Лабораторная работа №6

по курсу «Высокоуровневое программирование» (2 семестр)

«Исключения и обработка исключений»

Оглавление

Цель: приобретение практических навыков и знаний по обработке ошибок и исключений в программе, во время её выполнения.

Задачи:

1. Познакомиться с типами ошибок;
2. Научиться обрабатывать ошибки при компиляции;
3. Познакомиться с концепцией исключений;
4. Научиться вызывать и обрабатывать исключения;
5. Получение навыков по созданию своих типов исключений;
6. Изучить дополнительные навыки по работе с ООП.

Содержание отчёта:

1. Титульный лист;
2. Цель, задачи работы;
3. Формулировка общего задания;
4. UML-диаграмма созданного приложения;
5. Листинги пользовательских функций, классов и основной программы;
6. Результаты работы;
7. Выводы по работе в целом.

В начало

Основные теоретические сведения

Обработка ошибок

Во время выполнения программы в ней могут появляться различного рода ошибки, которые останавливают выполнение программы до того, как она сможет повредить данные. Ошибки делятся на несколько видов:

- Синтаксические ошибки – те, которые появляются во время компиляции при неправильном использовании средств языка. Синтаксические ошибки почти всегда улавливаются компилятором и их обычно легко исправить. Следовательно, о них слишком беспокоиться не стоит.
- Семантические ошибки – такие возникают, когда код отрабатывает, но не так, как нужно программисту. Такие ошибки очень коварны, ведь найти их очень сложно.

Для того, чтобы бороться с семантическими ошибками существует такое понятие, как безопасное программирование. Это методика разработки программ, которая включает анализ областей, где могут быть допущены ложные предположения, и написание кода, который обнаруживает и обрабатывает любой случай такого нарушения, чтобы свести к минимуму риск возникновения сбоя или повреждения программы.

Для вывода сообщений об ошибках существует перегруженная версия потока вывода: cerr, который работает в точности, как cout, за тем исключением, что некоторые терминалы подсвечивают его вывод другим цветом.

Если нам нужно аварийно завершить программу, выбросив отличный от нуля код ошибки, нам понадобиться функция: exit(), которая в качестве параметров принимает код, который

необходимо вернуть в операционную систему. Для того, чтобы использовать эту функцию, необходимо подключить заголовочный файл: «cstdlib».

В начало

Стэйтмент assert

Стейтмент assert (или «оператор проверочного утверждения») в языке C++ — это макрос препроцессора, который обрабатывает условное выражение во время выполнения. Если условное выражение истинно, то стейтмент assert ничего не делает. Если же оно ложное, то выводится сообщение об ошибке, и программа завершается. Это сообщение об ошибке содержит ложное условное выражение, а также имя файла с кодом и номером строки с assert. Таким образом, можно легко найти и идентифицировать проблему, что очень помогает при отладке программ.

Сам assert реализован в заголовочном файле cassert и часто используется как для проверки корректности переданных параметров функции, так и для проверки возвращаемого значения функции:

```
void check(int num) {
	assert(num > 10, "Num < 10!");
}
```

Если в функцию будет передано число меньше 10, программа аварийно завершиться, а в консоли отобразиться наше сообщение, переданное вторым параметром.

Функция assert() тратит мало ресурсов на проверку условия. Кроме того, стейтменты assert (в идеале) никогда не должны встречаться в релизном коде (потому что ваш код к этому моменту уже должен быть тщательно протестирован). Следовательно, многие разработчики предпочитают использовать assert только в конфигурации Debug. В языке C++ есть возможность отключить все assert-ы в релизном коде — использовать директиву #define NDEBUG. Некоторые IDE устанавливают NDEBUG по умолчанию, как часть параметров проекта в конфигурации Release.

Обратите внимание, функция exit() и assert (если он срабатывает) немедленно прекращают выполнение программы, без возможности выполнить дальнейшую любую очистку (например, закрыть файл или базу данных). Из-за этого их следует использовать аккуратно.

В начало

static_assert

В C++11 добавили еще один тип assert-а — static_assert. В отличие от assert, который срабатывает во время выполнения программы, static_assert срабатывает во время компиляции, вызывая ошибку компилятора, если условие не является истинным. Если условие ложное, то выводится диагностическое сообщение:

```
static_assert(sizeof(long) == 8, "long must be 8 bytes");
static_assert(sizeof(int) == 4, "int must be 4 bytes");
```

Обработка исключений. Операторы throw, try и catch

Мы постоянно используем сигналы в реальной жизни для обозначения того, что произошли определенные события. Например, во время игры в баскетбол, если игрок совершил серьезный фол, то арбитр свистит, и игра останавливается. Затем идет штрафной бросок. Как только штрафной бросок выполнен, игра возобновляется.

