Язык программирования C++

Обработка исключений.

Преподаватели:
Пысин Максим Дмитриевич, ассистент кафедры ИКТ
Краснов Дмитрий Олегович, аспирант кафедры ИКТ
Лобанов Алексей Владимирович, аспирант кафедры ИКТ
Крашенинников Роман Сергеевич, аспирант кафедры ИКТ


ВЬЕТНАМ?
C++

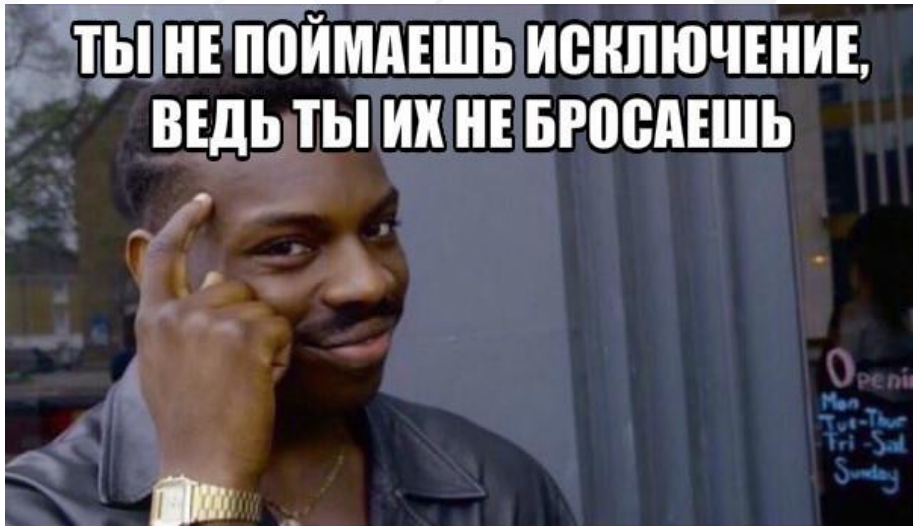
Обработка исключений

Обработка исключительных ситуаций (англ. exception handling) — механизм языков программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма.

Во время выполнения программы могут возникать ситуации, когда состояние внешних данных, устройств ввода-вывода или компьютерной системы в целом делает дальнейшие вычисления в соответствии с базовым алгоритмом невозможными или бессмысленными.

Самой **последней инстанцией** в процессе обработки исключений является **операционная система**. Когда программа завершает своё выполнение с необработанным исключением, то операционная система обычно уведомляет нас о том, что произошла ошибка необработанного исключения. Как она это сделает — зависит от каждой отдельно:

- либо выведет сообщение об ошибке;
- либо откроет диалоговое окно с ошибкой;
- либо просто вызовет сбой в работе.



Примеры исключений

- Целочисленное деление на ноль. Конечного результата у данной операции быть не может, поэтому ни дальнейшие вычисления, ни попытка использования результата деления не приведут к решению задачи.
- Ошибка при попытке считать данные с внешнего устройства. Если данные не удаётся получить, любые дальнейшие запланированные операции с ними бессмысленны.
- Исчерпание доступной памяти. Если в какой-то момент система оказывается не в состоянии выделить достаточный для прикладной программы объём оперативной памяти, программа не сможет работать нормально.
- Появление сигнала аварийного отключения электропитания системы. Прикладную задачу, по всей видимости, решить не удастся, в лучшем случае (при наличии какого-то резерва питания) прикладная программа может позаботиться о сохранении данных.
- Обращение к несуществующей памяти, т.е. выход за границы массива или обращение по пустому указателю.
- Ошибка работы с базой данных, неправильное SQL выражение или ошибка исполнения выражения.
- Ошибка при работе с сетью, неожиданный обрыв соединения, неправильные данные для подключения.
- Ошибка преобразования указателя от указателя на пустоту к классу не соответсвующему реально хранимому объекту.

Типы исключений

Исключительные ситуации, возникающие при работе программы, можно разделить на два основных типа: синхронные и асинхронные, принципы реакции на которые существенно различаются.

Синхронные исключения могут возникнуть только в определённых, заранее известных точках программы. Так, ошибка чтения файла или коммуникационного канала, нехватка памяти — типичные синхронные исключения, так как возникают они только в операции чтения или в операции выделения памяти соответственно.

