**Обработка ошибок С++**

**Введение:**

Абсолютно все программисты сталкиваются с разным родом ошибками при разработке программ. К сожалению, они просто неизбежны, хотя в окончательном варианте программы не должно быть ошибок, по крайней мере неприемлемых.

Существует множество способов классификации ошибок, я же выделю четыре основные группы:

1. Логические - ошибки программистов при написании кода.
2. Ошибки ввода данных - пользовательские.
3. Злонамеренные - ввод некорректных данных.
4. Системные - что-то "внешнее"

Хотелось бы сказать, что задача программиста - устранить все ошибки. Это звучит довольно соблазнительно, но, к сожалению, зачастую эта цель бывает недостижима. Приведу пример. Допустим, при выполнении нашей программы мы выдернем шнур из розетки, или же наш ноутбук разрядится, то следует ли это рассматривать как ошибку и самое главное, следует ли предусмотреть ее обработку? Во многих случаях ответ будет отрицательным. Нашей целью будет создание программы, отлов и обработка ошибок которой будет осуществляться в основном самой же программой. Для этого рассмотрим методы обработки ошибок:

1. **Не обрабатывать ошибки**.

Действительно, при написании небольших программ (например тестовых), когда мы уверены в правильной работе программы, обработка ошибок может занять лишнее время, что только пойдет программисту во вред.

1. **Коды возврата**.

Рассмотрим пример (*1\_1.cpp*): Функция *int Find\_letter\_pos*  возвращает позицию заданной переменной типа char в (в нашем случае буквы) в строке . Мы пробегаем по строке с помощью цикла и если не находим нужное нам соответствие, возвращаем значение -1. Главным преимуществом этого подхода является его простота. Однако есть ряд недостатков, которые могут быстро проявиться в нетривиальных случаях.

Во-первых, возвращаемые значения не всегда понятны. Если функция возвращает -1, обозначает ли это какую-то специфическую ошибку или это корректное возвращаемое значение? Часто это бывает трудно понять, не видя перед глазами код самой функции.

Во-вторых, если программа большая, коды возврата нужно постоянно проверять, что может быть достаточно неудобным.

В-третьих коды возврата не работают с конструкторами классов(*1\_2.cpp*). Что произойдет, если мы создадим объект, а внутри конструктора случится что-то катастрофическое? Конструкторы не могут использовать оператор *return* для возврата индикатора состояния, а передача по ссылке может причинить массу неудобств, и её нужно явно проверять. Кроме того, даже если мы это сделаем, объект все равно создастся, и лечить мы уже будем последствия (либо обрабатывать, либо удалять).

Подводя итог, основная проблема с кодами возврата заключается в том, что они плотно связаны с общим потоком выполнения кода, а это, в свою очередь, ограничивает наши возможности.

1. **Поток вывода сообщений об ошибках std::cerr**

Задача: написать функцию , принимающую в себя указатель C-style строки и выводящую саму строку(*2.cpp*) Казалось бы, абсолютно тривиальная задача, что же может пойти не так? Например , если в функцию будет передан нулевой указатель , то поведение программы определить будет сложно. Данная задача может быть подзадачей большой программы , и тогда отследить и обработать ошибку может быть крайне трудно.

Для этого в С++ есть std::***cerr*** — это объект вывода (как и cout), который находится в заголовочном файле iostream и выводит сообщения об ошибках в консоль (как и cout), но только эти сообщения можно еще и перенаправить в отдельный файл с ошибками. Т.е. основное отличие cerr от cout заключается в том, что cerr целенаправленно используется для вывода сообщений об ошибках, тогда как cout — для вывода всего остального. В нашем примере *(2. cpp)* мы не только пропускаем код, который напрямую зависит от правильности предположения что в функцию не предана нулевая ссылка, но также регистрируем ошибку, чтобы пользователь мог позже определить, почему программа выполняется не так, как нужно.

1. **Стейтменты assert и static\_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**](https://ravesli.com/urok-26-otladka-programm-stepping-i-breakpoints/).

Рассмотрим пример (*3\_1.cpp*). Допустим у нас есть структура Triangle, для которой все стороны (параметры) изначально равны 1. Нужно определить функцию , меняющие стороны треугольника. В (*3\_1.срр)* рассмотрен пример работы assert для этой задачи.

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. Поскольку static\_assert обрабатывается компилятором, то условная часть static\_assert также должна обрабатываться во время компиляции. Поскольку static\_assert не обрабатывается во время выполнения программы, то стейтменты static\_assert могут быть размещены в любом месте кода (даже в глобальном пространстве). Пример работы static\_assert рассмотрен в *(3\_2.сpp)*

1. **Исключения.**

Возвращаясь к главной проблеме кодов возврата, обработка исключений как раз и обеспечивает механизм, позволяющий отделить обработку ошибок или других исключительных обстоятельств от общего потока выполнения кода. Это предоставляет больше свободы в конкретных ситуациях, уменьшая при этом беспорядок, который вызывают коды возврата. Исключения в языке C++ реализованы с помощью 3-х ключевых слов, которые работают в связке друг с другом: **throw, try**и**catch**. Важно, что исключения обрабатываются немедленно! *(4\_1.сpp).*

Рассмотрим исключения подробнее.

**Генерация исключений**

В жизни часто бывает удобно пользоваться какими-то сигналами, для того, чтобы отметить, что произошли какие-то события. Например при игре в футбол , если происходит фол, то арбитр свистит и останавливает игру (предположим что счетчик времени игры тоже останавливается). Команда пробивает штрафной удар , и игра продолжается.

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 (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.

**Поиск исключений**

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

В языке C++ используется **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.

**Обработка исключений**

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.

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**](https://ravesli.com/urok-97-peredacha-argumentov-po-znacheniyu/) (параметром блока catch является значение), но исключения нефундаментальных типов данных должны быть пойманы [**по константной ссылке**](https://ravesli.com/urok-98-peredacha-argumentov-po-ssylke/) (параметром блока catch является константная ссылка), дабы избежать ненужного копирования.

**Блоки catch**

Если исключение направлено в блок catch, то оно считается «обработанным», даже если блок catch пуст. Однако, как правило, мы хотим, чтобы наши блоки catch делали что-то полезное. Есть три распространенные 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 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.

Работа throw, try и catch продемонстрирована в *(4\_2.сpp).*

**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. *(4\_3. cpp).* В этом примере, если мы поймаем исключение типа int, что оно нам сообщит? Был ли передаваемый index недопустим? Может оператор + вызвал целочисленное **переполнение** или может **оператор new** не сработал из-за нехватки памяти?

**Кратко, главная информация об исключениях**

   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.