Обработка ошибок C++

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:

0. **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*):

```
#include <iostream>
using namespace std;

int Find_letter_pos(const char* str,
char c)
{
  for (size_t i = 0; size < strlen
(str); ++i)
  {
    if (str[i] == c)
      return pos;
  }
  return -1; // ничего не говорит
}
```

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*).

```
#include <iostream>
using namespace std;

struct Date_of_birth
{
  int day;
  int month;
  int year;
  Date_of_birth(int a, int b, int c)
  {
    if (( a < 1) || (a > 31) || ( b <
1) || (b > 12) || (c > 2021)) //
 понятно, что не все случаи, но для при
мера хватит
    {return -1;} //
 не-а, это конструктор, никаких return!
    day = a;
    month = b;
    year = c;
  }
}
```

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.

2. Поток вывода сообщений об ошибках std::cerr

Задача: написать функцию , принимающую в себя указатель C-style строки и выводящую саму строку(*2.cpp*)

```
#include <iostream>
using namespace std;

void print(const char* str)
{
  if (str) //
 проверяем строку на нулевой указатель
    cout << str;
  else
  {
    cerr << "null parameter"; //
 сообщение об ошибке
  }
}
```

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 C++ 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.

3. 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.

Рассмотрим пример (*3_1.cpp*).

```cpp
#include <iostream>
#include <cassert>
using namespace std;

struct Triangle
{
  double a = 1;
  double b = 1;
  double c = 1;
};
```

```cpp
void set_Triangle(Triangle* tr, double x, double y, double z)
{
  assert ((x > 0) && (y > 0) && (z > 0)); // проверка 1
  assert ((2 * max(x,y,z) < x + y + z) && ("Can't build the triangle because of the rule about the max side")); // проверка 2
  //по умолчанию C-style строка всегда true, а false && true = false. Удобный способ вывести сообщение
  tr->a = x;
  tr->b = y;
  tr->c = z;
}
```

Допустим у нас есть структура Triangle, для которой все стороны (параметры) изначально равны 1. Нужно определить функцию , меняющие стороны треугольника. В (*3_1.cpp)* 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.cpp)*

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

constexpr int calculate (const int x)
{
  return x * sizeof(int);
}

int main()
{
  constexp int sz = calculate(20);
  static_assert (sz < 10 , "Size is too big");

        return 0;
}
```

4. 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.cpp).*

```
#include <iostream>
using namespace std;

int main()
{
  try
  {
    throw -1;
    cout << "That never prints";
  }
  catch (int a)
  {
    cerr << "we caught a walue = "
 << a;
  }
   return 0;
}
```

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 возобновляетсся.

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 нефундаментальных типов данных должны быть пойманы по константной ссылке (параметром блока catch является константная ссылка), дабы избежать ненужного копирования. Также для catch работает неявное преобразование типов c++.

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.cpp).*

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    cout << "Enter a number: ";
    double a;
    cin >> a;

    try // ищем исключения внутри этого блока и отправляем их в соответствующий обработчик catch
    {
        // Если пользователь ввел отрицательное число, то выбрасывается исключение
        if (a < 0.0)
            throw "Can not take sqrt of negative number"; // выбрасывается исключение типа const char*

        // Если пользователь ввел положительное число, то выполняется операция и выводится результат
        cout << "The sqrt of " << a << " is " << sqrt(a) << '\n';
    }
    catch (const char* exception) // обработчик исключений типа const char*
    {
        std::cerr << "Error: " << exception << '\n';
    }
}
```

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).*

```
// Используем перегрузку operator[] для ArrayInt

try
{
    int *value = new int (array[index1] + array[index2]);
}
catch (int value)
{
    // Какие исключения мы здесь ловим?
}
```

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?

Одним из способов решения этой проблемы является использование классов-исключений. Класс-Исключение — это обычный класс, который выбрасывается в качестве исключения. Создадим простой класс-исключение, который будет использоваться с нашим ArrayInt. *(4_4.cpp)*

```
#include <iostream>
#include <string>
using namespace std;

class ArrayException
{
private:
 string m_error;

public:
  ArrayException(string error)
    : m_error(error)
  {
  }

   const char* getError() { return m_error.c_str(); }
};

class ArrayInt
{
private:

  int m_data[4]; // ради сохранения простоты примера укажем значение 4 в качестве длины массива
public:
  ArrayInt() {}

  int getLength() { return 4; }

  int& operator[](const int index)
  {
    if (index < 0 || index >= getLength())
      throw ArrayException("Invalid index");

    return m_data[index];
  }

};

int main()
{
  ArrayInt array;

  try
  {
    int value = array[7];
  }
  catch (ArrayException &exception)
  {
    cerr << "An array exception occurred (" << exception.getError() << ")\n";
  }
}
```

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

Класс 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 exception мы можем настроить обработчик исключений типа exception, который будет ловить и обрабатывать как exception, так и все дочерние ему классы-исключения.

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.