Асинхронные исключения могут возникать в любой момент времени и не зависят от того, какую конкретно инструкцию программы выполняет система. Типичные примеры таких исключений: аварийный отказ питания или поступление новых данных.

Некоторые типы исключений могут быть отнесены как к синхронным, так и к асинхронным. Например, инструкция деления на ноль формально должна приводить к синхронному исключению, так как логически оно возникает именно при выполнении данной команды, но на некоторых платформах за счёт глубокой конвейеризации исключение может фактически оказаться асинхронным.

Типы обработки исключений

Существует два принципиально разных механизма функционирования обработчиков исключений.

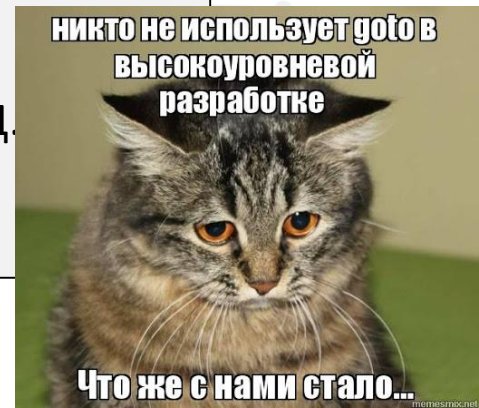
Обработка с возвратом подразумевает, что обработчик исключения ликвидирует возникшую проблему и приводит программу в состояние, когда она может работать дальше по основному алгоритму. В этом случае после того, как выполнится код обработчика, управление передаётся обратно в ту точку программы, где возникла исключительная ситуация и выполнение программы продолжается. Обработка с возвратом типична для обработчиков асинхронных исключений, для обработки синхронных исключений она малопригодна.

Обработка без возврата заключается в том, что после выполнения кода обработчика исключения управление передаётся в некоторое, заранее заданное место программы, и с него продолжается исполнение. То есть, фактически, при возникновении исключения команда, во время работы которой оно возникло, заменяется на безусловный переход к заданному оператору.

Неструктурная обработка исключений

Неструктурная обработка исключений реализуется в виде механизма регистрации функций или команд-обработчиков для каждого возможного типа исключения. Вызов регистрации «привязывает» обработчик к определённому исключению, вызов разрегистрации — отменяет эту «привязку». Если исключение происходит, выполнение основного кода программы немедленно прерывается и начинается выполнение обработчика. По завершении обработчика управление передаётся либо в некоторую наперёд заданную точку программы, либо обратно в точку возникновения исключения.

```
УстановитьОбработчик(ОшибкаБД, ПерейтиНа ОшБД)
   // На исключение "ОшибкаБД" установлен обработчик - команда "ПерейтиНа ОшБД"
   // Здесь находятся операторы работы с БД
   ПерейтиНа СнятьОшБД // Команда безусловного перехода - обход обработчика исключений
ОшБД:  // метка - сюда произойдёт переход в случае ошибки БД по установленному обработчику
   // Обработчик исключения БД
СнятьОшБД:
   // метка - сюда произойдёт переход, если контролируемый код выполнится без ошибки БД.
   СнятьОбработчик(ОшибкаБД)
   // Обработчик снят
```



Структурная обработка исключений

Структурная обработка исключений требует обязательной поддержки со стороны языка программирования — наличия специальных синтаксических конструкций. Такая конструкция содержит блок контролируемого кода и обработчик (обработчики) исключений.
Блоки обработки исключений могут многократно входить друг в друга, как явно (текстуально), так и неявно. Если ни один из обработчиков в текущем блоке не может обработать исключение, то выполнение данного блока немедленно завершается, и управление передаётся на ближайший подходящий обработчик более высокого уровня иерархии

```
НачалоБлока
        // Контролируемый код
        если (условие) то СоздатьИсключение Исключение2
Обработчик Исключение1
        // Код обработчика для Исключения1
Обработчик Исключение2
        // Код обработчика для Исключения2
ОбработчикНеобработанных
        // Код обработки ранее не обработанных исключений
КонецБлока
```

Блок с гарантией завершения

Помимо блоков контролируемого кода для обработки исключений, языки программирования могут поддерживать **блоки с гарантированным завершением**. Их использование оказывается удобным тогда, когда в некотором блоке кода, независимо от того, произошли ли какие-то ошибки, необходимо перед его завершением выполнить определённые действия. Простейший пример: если в процедуре динамически создаётся какой-то локальный объект в памяти, то перед выходом из этой процедуры объект должен быть уничтожен (чтобы избежать утечки памяти), независимо от того, произошли после его создания ошибки или нет.

