«выбрасыванием исключения»***).

```
throw -1; // генерация исключения типа int
throw ENUM_INVALID_INDEX; // генерация исключения типа enum
throw "Can not take square root of negative number"; // генерация исключения типа const char* (строка C-style)
throw dX; // генерация исключения типа double (переменная типа double, которая была определена ранее)
throw MyException("Fatal Error"); // генерация исключения с использованием объекта класса MyException
```

Исключения

```
try
{
    // Здесь мы пишем стейтменты, которые будут генерировать следующее
исключение
    throw -1; // типичный стейтмент throw
}
catch (int a)
{
    // Обрабатываем исключение типа int
    std::cerr << "We caught an int exception with value" << a << '\n';
}
```

Исключения

Обработка исключений

- ➔ При выбрасывании исключения (оператор throw), точка выполнения программы немедленно переходит к ближайшему блоку try. Если какой-либо из обработчиков catch, прикрепленных к блоку try, обрабатывает этот тип исключения, то точка выполнения переходит в этот обработчик и, после выполнения кода блока catch, исключение считается обработанным.
- ➔ Если подходящих обработчиков catch не существует, то выполнение программы переходит к следующему блоку try. Если до конца программы не найдены соответствующие обработчики catch, то программа завершает свое выполнение с ошибкой исключения.
- ➔ Компилятор не выполняет неявные преобразования при сопоставлении исключений с блоками catch.

Исключения. Что должен делать блок **catch**?

Если исключение направлено в блок catch, то оно считается «обработанным», даже если блок catch пуст. Стратегия обработки исключений:

- ➔ Во-первых, блок catch может вывести сообщение об ошибке (либо в консоль, либо в лог-файл).
- ➔ Во-вторых, блок catch может возвратить значение или код ошибки обратно в caller.
- ➔ В-третьих, блок catch может сгенерировать другое исключения. Поскольку блок catch не находится внутри блока try, то новое сгенерированное исключение будет обрабатываться следующим блоком try.

Исключения

Если произошла ошибка необработанного исключения, каждая операционная система может решить эту ситуацию по-своему:

- ➔ либо выведет сообщение об ошибке;
- ➔ либо откроет диалоговое окно с ошибкой;
- ➔ либо просто сбой.

`catch (...)` – обрабатывает любой тип исключения

Классы-исключения

Конструкторы — это часть классов, в которых исключения могут быть очень полезными. Если конструктор не сработал, то сгенерируйте исключение, которое сообщит, что объект не удалось создать. Создание объекта прерывается, а **деструктор** никогда не выполняется (обратите внимание, это означает, что ваш конструктор должен самостоятельно выполнять очистку памяти перед генерацией исключения).

Классы-исключения

Класс-Исключение — это обычный класс, который выбрасывается в качестве исключения.

std::exception — это небольшой **интерфейсный класс**, который используется в качестве родительского класса для любого исключения, которое выбрасывается в Стандартной библиотеке C++.

Исключения

Недостатки исключений:

- ➔ Очистка памяти
- ➔ Исключения и деструкторы

  исключения *никогда* не должны генерироваться в деструкторах. Из-за того, что происходит раскручивание стека компилятор не знает, продолжать ли процесс раскручивания стека или обработать новое исключение. Результатом будет завершение программы.

- ➔ Проблемы с производительностью

Исключения

Исключения и их обработку лучше всего использовать, **если выполняются все следующие условия**:

- ➔ Обрабатываемая ошибка возникает редко.
- ➔ Ошибка является серьезной, и выполнение программы не может продолжаться без её обработки.
- ➔ Ошибка не может быть обработана в том месте, где она возникает.
- ➔ Нет хорошего альтернативного способа вернуть код ошибки обратно в caller.

Классы-исключения

Спецификатор noexcept определяет функцию как *не выбрасывающую* исключений. Чтобы определить функцию как *не выбрасывающую*, мы можем использовать спецификатор **noexcept** в объявлении функции, поместив его справа от списка параметров функции:

```
void doSomething() noexcept;
```

noexcept на самом деле не запрещает функции выбрасывать исключения или вызывать другие функции, которые *потенциально могут выбросить исключения*. Скорее всего, при возникновении исключения, если оно происходит из noexcept-функции, будет вызвана функция std::terminate(). И обратите внимание, что если std::terminate() вызывается внутри noexcept-функции, то раскручивание стека может происходить, а может и не происходить (в зависимости от реализации и оптимизации). А это означает, что ваши объекты могут быть уничтожены должным образом до завершения работы, а может и не произойти этого уничтожения.