Лабораторная работа №6

по курсу «Высокоуровневое программирование» (2 семестр)

«Исключения и обработка исключений»

Оглавление

Цель: приобретение практических навыков и знаний по обработке исключительных ситуаций в логике и синтаксисе программы.

Задачи:

1. Познакомиться с типами ошибок;
2. Научиться обрабатывать ошибки при компиляции;
3. Познакомиться с концепцией исключений;
4. Научиться вызывать и обрабатывать исключения.

Содержание отчёта:

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

Основные теоретические сведения

Обработка ошибок

Ошибками в языках программирования называются все ситуации, которые не позволят получить успешный результат. Они бывают двух видов:

- Синтаксические ошибки – ошибки, которые появляются во время компиляции при неправильном использовании средств языка.
- Семантические ошибки возникают, когда технически код работает верно, но логический результат не совпадает с ожидаемым.

Безопасное программирование – методика разработки программ, которая подразумевает анализ областей, в которых могут быть допущены ложные предположения, и написание кода, который обнаруживает и обрабатывает нарушения, чтобы свести к минимуму риск возникновения сбоя 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 понадобится функция: *exit*(), которая в качестве параметров принимает код и передает его в операционную систему. Чтобы использовать эту функцию, необходимо подключить заголовочный файл: «*cstdlib*».

Стэйтмент assert

Стэйтмент *assert* (оператор проверочного утверждения) в языке C++ — это макрос препроцессора, который обрабатывает условное выражение во

время выполнения. Если условное выражение истинно, то стэйтмент *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. *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 завершится, а в консоли отобразится сообщение, переданное вторым параметром.

Обратите внимание, функция *exit*() и *assert* (если он срабатывает) немедленно прекращают выполнение программы, без возможности выполнить безопасное завершение программы (например, закрыть файл или базу данных).

В C++11 добавили еще один тип assert – 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 как сигнал о возникновении исключения или ошибки (аналогично тому, как свистит арбитр). Сигнал об исключении, называется генерацией или выбрасыванием исключения.

После ключевого слова *throw* 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’t take square root of negative number"; // генерация исключения типа const char* (строка C-style)
	double dX = 0;
	throw dX; // генерация исключения типа double
	throw MyException("Fatal Error"); // генерация исключения с использованием объекта класса MyException
```

Как только просвистел арбитр, игроки останавливаются, и игра временно прекращается. Обычный ход игры нарушен.

В языке C++ ключевое слово *try* используется для определения блока стэйтментов. Блок *try* действует как наблюдатель в поисках исключений, которые были выброшены операторами в этом же блоке *try*, например:

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

int main() {
	try {
		// Внутри находится код, который может вызвать исключение
```

```
            getException("My exce");
        }
        return 0;
}
```

Обратите внимание, блок *try* не определяет, как мы будем обрабатывать исключение. Он только сообщает о его наличии.

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.

Обработка исключений осуществляется блоками *catch*, которые обрабатывают исключения определенного типа данных. Пример блока *catch*, который обрабатывает 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 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 завершено выполнение блока *catch*, оно считается обработанным, и выполнение программы возобновляется.

Параметры *catch* работают так же, как и параметры функции. Исключения фундаментальных типов данных могут быть пойманы по значению (параметром блока *catch* является значение), но исключения нефундаментальных типов данных должны быть пойманы по константной ссылке (параметром блока *catch* является константная ссылка), дабы избежать ненужного копирования.

Непойманные исключения

Функции могут генерировать исключения любого типа данных, и, если исключение не поймано, это приведет к раскручиванию стека и потенциальному завершению выполнения программы.

К счастью, язык C++ предоставляет 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.

```
try {
	getException("My exce");
}
catch (...) {
	cerr << "Error! Some error!" << endl;
}
```

Классы-исключения

Одной из основных проблем использования фундаментальных типов данных в качестве типов исключений (например, *int*) является то, что они являются неоднозначными. Так же, как и неясность того, какая именно из функций в блоке *try* вызвала исключение. Одним из способов решения этой проблемы является использование классов-исключений.

Класс-Исключение – это класс, который предназначен для того, чтобы выбрасываться 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{};
};
```

Применение:

```
void getException(const char* my_exce) {
	throw MyException(my_exce);
}
int main() {
	try {
		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 отдельным образом.

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.

Наследование исключений

Так как мы можем выбрасывать объекты классов в качестве исключений, а классы могут быть наследниками других классов, значит обработчики могут обрабатывать исключения не только одного определенного класса, но и исключения дочерних ему классов. Обработчики исключений дочерних классов должны находиться перед обработчиками родительских, потому как проверка catch проходит последовательно до первого успешного результата. 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++.

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.

```
#include <iostream>
#include <exception> // для std::exception
#include <string>
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. Однако, вы можете генерировать исключения других классов из Стандартной библиотеки C++ или создать свои собственные классы-исключения, дочерние классу *std*::*exception*, и переопределить виртуальный константный метод *what*() для вывода сообщения об ошибке.

std::*runtime_error* (находится в заголовочном файле *stdexcept*) является популярным выбором, так как имеет общее имя, а его 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;
}
```

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*.

Задание

Проанализируйте код и найдите места в программе, где возможны исключения (как минимум 3 вида ошибок), например: некорректный ввод пользователя, логически недопустимые значения полей или выход за границы допустимых значений типов, несовпадение типов, передаваемых в параметрах функции или математические ошибки, как возможное деление на ноль и т. д. Используя механизм обработки исключений ликвидируйте такие места сделав код безопасным.

Протестируйте свою программу с различными категориями данными. Исправьте встречающиеся ошибки.