Конспекты по C++
Руководитель: Иван Сорокин

Собрано 22 ноября 2017 г. в 23:12

Содержание

1 Exceptions

1.1 Введение

В ходе написания программ, часто возникают случаи, когда нам приходится отвлекаться от написания основной логики. Нам нужно учитывать граничные случаи и ошибки, которые могут возникнуть как во внутреннем коде нашей программы, так и во внешнем коде. Это может быть: нехватка динамической памяти, неправильный ввод пользователя, ошибка с файловой системой и т. д. Все это приходится продумывать, и иногда это может сильно увеличить количество кода и время его написания. Чтобы упростить обработку таких случаев существует паттерн исключений.

Исключение - это ситуация, которую МЫ сочли исключительной, и которую хотим отделить от основной логики программы. Естественно код обработки исключения мы тоже должны написать сами. Но компилятор помогает нам, генерировать дополнительную информации об исключении, отслеживать и подбирать необходимый код обработки в зависимости от исключения.

Пусть мы пишем функцию деления:

```
void div(int a, int b) {
    return a / b;
}
```

Пусть исключительная ситуация возникает тогда, когда $b = 0$. И мы хотим сообщить об этом пользователю нашей функции.

```
class Division_by_zero() {
    int dividend
    string message;
    Division_by_zero(int &dividend, string const &message) :
        dividend(dividend), message(message) { }
};
```

Мы объявили класс, объекты которого будут хранить в себе всю информации об исключении. Тип класса-исключения называется просто типом исключения.

```
void div(int a, int b) {
    if (b == 0) // возникает исключительное состояние
        throw Division_by_zero(b, "in function div(int, int)"); // генерируем
 ↪  исключение.
    return a / b;
}
```

Если возникает ситуация, когда нам передали в качестве делителя ноль, то мы создаем/генерируем/возбуждаем исключение. Это значит, что дальше не будет исполняться основная логика программы, пока не будет обработано исключение.

А так пользователь может обрабатывать исключение:

```
int main() {
    int n;
    cin >> n;
    try { // здесь указываем операторы, в которых мы хотим ловить исключения.
        for (int i = 0, a, b; i < n; ++i) {
            cin >> a >> b;
            cout << div(a, b);
        }
    } catch(Division_by_zero obj) { //здесь указываем тип исключения, которое мы
    ↪ хотим обработать
    // здесь обрабатываем исключение
        cout << obj.dividend << "by 0 " << obj.message();
    }
}
```

Как только возникает попытка деления на ноль. генерируется исключение, в коде `div(int, int)` оно не обрабатывается, поэтому выбрасывается во внешний код, где мы его ловим и выводим сообщение об ошибке.

Если исключения не возникает, то код отработает нужным образом: мы выведем результаты всех делений.

1.2 Описание конструкций

Теперь рассмотрим используемые здесь конструкции подробнее:

`try{}` – защищенный блок. Здесь мы пишем код, в котором мы хотим ловить и обрабатывать исключения.

`catch(){}` – блок перехвата исключений или блок обработки или обработчик. Здесь буду ловиться исключения, тип которых совпадает по определенным правилам с типом указанным в (), и обрабатывать инструкциями в {}

`throw` – оператор инициализации исключения. Он генерирует исключение. (Обычно говорят, "бросает" или "выбрасывает" исключение)

Теперь давай рассмотрим детали работы этого механизма.

Блок `try-catch` - реализует обработку исключений. И имеет общий вид:

```
try { /*операторы защищенного блока*/ }
// catch-блоки
catch() {/*код обработки*/}
```

```
...
catch() {/*код обработки*/}
```

То есть обработчиков может быть несколько, каждый обрабатывает свой тип исключений.
Общий вид catch-блока:

- `catch(Type) { /*обработчик исключения*/ }` Мы не используем генерируемый исключением объект.
- `catch(/*declaration exception_variable*/) { /*обработчик исключения*/ }`
- `catch(...) { /*обработчик исключения*/ }` Мы ловим исключение с любым типом, но не можем получить к нему доступ.

Что происходит, когда мы генерируем исключение:

1. Создается копия объекта переданного в оператор throw. Этот объект будет существовать до тех пор, пока исключение не будет обработано. Если тип объекта имеет конструктор копирования, то он будет вызван.
2. Прерывается выполнение защищенного try-блока.
3. Выполняется раскрутка стека, пока исключение не будет обработано.