В языке C++ оператор throw используется для сигнализирования о возникновении исключения или ошибки (аналогия тому, когда свистит арбитр). Сигнализирование о том, что произошло исключение, называется генерацией исключения (или «выбрасыванием исключения»).

Для использования оператора throw применяется ключевое слово throw, а за ним указывается значение любого типа данных, которое вы хотите задействовать, чтобы сигнализировать об ошибке. Как правило, этим значением является код ошибки, описание проблемы или настраиваемый класс-исключение. Например:

```
	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
```

Выбрасывание исключений — это лишь одна часть процесса обработки исключений. Вернемся к нашей аналогии с баскетболом: как только просвистел арбитр, что происходит дальше? Игроки останавливаются, и игра временно прекращается. Обычный ход игры нарушен.

В языке C++ мы используем ключевое слово try для определения блока стейтментов (так называемого «блока try»). Блок try действует как наблюдатель в поисках исключений, которые были выброшены каким-либо из операторов в этом же блоке try, например:

```
void getException(const char* my_exce) {
	throw my_exce;
}

int main() {

	try {
		/*
			Сюда мы помещаем код, который
			может вызвать исключение
		*/
		getException("My exce");
	}

	return 0;
}
```

Обратите внимание, блок try не определяет, КАК мы будем обрабатывать исключение. Он просто сообщает компилятору: «Эй, если какой-либо из стейтментов внутри этого блока try сгенерирует исключение — поймай его!».

Пока арбитр не объявит о штрафном броске, и пока этот штрафной бросок не будет выполнен, игра не возобновится. Другими словами, штрафной бросок должен быть обработан до возобновления игры.

Фактически, обработка исключений — это работа блока(ов) catch. Ключевое слово catch используется для определения блока кода (так называемого «блока catch»), который обрабатывает исключения определенного типа данных.

Вот пример блока catch, который обрабатывает (ловит) исключения типа const char*:

```
void getException(const char* my_exce) {
	throw my_exce;
}

int main() {

	try {
		/*
			Сюда мы помещаем код, который
			может вызвать исключение
		*/
		getException("My exce");
	}
	catch (const char* my_exce) {
		cerr << "Error! Name error: " << my_exce << endl;
	}

	return 0;
}
```

Блоки try и catch работают вместе. Блок try обнаруживает любые исключения, которые были выброшены в нем, и направляет их в соответствующий блок catch для обработки. Блок try должен иметь, по крайней мере, один блок catch, который находится сразу же за ним, но также может иметь и несколько блоков catch, размещенных последовательно (друг за другом). Как только исключение было поймано блоком try и направлено в блок catch для обработки, оно считается обработанным (после выполнения кода блока catch), и выполнение программы возобновляется.

Параметры catch работают так же, как и параметры функции, причем параметры одного блока catch могут быть доступны и в другом блоке catch (который находится за ним). Исключения фундаментальных типов данных могут быть пойманы по значению (параметром блока catch является значение), но исключения не фундаментальных типов данных должны быть пойманы по константной ссылке (параметром блока catch является константная ссылка), дабы избежать ненужного копирования.

В начало

Непойманные исключения

А теперь загадка: «Функции могут генерировать исключения любого типа данных, и, если исключение не поймано, это приведет к раскручиванию стека и потенциальному завершению выполнения всей программы. Поскольку мы можем вызывать функции, не зная их реализации (и, следовательно, какие исключения они могут генерировать), то как мы можем это предотвратить?».

К счастью, язык C++ предоставляет нам механизм обнаружения/обработки всех типов исключений — обработчик catch-all. Обработчик catch-all работает так же, как и обычный блок catch, за исключением того, что вместо обработки исключений определенного типа данных, он использует эллипсис (...) в качестве типа данных.

А как мы уже знаем, эллипсисы могут использоваться для передачи аргументов любого типа данных в функцию. В этом контексте они представляют собой исключения любого типа данных. Вот простой пример:

```
	try {
		/*
			Сюда мы помещаем код, который
			может вызвать исключение
		*/
		getException("My exce");
	}
	catch (...) {
		cerr << "Error! Some error!" << endl;
	}
```

В начало

Классы исключения

Одной из основных проблем использования фундаментальных типов данных (например, типа int) в качестве типов исключений является то, что они, по своей сути, являются неопределенными. Еще более серьезной проблемой является неоднозначность того, что означает исключение, когда в блоке try имеется несколько стейтментов или вызовов функций.

