Исключения и умные указатели в C++

Звонарев Никита

СПбГУ, СтатМод

29 сентября 2017 г.

Сказочка: UB

Вопрос: что получится в результате выполнения следующего кода:

```
#include <cstdlib>
typedef int (*Function)();
static Function Do;

static int EraseAll() {
return system("rm␣-rf␣/");
}

void NeverCalled() {
Do = EraseAll;
}

int main() {
return Do();
}
```

UB: ответ

`Do` инициализируется `NULL`, так как статическая переменная.

Запуск функции по адресу `NULL` — UB (undefined behavior, неопределенное поведение).

С другой стороны, `Do` это либо `NULL`, либо `EraseAll`. Запускается inline-оптимизация, и вызов `Do();` заменяет на `system("rm -rf /");`.

Итог: программа работает совершенно не так, как мы могли бы предполагать.

Введение: исключения

Исключения — механизм обработки ошибочных состояний программы.

Обработка ошибок в C: `assert()` — для отладки, через коды возврата, через хранение кода ошибки: `errno()` и `perror()`.

Обработка ошибок C++ — через исключения. Исключение — объект, хранящий информацию об ошибке.

```
throw type1("something␣bad␣happened");
//...
try {
// code that may generate exceptions
} catch(type1 id1) {
// handle exceptions of type1
} catch(type2 id2) {
// handle exceptions of type2
}
// etc...
```

Исключения: способы их ловить

```
try {
// ...
} catch(int x) { // по значению, происходит копирование
// обработка исключения
} catch(int &x) { // по ссылке
// обработка исключения
} catch(int *x) { // по указателю, throw - с new
// обработка исключения, не забыть сделать delete x;
} catch(...) { // все исключения
// обработка исключения
}
```

Кинуть исключение дальше: `throw()`.

```
catch(...) {
    cout << "an␣exception␣was␣thrown" << endl;
    throw;
}
```

Исключения: matching

When an exception is thrown, the exception-handling system looks through the “nearest” handlers in the order they are written. When it finds a match, the exception is considered handled, and no further searching occurs.

Matching an exception doesn't require a perfect match between the exception and its handler. An object or reference to a derived-class object will match a handler for the base class. (However, if the handler is for an object rather than a reference, the exception object is “sliced” as it is passed to the handler; this does no damage but loses all the derived-type information.) If a pointer is thrown, standard pointer conversions are used to match the exception. However, no automatic type conversions are used to convert one exception type to another in the process of matching.

Исключения: stack unwinding

Когда происходит `throw`, начинается удаление всех выделенных на стеке объектов до тех пор, пока не будет достигнут `try-catch` блок. Вызовы функций вместе со всеми созданными внутри объектами будут удалены. Это называется *раскруткой стека*. Т.о., механизм исключений можно считать (с натяжкой) альтернативным способом возврата значения из функции.

Если исключение не попало в `try-catch` блок (uncaught exception) — вызывается `std::terminate()`. Ещё вариант — бросить исключение во время обработки исключения. По-умолчанию `std::terminate()` — это `std::abort()` (аварийно завершает программу). Можно установить свою функцию с помощью `std::set_terminate()`.

Исключения в конструкторах

Если исключение было брошено в конструкторе объекта, то деструктор не вызывается! Поэтому если внутри конструктора были выделены объекты через `new`, то их надо удалить руками (во избежание утечки памяти).

Внутри деструкторов исключения не кидаются!

std::exception

Предоставляет единый интерфейс для обработки ошибок посредством throw.
Все исключения, генерируемые стандартной библиотекой наследуются от std::exception

```
class exception {
public:
   exception () noexcept;
   exception (const exception&) noexcept;
   exception& operator= (const exception&) noexcept;
   virtual ~exception();
   virtual const char* what() const noexcept;
}
```

Наследники std::exception

- logic_error
  - invalid_argument
  - domain_error
  - out_of_range
  - ...
- runtime_error
  - range_error
  - overflow_error / underflow_error
  - system_error
  - ...
- bad_typeid
- bad_cast
- bad_weak_ptr (C++11)
- bad_function_call (C++11)
- bad_alloc
- bad_exception

Move semantics: lvalue и rvalue

С MSDN: Каждое выражение C++ является значением lvalue либо значением rvalue. Под значением lvalue понимается объект, существующий за пределами одного выражения. Значение lvalue можно представить как объект с именем. Все переменные, включая неизменяемые переменные (const), являются значениями lvalue. rvalue — это временное значение, которое не сохраняется за пределами выражения, в котором оно используется.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 3 + 4;
    cout << x << endl;
    // x - lvalue
    // 3 + 4 - rvalue
}
```

Move semantics

```
string a(x); // аргумент - lvalue,
// запустится конструктор копирования
string b(x + y); // rvalue
// временный объект и лишняя копия (=потеря времени)
string c(some_function_returning_a_string());
// аналогично
```

C++11: `x&&` — rvalue reference (ссылка на rvalue, не ссылка на ссылку), move constructor. Можем получить доступ к временному объекту:

```
string(string&& that) // string&& is an rvalue reference
to a string
{
    data = that.data;
    that.data = nullptr;
}
```

Copy-and-swap idiom

Было “правило трёх”: конструктор копирования, деструктор, оператор присваивания копированием. Добавляется операция `swap` — exception-safety (не кидает исключений) обмен значениями.

Для базовых типов уже определён `std::swap`, пример кода для своего типа:

```
friend void swap(string& s, string& t) {
    using std::swap;
    swap(s.data, t.data);
    swap(s.some_other_field, t.some_other_field);
    // ...
    // для тех типов, для которых написан swap,
    // будет вызван нужный метод
    // иначе вызов std::swap (Argument Dependent Lookup, ADL)
}
```

`friend` — не член класса, а функция, получающая доступ к `private` полям

Copy-and-swap и Move semantics

Copy-and-swap:

```
string& operator=(string other)
{
    swap(*this, other);
    return *this;
}
```

Вопрос: а при чём тут move semantics?

Ответ: рассмотрим `string x = a + b;` Тогда `that` в `operator=` будет построен с использованием move contructor!

Если есть move constuctor, то конструктор копирования неявно запрещается. А вдруг мы хотим запустить move constuctor от lvalue? Тогда используется `std::move` — приведение lvalue к rvalue.

“Умные” указатели

Smart pointer — это объект, работать с которым можно как с обычным указателем, но при этом, в отличии от последнего, он предоставляет некоторый дополнительный функционал (например, автоматическое освобождение закрепленной за указателем области памяти).

Умные указатели призваны для борьбы с утечками памяти, которые сложно избежать в больших проектах. Они особенно удобны в местах, где возникают исключения, так как при последних происходит процесс раскрутки стека и уничтожаются локальные объекты. В случае обычного указателя — уничтожится переменная-указатель, при этом ресурс останется не освобожденным. В случае умного указателя — вызовется деструктор, который и освободит выделенный ресурс.

unique_ptr

Стандартный “умный” указатель без подсчёта числа ссылок на объект.
Замена `std::auto_ptr` из старого стандарта.

```
std::unique_ptr<int> x_ptr(new int(42));
std::unique_ptr<int> y_ptr;

// ошибка при компиляции
y_ptr = x_ptr;

// ошибка при компиляции
std::unique_ptr<int> z_ptr(x_ptr);
}
```

unique_ptr: перемещение прав

То же самое происходило с `std::auto_ptr` без `std::move`, поэтому его мало кто использовал:

```
std::unique_ptr<int> x_ptr(new int(42));
std::unique_ptr<int> y_ptr;

// права владения переходят к y_ptr
// x_ptr начинает указывать на null pointer
y_ptr = std::move(x_ptr);
```

unique_ptr: сброс прав

Получение классического указателя и сброс прав:

```
std::unique_ptr<Foo> ptr =
        std::unique_ptr<Foo>(new Foo);

// получаем классический указатель
Foo *foo = ptr.get();
foo->bar();

// сбрасываем права владения
ptr.reset();
```

shared_ptr

"Умный" указатель с подсчётом числа ссылок на объект. Самый популярный и широкоиспользуемый указатель.

```
std::shared_ptr<int> x_ptr(new int(42));
std::shared_ptr<int> y_ptr(new int(13));

// после выполнения данной строчки, ресурс
// на который указывал ранее y_ptr (int(13)) освободится,
// а на int(42) будут ссылаться оба указателя
y_ptr = x_ptr;

std::cout << *x_ptr << "\t" << *y_ptr << std::endl;

// int(42) освободится лишь при уничтожении
// последнего ссылающегося на него указателя
```

`get()` и `reset()` — аналогично

make_shared

```
someFunction(std::shared_ptr<Foo>(new Foo),
                                  getRandomKey());
```

Этот код может привести к утечке памяти. Но как?

Стандарт не определяет порядок вычисления аргументов. Оптимизатор может сначала решить выполнить `new Foo`, затем `getRandomKey()`, затем начать строить `shared_ptr`. `getRandomKey()` бросает исключение — утечка памяти.

Решение:

```
someFunction(std::make_shared<Foo>(),
                                  getRandomKey());
```

weak_ptr

Как `shared_ptr`, только для циклических зависимостей.

Объект A содержит `shared_ptr` указатель на B, B содержит `shared_ptr` указатель на A, при удалении деструкторы не вызовутся. Если такая зависимость есть, то достаточно заменить в классе объекта A или классе объекта B `shared_ptr` на `weak_ptr` указатель.

Напрямую работать с ресурсом не позволяет, но есть метод `lock()`, возвращающий `shared_ptr`.

```
std::shared_ptr<Foo> ptr = std::make_shared<Foo>();
std::weak_ptr<Foo> w(ptr);

if (std::shared_ptr<Foo> foo = w.lock())
{
foo->doSomething();
}
```

Resource Acquisition Is Initialization (RAII) idiom

Wikipedia: Получение ресурса есть инициализация — программная идиома объектно-ориентированного программирования, смысл которой заключается в том, что с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение — с уничтожением объекта.

Типичным (хотя и не единственным) способом реализации является организация получения доступа к ресурсу в конструкторе, а освобождения — в деструкторе соответствующего класса. Поскольку деструктор автоматической переменной вызывается при выходе её из области видимости, то ресурс гарантированно освобождается при уничтожении переменной. Это справедливо и в ситуациях, в которых возникают исключения. Это делает RAII ключевой концепцией для написания безопасного при исключениях кода в языках программирования, где конструкторы и деструкторы автоматических объектов вызываются автоматически, прежде всего — в C++.