При раскрутке стека, вызываются деструкторы локальных переменных в обратном порядке их объявления. При разрушении всех локальных объектов текущей функции процесс продолжается в вызывающей функции. Раскрутка стека продолжается пока не будет найдем try-catch -блок. При нахождении try-catch-блока, проверяется, может ли исключение быть поймано одним их catch-блоков.

1.3 Как ловится исключение?

Catch-блоки проверяются в том порядке, в котором написаны. По следующим критериям:

1. Если тип, указанный в catch-блоке, совпадает с типом исключения или является ссылкой на этот тип.
2. Класс, заданный в catch-блоке, является предком класса, заданного в throw, и наследование открытое (public).
3. Указатель, заданный в операторе throw, может быть преобразован по стандартным правилам к указателю, заданному в catch-блоке.
4. В catch-блоке указанно многоточие

Если найдет нужный catch-блок, то выполняется его код, остальные catch-блоки игнорируются, а выполнение продолжается после try...catch-блока и исключение считается обработанным. Если ни один catch-блок не подошел, процесс раскрутки стека продолжается.

NB) Так как поиск ведется последовательно, то нужно учитывать порядок catch-блоков (Например, catch(...) должен быть последним).

В некоторых случаях внутри catch-блока может быть необходимо не завершать раскрутку стека. Для этого существует специальная форма оператора throw без аргумента. Она означает проброс текущего исключения. Исключение все еще считается не обработанным.

NB) При это следует заметить, что при повторном выбросе исключения рассматривается не параметр текущего catch-блока, а именно изначальный статический объект. Именно он скопируется в качестве параметра в следующий обработчик. Поэтому его этот статический объект живет пока его исключение не обработается полностью. Поэтому при приведении тип исключения не теряется.

```
class Exception {
public:
    Exception(): value(0) { }
    int value;
};

void third() {
    throw Exception();
    std::cout << "end third\n";
}

void second() {
    try {
        third();
    }
    catch (Exception exc) {
        std::cout << "in second Exception-value = " << exc.value << std::endl;
        exc.value = 100;
        throw;
    }
    std::cout << "end second\n";
}

void first() {
    try {
        second();
    }
    catch (Exception exc) {
```

```
29          std::cout << "in first Exception-value = " << exc.value << std::endl;
30      }
31      std::cout << "end first\n";
32  }
33
34  int main() {
35      first();
36      std::cout << "end main\n";
37      return 0;
38  }
```

Вывод программы:
in second() Exception-value = 0
in first() Exception-value = 0
end first
end main

NB) Также при наследовании классов исключений следует различать catch(type obj) и catch(type& obj). В первом случае при входе в catch блок делается копия объекта-исключения. Во втором случае obj лишь ссылается на этот объект и копии не создается.

Пример:

```
1   struct Exception_base {
2       virtual char const* msg() const {
3           return "base";
4       }
5   };
6
7   struct Exception_derived : Exception_base {
8       virtual char const* msg() const {
9           return "derived";
10      }
11  };
12
13  int f() {
14      try {
15          throw derived();
16      }
17      catch (base e) {
18          std::cout << e.msg() << std::endl;
19      }
20  }
```

```
21
22  int g() {
23      try {
24          throw derived();
25      }
26      catch (base const& e) {
27          std::cout << e.msg() << std::endl;
28      }
29  }
```

В данном примере g() выводит «derived», а функция f() выводит «base», поскольку объект исключения был скопирован с базы объекта, который мы передали в оператор throw и новая копия имеет тип base.

1.4 Function-try-block

Часто мы хотим, чтобы все тело функции находилось в try-блоке. Тогда это try-блок называется функциональным. И для него есть отдельный синтаксис.

```
1  int main() try {
2      //main's body
3  }
4  catch (...) { }
```

Здесь функциональные try-блоки являются синтаксическим сахаром, но есть ситуации когда без них не обойтись: обработка исключений в конструкторе.