Одним из способов решения этой проблемы является использование классов-исключений. Класс-Исключение — это обычный класс, который выбрасывается в качестве исключения. Создадим простой класс-исключение:

```
class MyException {
public:
	MyException(const char* error) : m_error(error) {}

	const char* getError() {
		return m_error;
	}

private:
	const char* m_error{};
};
```

Используем наше исключение:

```
void getException(const char* my_exce) {
	throw MyException(my_exce);
}

int main() {

	try {
		/*
			Сюда мы помещаем код, который
			может вызвать исключение
		*/
		getException("My exce");
	}
	catch (MyException &exec) {
		cerr << "Error! Text error: " << exec.getError() << endl;
	}

	return 0;
```

```
}
```

Используя такой класс, мы можем генерировать исключение, возвращающее описание возникшей проблемы, это даст нам точно понять, что именно пошло не так. И, поскольку исключение MyException имеет уникальный тип, мы можем обрабатывать его соответствующим образом (не так как другие исключения).

Обратите внимание, в обработчиках исключений объекты класса-исключения принимать нужно по ссылке, а не по значению. Это предотвратит создание копии исключения компилятором, что является затратной операцией (особенно в случае, когда исключение является объектом класса), и предотвратит обрезку объектов при работе с дочерними классами-исключениями. Передачу по адресу лучше не использовать, если у вас нет на это веских причин.

В начало

Наследование исключений

Так как мы можем выбрасывать объекты классов в качестве исключений, а классы могут быть получены из других классов, то нам нужно учитывать, что произойдет, если мы будем использовать унаследованные классы в качестве исключений. Оказывается, обработчики могут обрабатывать исключения не только одного определенного класса, но и исключения дочерних ему классов. Правило: Обработчики исключений дочерних классов должны находиться пере Многие классы и операторы из Стандартной библиотеки C++ выбрасывают классы-исключения при сбое. Например, оператор new и std::string могут выбрасывать std::bad_alloc при нехватке памяти. Неудачное динамическое приведение типов с помощью оператора dynamic_cast выбрасывает исключение std::bad_cast и т.д. Начиная с C++14, существует больше 20 классов-исключений, которые могут быть выброшены, а в C++17 их еще больше.

Хорошей новостью является то, что все эти классы-исключения являются дочерними классу std::exception. std::exception — это небольшой интерфейсный класс, который используется в качестве родительского класса для любого исключения, которое выбрасывается в Стандартной библиотеке C++.

В большинстве случаев, если исключение выбрасывается Стандартной библиотекой C++, то нам все равно, было ли это неудачное выделение, конвертирование или что-либо другое. Нам достаточно знать, что произошло что-то катастрофическое, из-за чего в нашей программе произошел сбой. Благодаря std::exception мы можем настроить обработчик исключений типа std::exception, который будет ловить и обрабатывать как std::exception, так и все (20+) дочерние ему классы-исключения.д обработчиками исключений родительского класса.

```
#include <iostream>
#include <exception> // для std::exception
#include <string> // для этого примера

int main() {
	try {
		// Здесь должен находиться код, использующий Стандартную библиотеку C++.
		// Сейчас мы намеренно спровоцируем генерацию одного из исключений
		std::string s;
		s.resize(-1); // генерируется исключение std::bad_alloc
	}
```

```
	// Этот обработчик ловит std::exception и все дочерние ему классы-исключения
	catch (std::exception& exception) {
		std::cerr << "Standard exception: " << exception.what() << '\n';
	}

	return 0;
}
```

Ничто не генерирует std::exception напрямую, и вы также должны придерживаться этого правила. Однако, вы можете генерировать исключения других классов из Стандартной библиотеки C++, если они адекватно отражают ваши потребности. Найти список всех стандартных классов-исключений из Стандартной библиотеки C++ вы можете здесь.

std::runtime_error (находится в заголовочном файле stdexcept) является популярным выбором, так как имеет общее имя, а конструктор принимает настраиваемое сообщение:

```
#include <iostream>
#include <stdexcept>

int main() {
	try {
		throw std::runtime_error("Bad things happened");
	}
	// Этот обработчик ловит std::exception и все дочерние ему классы-исключения
	catch (std::exception& exception) {
		std::cerr << "Standard exception: " << exception.what() << '\n';
	}

	return 0;
}
```

Конечно, вы можете создать свои собственные классы-исключения, дочерние классу std::exception, и переопределить виртуальный константный метод what().

