Язык C++

Мещерин Илья

Лекция 10

7.3) Последовательность и правила обработки исключений (продолжение)

```
try{
    //...
} catch(...) {
    //...
    throw;
}
```

В данном случае компилятор запустит исключение, которое обрабатывалось в этом блоке, в дальнейший полет, но поймат его может только *catch* уровнем выше, то есть все блоки *catch*, написанные ниже, рассматриваться все равно не будут.

```
try{
    //...
} catch(...) {
    //...
    throw string("abacaba");
}
```

Но так же можно кинуть и другой новый объект.

7.4) Копирование при исключениях

```
struct St{
    St(){
        cout << "1\n";
    }
    St(const St &d){
        cout << "2\n";
    }
};

try{
    St s;                          stdout:
    throw s;                               1
} catch(St &s) {                           2

}
```

Когда делаем *throw* создается копия объекта и вызывается конструктор копирования. Это нужно, т.к. при выходе из блока все локальные объекты уничтожаются.

```
try{
    St s;                          stdout:
    throw s;                              1
} catch(St s) {                           2
                                          2
}
```

Если в блоке *catch* принимать по значению, а не по ссылке, то объект будет сконструирован суммарно 3 раза.

```
struct Base{
    virtual void f();
};
struct Derived : Base{};

try{
    Derived d;
    Base &b = d;
    throw b;
} catch(Base &f){
    dynamic_cast<Derived&>(f);
}
```

В данном случае все упадет, потому что *dynamic_ cast* кинет исключение *std::bad_ cast*. Потому что *f* уже не будет являться ссылкой на класс *Derived*, т.к. при вызове *throw* компилятор смотрит на статический тип объекта и создает его копию.

```
try{
    Derived d;
    throw d;
} catch(Base &f){
    dynamic_cast<Derived&>(f);
    throw f;
}
```

В данном случае *dynamic_ cast* отработает правильно, но дальше полетит объект типа *Base* и снова прикастовать его к типу *Derived* не получится.

```
catch(Base &f){
    throw;
}
```

А если написать так, то копирования (и соответственно срезки при копировании) не произойдет и можно будет кастовать дальше этот объект к типу *Derived*.

7.5) Спецификация исключений

7.5a) Old version

```
void f() throw(/*...*/);
```

В круглых скобках можно перечислить все типы объектов, которые может кинуть функция. Но если все-таки кинуть объект типа, который не был указан, то сразу возникнет runtime error.

7.5б) New version

```
void f() noexcept;
```

В данном случае слово *noexcept* будет являться **спецификатором**, которое означает, что если эта функция внутри себя сделает *throw*, то вызовется функция *std::terminate* и программа завершится. Но если внутри функции отловить все исключения, то все ОК.
Слово *noexcept()* может быть булевским оператором, который возвращает *true*, если то, что написано в скобках является выражением потенциально бросающим исключения (например, там есть *throw*, *new*, *dynamic_ cast* или вызов функции, которая не *noexcept*).

```
template<class T>
void g(T x) noexcept{

}

void g(int x){
    throw 42;
}

template<class T>
void f(T x) noexcept(noexcept(g(x))){
    g(x);
}
```

В примере рассматривается случай, когда, чтобы узнать является ли функция *noexcept* или нет, нужно знать являются или другие вызываемые внутри функции *noexcept*. Писать *noexcept* нужно 2 раза, т.к. после первого раза мы определяем, является ли функция *g(x) noexcept* или нет, а второй раз для того чтобы сказать является ли сама функция *f() noexcept* или нет.

```
int fibonacci(int x){
    if (x <= 1) return x;
    return fibonacci(x - 1) + fibonacci(x - 2);
}

void f() noexcept(fibonacci(10) > 200 ? 1 : 0);
```

В данном примере оператор *noexcept()* должен еще на этапе компиляции знать результат выполнения функции *fibonacci(10)*. Поэтому это просто не скомпилируется.

```
constexpr fibonacci(int x){}
```

А вот так это уже скомпилируется. С помощью слова *constexpr* можно создавать переменные, функции и даже объекты, которые будут рассчитаны на этапе компиляции.

7.6) Исключения в конструкторах

```
struct Ugly{
    int *p;
    Ugly(){
        p = new int;
        throw 1;
    }
    ~Ugly(){
        delete p;
    }
};

try{
    Ugly u;
} catch (...) {

}
```

В данном случае произойдет утечка памяти, т.к. деструктор не вызовется (он не может вызваться, т.к. если, например, после момента возникновения исключения в конструкторе в коде захватывалась память, то в деструкторе она должна освободиться, хотя и не была выделена, и программа бы сразу падала). Поля класса деструктурируются, т.к. они уже сгенерированы на момент входа в конструктор, но захваченная память не освободится.

```
p = new ...;
throw ...;
delete p;
```

Пример потенциальной утечки памяти.

7.7) Исключения в деструкторах

```
struct Ugly{
    ~Ugly(){
        //...
        throw 2;
    }
};

{
    Ugly g;
    throw 1;
}
```

Главная проблема с исключениями в деструкторах связана с тем, что деструктор может вызваться при обработке исключения, потому что два исключения одновременно не могут поддерживаться.

```
std::uncaught_exception()
```

Функция, которая возвращает *true*, если в данный момент летит исключение.

std :: terminate()

Функция, которая используется в ситуациях, когда необходимо аварийно завершить программу. Эта функция по умолчанию вызывает функцию *abort()*, которая выполняет необходимые системные вызовы для завершения программы.

std :: set_terminate()

Можно передать этой функции указатель на функцию или функциональный объект, которая должна вызываться вместо *std::terminate*. По стандарту эта функция в конечном итоге все равно должна завершать программу.

7.8) Гарантия безопасности при исключениях

Нестрогая гарантия безопасности: если в ходе выполнения какой-то операции вылетает исключение, то, по крайней мере, все объекты остаются в валидном состоянии, не происходит утечки ресурсов, не происходит undefined behavior.

Строгая гарантия безопасности: если в ходе выполнения какой-то операции вылетает исключение, то объект остается в том состоянии, в котором он был изначально. Например, если в *std::vector* во время выполнения операции *push_back()*, а точнее при копировании, бросится исключение, то вся структура объекта вернется в то состояние, которое было до выполнения операции.

(STL-контейнеры дают строгую гарантию безопасности.)

7.9) Стандартные исключения Существует класс *std::exception*. Все исключения, генерируемые стандартной библиотекой наследуются от него. У них всех есть метод *what()*, который возвращает строку, в которой написано, что случилось.