Принципиальное отличие блока с гарантированным завершением от обработки — то, что он не обрабатывает исключение, а лишь гарантирует выполнение определённого набора операций перед тем, как включится механизм обработки.

```
НачалоБлока
  ... // Основной код
Завершение
  ... // Код завершения
КонецБлока
```

Обработка исключений C++(TRY CATCH)

В C++ мы используем ключевое слово try для определения блока в котором по нашему предположению могут возникать ошибки. Блок try действует как наблюдатель, в поисках исключений, которые были выброшены каким-либо из операторов в этом же блоке try.
Обратите внимание, блок try не определяет, КАК мы будем обрабатывать исключение.
Фактически, обработка исключений — это работа блока(ов) catch. Ключевое слово catch используется для определения блока кода (так называемого «блока catch»), который обрабатывает исключения определённого типа данных.

```
try {
    // код который может сгенерировать исключение
    throw -1;
}
catch (int e){
    // код который реагирует на появившееся исключение
    cerr << "Caught exeption." << endl;
}
```

Блоки try и catch работают вместе. Блок try обнаруживает любые исключения, которые были выброшены в нём, 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).

Генерация исключений(THROW)

Параметры catch работают так же, как и параметры функции, причём 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 C++ оператор throw используется, чтобы сигнализировать о возникновении исключения или ошибки. Оповещение о том, что произошла нештатная ситуация в работе кода, называется генерацией исключения (или ещё «выбрасыванием исключения»).


```
throw -1; // генерация исключения типа int
throw ENUM_INVALID_INDEX; // генерация исключения типа enum
throw "Невозможно извлеч корень из отрицательного числа"; // генерация
 исключения типа const char* (строка C-style)
throw dX; // генерация исключения типа double (переменная типа double,
 которая была определена ранее)
throw MyException("Критическая ошибка"); // генерация исключения с исп
ользованием объекта класса MyException
```

Пример исключений

```
int main(){
    int a = 0;
    int b = 1;
    int c = 1;
    cout << "Start with divide by zero." << endl;
    try {
        cout << "Before exception." << endl;
        c = divideBy(b, a);
        cout << "After exception." << endl;
    }
    catch (runtime_error e) {
        cout << e.what() << endl;
        c = 0;
    }
    catch(...){
        cout << "Any exception." << endl;
    }
    cout << "End with divide by zero. c value is " << c << endl;
}
```

```
int divideBy(int what, int by){
    if(by == 0)
        throw logic_error("Division by zero");
    return what / by;
}
```

```
Start with divide by zero.
Before exception.
Any exception.
End with divide by zero. c value is 1
```

```
int divideBy(int what, int by){
    if(by == 0)
        throw runtime_error("Division by zero");
    return what / by;
}
```

```
Start with divide by zero.
Before exception.
Division by zero
End with divide by zero. c value is 0
```

Последовательность обработки

При выбрасывании исключения (оператор `throw`), точка выполнения программы немедленно переходит к ближайшему блоку try. Если какой-либо из обработчиков catch, прикреплённых 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 в следующий блок try. Если до конца программы не найдены соответствующие обработчики catch, то программа завершает своё выполнение с ошибкой исключения.

Обратите внимание, компилятор не выполняет неявные преобразования при сопоставлении исключений с блоками catch! Например, исключение типа char не будет обрабатываться блоком catch типа int, исключение типа int, в свою очередь, не будет обрабатываться блоком catch типа float.