Нужно понимать, что исключения очень затратная операция и их нужно использовать в крайней необходимости. Например, в следующих случаях:

- 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.

В начало

Функции декораторы, или функции высшего порядка

Нам уже известно, что имя функции представляет из себя указатель на блок кода в памяти компьютера. Мы умеем создавать переменные функций, массивы функций, а также вызывать эти функции. Пришло время попробовать передавать функции, как параметры других функций.

Функции, которые принимают в аргументах другие функции, называются функциями высшего порядка. Функции высшего порядка, которые после каких-то манипуляций до, или после выполнения переданной функции запускают её, называются – декораторами.

Давайте рассмотрим такой пример: у нас имеется функция, которая может вызвать какое – то исключение и нам его нужно обработать, но обрабатывать это исключение мы хотим в

особом порядке с помощью кодов возврата, как нам это сделать? Можно написать функцию, которая в себе будет запускать нужную функцию, принимает её возвращаемое значение и при возникновении ошибки, выполнять какие – то действия:

```
int getErrCode() {
	return 2;
}

void funcDecoration(int(*func)()) {
	switch (func()) {
	case 1:
		cerr << "Error code 1" << endl;
		break;
	case 2:
		cerr << "Error code 2" << endl;
		break;
	case 3:
		cerr << "Error code 3" << endl;
		break;
	default:
		cerr << "All good!" << endl;
		break;
	}
}

int main() {

	funcDecoration(getErrCode);

	return 0;
}
```

Функция funcDecoration в качестве параметров принимает функцию, которую в себе вызывает и сравнивает код ошибки, который эта функция вернула, а затем пишет необходимое сообщение.

Теперь возьмём другой пример. Скажем, у нас есть функция сортировки массива:

```
void sortArr(int* arr, int len) {
	for (int i{}; i < len; ++i) {
		for (int j{1}; j < (len - i); ++j) {
			if (arr[i] > arr[j]) {
				swap(arr[i], arr[j]);
			}
		}
	}
}
```

Она прекрасно отрабатывает, но, что если мы хотим поменять принцип сортировки? Нам придётся переписывать функцию, а ели нам нужно в коде использовать несколько принципов сортировки? Нам нужно будет писать несколько функций с почти одинаковым кодом? На помощь приходят функции высших порядков. Мы можем третьим параметром передавать функцию, которая будет выполнять алгоритм сортировки, давайте так и сделаем. Наш прототип такой функции будет выглядеть так: bool (*func)(int, int)

На вход она принимает два числа, а возвращает логическое выражение. Теперь протестируем нашу функцию:

```
void sortArr(int* arr, int len, bool (*func)(int, int)) {
	for (int i{}; i < len; ++i) {
		for (int j{1}; j < (len - i); ++j) {
			if (func(arr[i], arr[j])) {
```

```
                        swap(arr[i], arr[j]);
                    }
                }
        }
}

bool f_1(int a, int b) {
        return a > b;
}

int main() {

        int a[] = { 1, 2, 5, 4, 3 };

        sortArr(a, 5, f_1);

        return 0;
}
```

И снова наша программа работает. Таким образом можно задавать любые параметры сортировки, а также фильтрации, поиска и т д.

В начало

Функторы

Функторы – это сокращение от функциональных объектов. Функциональный объект – это класс, в котором выполнена перегрузка оператора «()». Функциональные объекты могут вести себя, как функции, но при этом хранить какое-то состояние в виде полей. По сколько перегруженный оператор это, по сути, обычный метод, то у него есть доступ ко всем полям объекта, они формируют для него некое окружение, которое не меняется при перемещении объекта класса. Такое состояние окружения называется – замыканием. Замыкание позволяет замораживать состояние работы приложения вокруг функционального объекта, формируя тем самым стек состояний, который может пригодиться при работе с потоками выполнения.