Вот есть класс, котором мы хотим ловить и обрабатывать исключения.

```
1  class St {
2  public:
3      St(): member() {
4          try {
5              // Constructor's code
6          }
7          catch (...) { }
8      }
9  private:
10     Member_type member;
11 }
```

Но заметим, что вызов конструкторов мемберов не находится внутри try-блока и исключения возникшие в их конструкторах не поймаются. Поэтому мы используем здесь функциональный try-блок:

```
class St {
public:
    St() try: member() {
        // Constructor's code
    }
    catch (...) {

    } // implicit throw
}
private:
    Member_type member;
}
```

Но у функциональный try-блок в конструкторах, есть особенность: они всегда бросают исключение повторно.

Уничтожение объекта при исключении в конструкторе.

Также важно помнить, что если в конструкторе происходит исключение, то для него не вызовется деструктор, так объект еще не считается созданным.

1.5 Best practice

Часто исключений применяются для корректной работы с ресурсами. То есть если возникает исключение и владеем какими-то ресурсам, то в случае генерации исключения следует их освободить.

Например, мы пишем вектор, и мы хотим скопировать данные в другой участок памяти, чтобы скопировать вектор. При этом если во время копирование какого-то объекта возникнет исключение, то хорошо если уже созданные объекты будут разрушены.

```
template <typename T>
void copy_construct(T* dist, T const * sourse, size_t size) {
    size_t i = 0;
    try {
        for (; i != size; ++i) {
            new (dist + i) T(sourse[i]);
        }
    }
    catch (...) { // если ошибка при копировании
```

```
10          for (size_t j = i; j != 0; --j) {
11              dist[j - 1].~T(); // вызовем деструкторы созданных объектов
12          }
13          throw;
14      }
15  }
```

Полезно знать про стандартные исключения, такие как `std::bad_alloc`, `std::bad_cast`, `std::bad_typeid` и т. д. Они также связанны наследованием и имеют общего предка `std::exception`.

Подробнее можно почитать здесь:
https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm
http://en.cppreference.com/w/cpp/error/exception

Хорошим тоном является наследование от `std::exception`.

Когда мы организовываем исключения в иерархии классов, то получаем мощный механизм описания исключение и способов их обработки. Создав такую структуры мы можем обрабатывать как более общие ошибки, так и более специализированные. Пример:

```
1      class StackException {};
2      class popOnEmpty(): public StackException {};
3      class pushOnFull(): public StackException {};
```

Причем сгенерировав исключение типа popOnEmpty(), мы можем в разных обработчиках независимо выбирать: обработать как popOnEmpty или как StackException, так как тип исключения не теряется при повторной генерации этого исключения.

1.6 Bad practice

- Писать код бросающий исключения в catch-блоке. Это не всегда бывает просто.
- Хотя если исключений не происходит, то по скорости выполнения программа не сильно упадет, но иногда нужно учитывать большой оверхед в случае возникновения и обработки исключения.

1.7 std::terminate()

Это функция, которая вызывается если у механизма исключений не получается корректно отработать, чтобы завершить программу. Случаи когда она вызывается:

- Исключение брошено и не поймано ни одним catch-блоком, то есть пробрасывается вне main().
- Исключение бросается во время обработки другого исключения. Это может произойти только в catch-блоке или деструкторе. А также в функциях, которые вызываются ими.

- Если функция переданная в `std::atexit` и `std::at_quick_exit` бросит исключение.
- Если функция нарушит гарантии noexcept specification. Например, если функция помеченная как noexcept бросит исключение.
- При подобных и не только ошибках в потоках.

По умолчанию `std::terminate()` вызвает `std::abort()`, но можно это изменить, написав свою функцию `my_terminate()` и зарегистрировать ее как терминальную.

```
void my_terminate() {
    cout << "It's not a bug, it's a feature!";
}
/**/
set_terminate(my_teminate);
```

2 Exception safety

2.1 Мотивирующий пример

Пусть у нас есть функция `vector::resize()`;

```
template<typename T>
void vector<T>::resize(size_t size) {
    if (size > curr_size) {
        cnt_add = size - curr_size; // количество элементов для добавления.
        for (size_t i = 0; i < cnt_add; ++i) {
            push_back(T());
        }
    }
}
```

Это код не использует исключения, то есть не вызывает их и не обрабатывает. Но если будет ошибка выделения памяти при расширении вектора, то исключение возникнет в функции push_back(), потом пробросятся через resize() наружу.