Так же существует обработчик исключений catch-all который отлавливает исключения всех типов и видов, но не может использовать значения этих исключений. Обработчик catch-all должен находится последним в цепочке блоков catch.

```
catch(...){
    cout << "Any exception." << endl;
}
```

Некоторые особенности

Если исключение направлено в блок catch, то оно считается «обработанным», даже если блок catch пуст. Однако, как правило, вы захотите, чтобы ваши блоки catch делали что-то полезное. Есть три распространённые вещи, которые выполняют блоки catch, когда они поймали исключение:

- 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 место вызова функции в которой произошло исключение и было обработано.
- В-третьих, блок catch может сгенерировать другое исключения. Поскольку блок catch не находится внутри блока try, то новое сгенерированное исключение будет обрабатываться следующим в иерархии блоком try.

Блок try ловит исключения не только внутри себя, но и внутри функций, которые вызываются в блоке try.

Непосредственно тот кто вызывает функцию и в которой выбрасывается исключение, не обязан обрабатывать это исключение, если он этого не хочет.

Когда исключение обработано, выполнение кода продолжается как обычно, начиная с конца блока catch

Пример исключений

```
void last(){ // вызывается функцией three()
    cout << "Функция last" << endl;
    cout << "Выбрасываем исключение типа int в функции last" << endl;
    throw -1;
    cout << "Конец last" << endl;
}

void three(){ // вызывается функцией two()
    cout << "Функция three << endl";
    last();
    cout << "Конец three" << endl;
}

void two(){ // вызывается функцией one()
    cout << "Функция two" << endl;
    try{
        three();
    }
    catch(double){
         cerr << "Поймано исключение типа double в функции two" << endl;
    }
    cout << "Конец two" << endl;
}
```

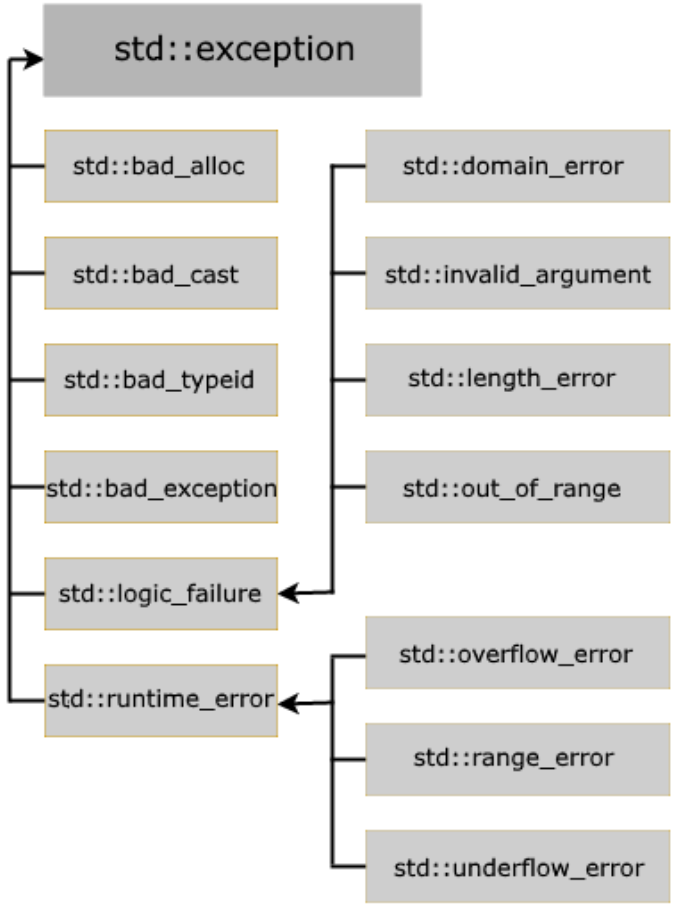
```
void one(){ // вызывается функцией main()
    cout << "Функция one" << endl;
    try{
        two();
    }
    catch (int){
         cerr << "Поймано исключение типа int в функции one" << endl;
    }
    catch (double){
         cerr << "Поймано исключение типа double в функции one" << endl;
    }
    cout << "Конец one" << endl;
}
```

Пример исключений

```
int main(){
    cout << "Начано main" << endl;
    try{
        one();
    }
    catch (int){
        cerr << "Поймано исключение типа int в функции main" << endl;
    }
    cout << "Конец main" << endl;
    return 0;
}
```

```
Начано main
Функция one
Функция two
Функция three << endlФункция last
Выбрасываем исключение типа int в функции
last
Поймано исключение типа int в функции one
Конец one
Конец main
```

Класс исключение



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

Обратите внимание, в обработчиках исключений объекты класса-исключения принимать нужно **по ссылке**, нежели по значению.