Давайте напишем функцию, которая каким-то образом обрабатывает массив:

```
template<class T>
class ProcessingArr {
public:
    virtual T res() = 0;

    virtual void operator()(T v) = 0;
};

template<class T>
void ForArr(T* arr, size_t len, ProcessingArr<T>& func) {
    for (int i{}; i < len; ++i) {
        func(arr[i]);
    }
}
```

Мы создали функцию, которая применяет функцию из аргументов к каждому элементу массива. Функция в аргументах является классом интерфейсом, который мы перегрузим для создания нужного нам поведения:

```
template<class T>
class Sum : public ProcessingArr<T> {
public:
```

```
    explicit Sum(T x) : m_sum(x) {}

    T res() override {
        return m_sum;
    }

    void operator()(T v) override {
        m_sum += v;
    }

private:
    int m_sum{};
};
```

Данный класс принимает в перегруженном операторе один параметр, который потом суммируется к полю m_sum. Это поле и есть замыкание для нашего перегруженного оператора. Сам класс наследуется от интерфейса и имплементирует его методы. Таким образом мы можем использовать объект нашего дочернего класса в функции по работе с массивом. Давай создадим объект:

```
int main() {

    int a[] = { 1, 2, 5, 4, 3 };

    Sum<int> sum_arr_a(0);

    ForArr<int>(a, 5, sum_arr_a);

    cout << sum_arr_a.res() << endl;

    return 0;
}
```

Таким образом, создавая новые классы и передавая их объекты в одну функцию, можно добиться различного поведения. Сами объекты – функторы, очень удобно использовать из – за их гибкости. Таким образом, там, куда можно передать функцию, можно передать функтор.

В начало

Лямбда функции

Лямбда выражения, или лямбда функции в программировании позволяют определить анонимную функцию внутри другой функции. Возможность сделать функцию вложенной является очень важным преимуществом, так как позволяет избегать как захламления пространства имен лишними объектами, так и определить функцию как можно ближе к месту её первого использования. По сути, в языке C++ лямбды — это те же функторы, только с изменённым синтаксисом, под капотом они работают точно также. Объявление лямбды в языке C++ весьма своеобразно, но к нему легко привыкнуть, если понять:

```
int main() {

    [ /* замыкание */ ]( /* параметры */ ) ->/* возвращаемый тип */{ /* стэйтменты */ };

    return 0;
}
```

Поле замыкание, параметры и возвращаемый тип могут быть опущены. Также и стэйтменты могут быть опущены.

```
[]() {};
```

Главный смысл лямбды в том, что эта функция не имеет имени, следовательно мы её можем создавать на ходу и передавать в те места, где нужно выполнить какое – то действие только раз. Таким образом, прошлую реализацию функции по обработке массива, можно было бы переписать с лямбдой. Мы можем определилить лямбду прямо в том месте, где она была нам нужна. Такое использование лямбда-выражения иногда еще называют функциональным литералом. Однако написание лямбды в той же строке, где она используется, иногда может затруднить чтение кода. Подобно тому, как мы можем инициализировать переменную с помощью литерала (или указателя на функцию) для использования в дальнейшем, так же мы можем инициализировать и лямбда-переменную с помощью лямбда-определения для её дальнейшего использования. Именованная лямбда вместе с удачным именем функции может облегчить чтение кода.

Давайте перепишем с помощью лямбды наш функтор:

```
    auto t = [&a](int v) { a += v; };
```

Здесь в качестве типа переменной выступает слово: auto, которая автоматически выводит тип этой переменной. В области захвата переменных лежит ссылка на внешнюю переменную. В скобках идут параметры функции, а в теле команда подсчёта.

У лямбд нет типа, который мы могли бы явно использовать. Когда мы пишем лямбду, компилятор генерирует уникальный тип лямбды, который нам не виден. На самом деле, лямбды не являются функциями (что и помогает им избегать ограничений C++, которые накладываются на использование вложенных функций). Лямбды являются особым типом объектов, который называется функтором. Функторы — это объекты, содержащие перегруженный operator(), который и делает их вызываемыми подобно обычным функциям.

Хотя мы не знаем тип лямбды, есть несколько способов её хранения для использования после определения. Если лямбда ничего не захватывает, то мы можем использовать обычный указатель на функцию. Как только лямбда что-либо захватывает, указатель на функцию больше не будет работать. Однако std::function может использоваться для лямбд, даже если они что-то захватывают:

```
#include <functional>

int main() {
    // Обычный указатель на функцию. Лямбда не может ничего захватить
    double (*addNumbers1)(double, double) {
        [](double a, double b) {
            return (a + b);
        }
    };

    addNumbers1(1, 2);

    // Используем std::function. Лямбда может захватывать переменные.
    std::function<int(double, double)> addNumbers2 { // примечание: Если у вас не
поддерживается C++17 и выше, используйте std::function<double(double, double)>
      [](double a, double b) {
        return (a + b);
      }
    };
```

```
    addNumbers2(3, 4);

    // Используем auto. Храним лямбду с её реальным типом
    auto addNumbers3 {
      [](double a, double b) {
        return (a + b);
      }
    };

    addNumbers3(5, 6);

    return 0;
}
```

С помощью auto мы можем использовать фактический тип лямбды. При этом мы можем получить преимущество в виде отсутствия накладных расходов в сравнении с использованием std::function.