То есть проблема в том, что мы не используя механизм исключений все равно можем получить от него проблемы. Как например, не пойманное исключение.

2.2 Определение

Поэтому существуют Гарантии безопасности исключений (Exception safety). Это некий контракт исключений, который представляет из себя ряд уровней безопасности, которые присваиваются всем методам класса. Они декларируют выполнение некоторого контракта относительно состояния объекта после выполнения операций над ним.

Уровни гарантий:

1. «**No guarantees**» - нет ни каких гарантий. После выполнения метода объект и данные в нем могут быть в любом состоянии. Предполагается, что продолжать работу программы нельзя.

2. «**Basic guarantees**» - Гарантируется, что инварианты класса сохраняются и не происходит утечек памяти или других ресурсов.

3. «**Strong guarantees**» - Включает в себя базовую гарантию. А также требует, что в случае исключения объект остается в том, состоянии, в котором он был до выполнения операции. То есть либо операция прошла успешно, или она не повлияла на объект.

4. «**Nothrow guarantees**» - Кроме базовой гарантии, гарантируется, что исключения не возникают.

2.3 Основные моменты

Теперь давайте рассмотрим важные моменты:

- Методы пользовательского интерфейса должны удовлетворять базовым или строгим гарантиям. Это избавляет пользователей от утечек памяти и инвалидных данных.
- Также важно, чтобы деструктор не пробросал исключений, иначе утечки неизбежны. Например, может возникнуть исключение и при очистки стека, возникает еще одно.
- Важность гарантии nothrow: Она есть у очень небольшого количества функции: swap, vector::pop_back, операции с итераторами, Это гарантия очень важна, так как с ее помощью достигается строгая гарантия, когда мы производим необходимы операции на временном объекте, а потом просто делаем с ним swap.

2.4 Best practice

Пример:

```
1 template <typename T>
2 vector<T>& vector<T>::operator=(vector const& other) {
3     return this->swap(vector(other)); // swap trick!
4 }
```

Мы копируем other во временный объект vector(other), а потом делаем swap с ним. Если произойдет исключение при копировании other во временный объект, то оно пробросится к нам. swap() не выполнится и исключение проброситься дальше. Наш объект не поменяется.

NB) Спецификатор noexcept (C++11) указывает компилятору, что выполняется гарантия nothrow. Это важно, для выбора конструктора копирования: перемещающего или нет, так как при перемещении бывает сложно обработать исключение.

Главным способом предотвращением утечек памяти и других ресурсов является идиома RAII-классов (об этом подробнее ниже).

Offtops:

Можно попросить оператор `new` не кидать исключение с помощью константы `std::nothrow`

3 RAII-classes

3.1 Пример

Начнем с примера. Пусть мы хотим открыть несколько файлов.

```
void read_some_file() {
    open_for_read("first_part.txt");
    open_for_read("second_part.txt");
    open_for_read("third_part.txt");
}
```

Причем гарантировать их закрытие в случает возникновения исключения. Тогда давайте напишем такой код:

```
void read_some_file() {
    try {
        open_for_read("first_part.txt");
        open_for_read("second_part.txt");
        open_for_read("third_part.txt");
    } catch (...) {
        // Но мы не может понять какой файл открылся, какой нет. :(
    }
}
```

Так как мы хотим запоминать какие файлы успели открыться, то перепишем так:

```
void read_some_file() {
    size_t i = 0;
    string files[3] = {"first_part.txt", "second_part.txt", "third_part.txt" };
    try {
        for (; i < 3; ++i) {
            open_for_read(files[i]);
        }
    } catch (...) {
        for (size_t j = i; j != 0; --j) {
            close(files[j - 1]);
        }
    }
}
```

Итого код увеличился в два раза.

С помощью RAII-класса файловых дескрипторов reader, можно упростить код:

```
void read_some_file() {
    Reader reader1("first_part.txt");
    Reader reader2("second_part.txt");
    Reader reader3("third_part.txt");
}
```

Этот получился довольно простой и безопасный. Почему безопасный? Потому, что объект reader открывает файл в конструкторе и закрывает в деструкторе. И если возникнет исключение, то при раскрутке стека удаляться все локальные объекты, в том числе и файловые дескрипторы, закрывая файлы. Если же какой-то файл не успел открыться, то не успел и создаться объект отвечающий за него.