Обработчики могут обрабатывать исключения не только одного определённого класса, но и исключения дочерних ему классов.

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

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.

Пример

```
class DivisionByZeroException: public exception{
    private:
        const char* msg = "Division by 0";
        int code = -10;
    public:
        void printSelf(){
            cout << what() << endl;
        }
        int getCode(){
            return code;
        }
        const char* what() {
            return msg;
        }
};


int divideBy(int what, int by){
    if(by == 0)
        throw DivisionByZeroException();
    return what / by;
}
```

Пример

```
int main(){
    int a = 0;
    int b = 1;
    int c = 1;
    cout << "Start with divide by zero." << endl;
    try {
        cout << "Before exception." << endl;
        c = divideBy(b, a);
        cout << "After exception." << endl;
    }
    catch(exception e) {
        cout << e.what() << endl;
    }
    catch (DivisionByZeroException e) {
        e.printSelf();
        c = e.getCode();
    }
    catch(...){
        cout << "Any exception." << endl;
    }
    cout << "End with divide by zero. c value is " << c << endl;
}
```

```
Start with divide by zero.
Before exception.
std::exception
End with divide by zero. c value is 1
```

Пример

```
int main(){
    int a = 0;
    int b = 1;
    int c = 1;
    cout << "Start with divide by zero." << endl;
    try {
        cout << "Before exception." << endl;
        c = divideBy(b, a);
        cout << "After exception." << endl;
    }
    catch (DivisionByZeroException e) {
        e.printSelf();
        c = e.getCode();
    }
    catch(exception e) {
        cout << e.what() << endl;
    }
    catch(...){
        cout << "Any exception." << endl;
    }
    cout << "End with divide by zero. c value is " << c << endl;
}
```

```
Start with divide by zero.
Before exception.
Division by 0
End with divide by zero. c value is -10
```

Производительность исключений

Исключения имеют свою небольшую цену производительности. Они увеличивают размер вашего исполняемого файла и также могут заставить его выполняться медленнее из-за дополнительной проверки, которая должна быть выполнена. Тем не менее, основное снижение производительности происходит при выбрасывании исключения. В этот момент стек начинает раскручиваться и выполняется поиск соответствующего обработчика исключений, что само по себе является относительно затратной операцией.

Примечание: Некоторые современные компьютерные архитектуры поддерживают модель исключений «**Исключения с нулевой стоимостью**» (zero-cost исключения). Исключения с нулевой стоимостью, если они поддерживаются, **не требуют дополнительных затрат при выполнении программы** в случае отсутствия ошибок (именно в этом случае мы больше всего заботимся о производительности). Тем не менее, исключения с нулевой стоимостью **потребляют ещё больше ресурсов в случае обнаружения исключения**.

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

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

ASSERT

Исключения тесно связаны с тестированием программы после любых изменений вносимых в программу. Для модульного тестирования в C++ существуют такие Фреймворки как:

- Google Test – библиотека для модульного тестирования от Google
- Boost Test – библиотека для модульного тестирования от создателей библиотеки boost и входящая в нее.
- QT Test – библиотека для модульного тестирования от создателей библиотеки QT и входящая в нее.
- Doctest – простая и легковесная библиотека для написания модульных тестов в небольших проектах
- Catch2 – простая и легковесная библиотека для написания модульных тестов в небольших проектах

Кроме описанных выше библиотек существует встроенный в стандартную библиотеку способ тестирования кода.

Один это макрос ASSERT(), который наследуется из языка си и импортируется как библиотека <cassert>. Этот макрос выведет ошибку, если выражение между круглыми скобками будет равно 0.

```
assertm(2+2==5, "There are five lights"); // assertion fails
std::cout << "Execution continues past the last assert\n"; // No
```

Альтернативой является функция времени компиляции static_assert(bool-constexpr), которая так же, как и макрос выведет в консоль ошибку, если выражение в скобках возвращает false.

Спасибо за внимание