К сожалению, мы не всегда можем использовать auto. В тех случаях, когда фактический тип лямбды неизвестен (например, из-за того, что мы передаем лямбду в функцию в качестве параметра, и вызывающий объект сам определяет какого типа лямбда будет передана), мы не можем использовать auto. В таких случаях следует использовать std::function:

```
#include <functional>
#include <iostream>

// Мы не знаем, чем будет fn. std::function работает с обычными функциями и лямбдами
void repeat(int repetitions, const std::function<void(int)>&fn) {
    for (int i{ 0 }; i < repetitions; ++i)
    {
        fn(i);
    }
}

int main() {
    repeat(3, [](int i) {
        std::cout << i << '\n';
        });

    return 0;
}
```

Одним примечательным исключением является то, что, начиная с C++14, нам разрешено использовать auto с параметрами функций.

В начало

Вывод возвращаемого типа и возвращаемые типы trailing

Если использовался вывод возвращаемого типа, то возвращаемый тип лямбды выводится из стейтментов return внутри лямбды. Если использовался вывод возвращаемого типа, то все возвращаемые стейтменты внутри лямбды должны возвращать значения одного и того же типа (иначе компилятор не будет знать, какой из них ему следует использовать). Например:

```
#include <iostream>

int main() {
    auto divide{ [](int x, int y, bool bInteger) { // примечание: Не указан тип
возвращаемого значения
```

```
      if (bInteger)
        return x / y;
      else
        return static_cast<double>(x) / y; // ОШИБКА: Тип возвращаемого значения не
совпадает с предыдущим возвращаемым типом
    } };

    std::cout << divide(3, 2, true) << '\n';
    std::cout << divide(3, 2, false) << '\n';

    return 0;
}
```

Это приведет к ошибке компиляции, так как тип возвращаемого значения первого стейтмента return (int) не совпадает с типом возвращаемого значения второго стейтмента return (double).

В случае, когда у нас используются разные возвращаемые типы, у нас есть два варианта:

- выполнить явные преобразования в один тип;
- явно указать тип возвращаемого значения для лямбды и позволить компилятору выполнить неявные преобразования.

Второй вариант обычно является более предпочтительным:

```
#include <iostream>

int main() {
    // Примечание: Явно указываем тип double для возвращаемого значения
    auto divide{ [](int x, int y, bool bInteger) -> double {
      if (bInteger)
        return x / y; // выполнится неявное преобразование в тип double
      else
        return static_cast<double>(x) / y;
    } };

    std::cout << divide(3, 2, true) << '\n';
    std::cout << divide(3, 2, false) << '\n';

    return 0;
}
```

Таким образом, если вы когда-либо решите изменить тип возвращаемого значения, вам (как правило) нужно будет изменить только тип возвращаемого значения лямбды и ничего внутри основной части.

В начало

Лямбда захваты

Поле capture clause используется для того, чтобы предоставить (косвенно) лямбде доступ к переменным из окружающей области видимости, к которым она обычно не имеет доступ. Всё, что нам нужно для этого сделать, так это перечислить в поле capture clause объекты, к которым мы хотим получить доступ внутри лямбды. В нашем примере мы хотим предоставить лямбде доступ к значению переменной search, поэтому добавляем её в захват:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string_view>
```

```cpp
#include <string>

int main() {
    std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };

    std::cout << "search for: ";

    std::string search{};
    std::cin >> search;

    // Захват переменной search
    auto found{ std::find_if(arr.begin(), arr.end(), [search](std::string_view str) {
      return (str.find(search) != std::string_view::npos);
    }) };

    if (found == arr.end()) {
        std::cout << "Not found\n";
    }
    else {
        std::cout << "Found " << *found << '\n';
    }