Это удобная идея получила название: RAII.

3.2 Определение

«Resource Acquision is Initialozation» или «Захват ресурса - это инициализация» - это идиома класса, который инкапсулирует управление каким-то ресурсом. Она значит, что объект этого класса, получает доступ к ресурсу и удерживает его в течении своей жизни, а потом этот ресурс высвобождается при уничтожении объекта. В конструкторе он должен захватить ресурс(открыть файл, выделить файл и т. д.), а в десткрукторе освободить его(закрыть файл, освободить память и т. д.).

Также важно подумать, что должно происходить при копировании объекта, часто мы просто явно запрещаем это делать.

3.3 Что это дает?

- Удобство кода: нам не приходится каждый раз в конце тела функции освобождать ресурсы. Мы просто заводим локальную переменную. И когда выполнение текущего блока будет завершено, локальные объекты RAII-классов удалятся и ресурсы освободятся автоматически.

- Безопасность исключений: Если вызывается исключение, то нам гарантируется, что стек очиститься и все локальные объекты удаляться, а значит и освободятся ресурсы. Без RAII приходится в ручную освобождать все ресурсы. Причем нужно учитывать какие ресурсы успели захватить до исключения, а какие нет.

- Часто важно освобождать ресурсы в обратном порядке, относительно того, как они были захвачены. Это как раз поддерживается раскруткой стека при удалении локальных объектов.

NB) Это идиома работает не только в C++, а любом языке с предсказуемым временем жизни объектов.

Вот небольшой пример RAII-класса, который будет управлять файлом. Ресурсом является данные типа FILE (формат файла в Си).

```
class File {
public:
    File(char const *filename, char const *mode)
        : _file(fopen(filename, mode)) { } //захват ресурса

    ~File() {
        fclose(_file); // освобождение ресурса
    }

    File(File const &) = delete;
    File operator=(File const &) = delete;

private:
    FILE *_file;
};
```

RAII часто встречается стандартной библиотеке:

- Smart pointers – инкапсулируют несколько видов управления памятью
- i/ofstream
- Различные контейнеры

3.4 Best practice

Вернемся к предыдущему примеру File. Пусть в конструкторе есть код, который может сгенерировать исключение, тогда возникает проблема освобождения ресурса.

```
File::File(char const *filename, char const *mode)
    : _file(fopen(filename, mode)) {
    //код допускающий исключение
}
```

Если в теле конструктора происходит исключение, то деструктор не вызовется, так как объект не считается созданным. Что делать?

Решение № 1 Написать try-catch

```
class File {
public:
    File(char const *filename, char const *mode) try
        : _file(fopen(filename, mode)) {
    //код допускающий исключение
    }
    catch (...) {
        destruct_obj();
    }

    ~File() {
        destruct_obj();
    }

private:
    void destruct_obj() {
        fclose(_file);
    }
    FILE * _file;
};
```

Минусы решения: можно лучше.

Решение № 2 Можно сделать отдельный подкласс, который хранит в себе ресурс.

```
class File {
    struct FileHandle {
        FileHandle(FILE *fh)
        : _fh(fh) { }

        ~FileHandle() {
            fclose(_fh);
        }

        FILE *_fh;
    };
public:
    File(char const * filename, char const * mode)
    : _file(fopen(filename, mode)) {
        // код допускающий исключения
    }

```

```
18      ~File() = default;
19
20  private:
21      FileHandle _file;
22  };
```

Теперь все тоже ок, так как при возникновении исключения, вызовутся деструкторы от все мемберов. Минусы решения: можно еще лучше

Решение № 3 (Изящное) Делегирующий конструктор.

```
1   class File
2   {
3       File(FILE * file)
4           : _file(file) { }
5
6   public:
7       File(char const * filename, char const * mode)
8           : File(fopen(filename, mode)) {
9           // код допускающий исключения
10      }
11
12      ~File() {
13          fclose(_file);
14      }
15
16  private:
17      FILE *_file;
18  };
```

Дело в том, что если мы в конструкторе вызываем другой конструктор, то после его выполнения объект считается созданным и уничтожится в нужное время.