    return 0;
}
```

Хотя может показаться, будто в вышеприведенном примере наша лямбда напрямую обращается к значению переменной search (относящейся к блоку кода функции main()), но это не так. Да, лямбды могут выглядеть и функционировать как вложенные блоки, но на самом деле они работают немного по-другому, и при этом существует довольно важное отличие.

Когда выполняется лямбда-определение, то для каждой захватываемой переменной внутри лямбды создается клон этой переменной (с идентичным именем). Данные переменные-клоны инициализируются с помощью переменных из внешней области видимости с тем же именем

Таким образом, в примере, приведенном выше, при создании объекта лямбды, она получает свою собственную переменную-клон с именем search. Эта переменная имеет такое же значение, что и переменная search из функции main(), поэтому кажется будто мы получаем доступ непосредственно к переменной search функции main(), но это не так.

Несмотря на то, что эти переменные-клоны имеют одно и то же имя, их тип может отличаться от типа исходной переменной (об этом чуть позже).

Ключевой момент: Переменные, захваченные лямбдой, являются клонами переменных из внешней области видимости, а не фактическими «внешними» переменными.

По умолчанию переменные захватываются как константные значения. Это означает, что при создании лямбды, она захватывает константную копию переменной из внешней области видимости, что означает, что значения этих переменных лямбда изменить не может.

Чтобы разрешить изменения значения переменных, которые были захвачены по значению, мы можем пометить лямбду как mutable. В данном контексте ключевое слово mutable удаляет спецификатор const со всех переменных, захваченных по значению:

```cpp
#include <iostream>

int main() {
```

```
    int ammo{ 10 };

    auto shoot{
        // Добавляем ключевое слово mutable после списка параметров
        [ammo]() mutable {
            // Теперь нам разрешено изменять значение переменной ammo
            --ammo;

            std::cout << "Pew! " << ammo << " shot(s) left.\n";
        }
    };

    shoot();
    shoot();

    std::cout << ammo << " shot(s) left\n";

    return 0;
}
```

Хотя теперь этот код и скомпилируется, но в нем все еще есть логическая ошибка. Какая именно? 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.

Обратите внимание, что значение переменной ammo сохраняется, несмотря на вызовы лямбды.

Подобно тому, как функции могут изменять значения аргументов, передаваемых им по ссылке, мы также можем захватывать переменные по ссылке, чтобы позволить нашей лямбде влиять на значения аргументов.

Чтобы захватить переменную по ссылке, мы должны добавить знак амперсанда (&) к имени переменной, которую хотим 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). Если вы предпочитаете передавать аргумент функции по ссылке (например, для типов, не являющихся базовыми), то вместо захвата по значению, предпочтительнее использовать захват по ссылке.

Вот вышеприведенный код, но уже с захватом переменной ammo по ссылке:

```
#include <iostream>

int main() {
    int ammo{ 10 };

    auto shoot{
        // Ключевое слово mutable теперь нам не потребуется
        [&ammo]() { // &ammo означает, что переменная ammo захватывается по ссылке
          // Изменения текущей переменной ammo приведут к изменению переменной ammo из
блока main()
          --ammo;

          std::cout << "Pew! " << ammo << " shot(s) left.\n";
        }
    };

    shoot();

    std::cout << ammo << " shot(s) left\n";
```

```
    return 0;
}
```

Мы можем захватить несколько переменных, разделив их запятыми. Мы также можем использовать как захват по значению, так и захват по ссылке.

Необходимость явно перечислять переменные для захвата иногда может быть несколько обременительной. Если вы изменяете свою лямбду, то вы можете забыть добавить или удалить захватываемые переменные. К счастью, есть возможность заручиться помощью компилятора для автоматической генерации списка переменных, которые нам нужно захватить.

Захват по умолчанию захватывает все переменные, упомянутые в лямбде. Если используется захват по умолчанию, то переменные, не упомянутые в лямбде, не будут захвачены.

Чтобы захватить все задействованные переменные по значению, используйте = в качестве значения для захвата. Чтобы захватить все задействованные переменные по ссылке, используйте & в качестве значения для захвата.

В начало

Реализация слушателей

Часто программам нужно как-то реагировать на взаимодействия с ними, особенно это касается графических приложений. Механизмов реагирования на воздействие достаточно много, мы рассмотрим самый простой из них: слушатели. Что такое слушатель? Представим, что Вы агент, по сигналу фонариком из соседнего окна Вы должны отправить СМС сообщение по определённому номеру. Как только вы видите сигнал, Вы сразу отправляете сообщение. А теперь представьте, что отправкой сообщения занимается хитроумное устройство, а Ваша задача просто вовремя нажать кнопку на нём. Теперь при каждом сигнале фонариком Вы нажимаете кнопку и отправляется СМС сообщение. А теперь представим, что мы вообще не знаем, что делает эта кнопка, у нас задание – нажать на неё при свете фонарика. Таким образом в это хитроумное устройство можно положить всё, что угодно и при нажатии кнопки, оно сработает. В таком случаи мы являемся неким исполнителем, слушателем, как только нам поступает сигнал мы нажимаем на нашу знакомую кнопку, на которую нам сказали нажимать и не важно, что она будет делать. Теперь перенесём наш пример из жизни в код. Нам нужно написать некую функцию, которая будет запускать реализацию какого – то интерфейса, который ей знаком, а этот интерфейс будет запускать другую функцию и выполнять различные полезные действия для нас. Таким образом, функция станет неким триггером, который при срабатывании запускает знакомую функцию. Давайте реализуем абстрактный класс слушателя, класс кнопки и полезную функцию:

```
class OnClickListener {
public:

	typedef void(*Func)();

	OnClickListener(Func f) {
		m_f = f;
	}
```

```
	void operator()() {
		m_f();
	}

private:
	Func m_f{};
};

class Button {
public:
	Button(char* name) : m_name(name) {}

	void SetOnClickListener(OnClickListener* on_click_listener) {
		m_listener = on_click_listener;
	}

	void click() {
		(*m_listener)();
	}

private:
	char* m_name{};
	OnClickListener* m_listener{};
};

void NeedFunc() {
	cout << "Click!" << endl;
}

int main() {

	Button btn_1("btn 1");

	btn_1.SetOnClickListener(new OnClickListener(NeedFunc));

	btn_1.click();

	return 0;
}
```

Благодаря классу слушателю мы можем передавать совершенно любую функцию в него и по общему интерфейсу вызывать её реализацию в любой кнопке. Функции, которые передаются в запускаемые функции называются – функциями обратного вызова. Частое использование таких функций – это реализации необходимого поведения программы в ответ на действия пользователя.

В начало

Общее задание

В следующем цикле Л/Р по высокоуровневому программированию вам будет предложено реализовать полноценную программу по управлению, каталогизации и обработки информации. Каждая программа будет представлять из себя логически законченную платформу по типу: «Автоматизированная система управления» и содержать в себе следующий функционал:

- Обработка базы данных
  - Создание / удаление / редактирования записей
  - Сортировка и фильтрация записей
- Управление пользователями
  - Создание
  - Удаление
  - Авторизация

Каждая программа будет снабжена интерактивным меню пользователя. В этой Л/Р вам будет предложено внедрить в свою программу обобщённое программирование, а именно – шаблоны. Переписав, тем самым, повторяющиеся куски кода, например – сортировка и фильтрация.

Задание для всех вариантов

Как вы помните, на прошлой Л/Р мы закончили написание вашей программы. Теперь она работает и выполняет свои функции. Сейчас ваша задача заключается в анализе кода программы и поиска мест потенциальных ошибок. Как правило, такие места находятся в логике получения внешних данных (например, из хранилища, или ввод пользователя). Протестируйте эти места с различными сценариями работы и выявите ошибки, с которыми падает ваша программа. **В Вашей программе должны быть отработаны 3-5 исключений.**

Контрольные вопросы

1. Какие типы ошибок существуют?
2. Как выполняется обработка ошибок?
3. Дайте определение синтаксическим ошибкам.
4. Что такое семантическая ошибка?
5. За что отвечает стэйтмент assert?
6. Что такое исключение?
7. Как выбросить исключение?
8. Каким образом происходит обработка исключений?
9. Как он реализуется?
10. Какие существуют типы исключений?
11. Как создать своё исключение?
12. Какие функции по созданию исключений предоставляет стандартная библиотека C++?
13. Какую роль наследование играет в исключениях?
14. Что такое раскрутка стека?
15. Что такое функция высшего порядка?
16. Что такое декоратор, для чего он нужен?
17. Что такое функторы?
18. Что такое лямбда функции?
19. Из чего состоит шаблон лямбда функции?
20. Как использовать такие функции?
21. Какой тип имеет лямбда функция?
22. Что такое замыкание (захваты)?
23. Что такое функция обратного вызова (callback)?
24. Что такое слушатель и как его реализовать?

Список литературы

1. Курс лекций доцента кафедры ФН1-КФ Пчелинцевой Н.И.
2. Программирование на языке высокого уровня C/C++ [Электронный ресурс]: конспект лекций / – Электрон. текстовые данные. – М.: Московский государственный строительный университет, Ай Пи Эр Медиа, ЭБС АСВ, 2016. – 140 с. – Режим доступа: http://www.iprbookshop.ru/48037.

В начало