Разбор типовых ошибок на примерах Code Review

Русанов Александр
Developer of Apriorit

apriorit

1. Делайте Code Review

Code review дает вам следующие преимущества:

- Улучшается архитектура и качество кода.
- Находятся баги еще до тестирования.
- Разработчики знают, что происходит в других частях проекта.
- Новички быстрее учатся и втягиваются в проект.

2. Компилируйте без warnings

- Компилятор знает C++ лучше вас: позвольте ему помочь.
- Избавляйтесь от warnings путем изменения своего кода, а не понижением уровня предупреждений.
- Включите treat warnings as errors в проекте, дабы наложить физическое ограничение на отсутствие warnings при сборке.
- Для внешней библиотеки, причиняющей warnings, можно понизить уровень предупреждений.
- Включите статический анализатор кода в вашей IDE в режиме подробного вывода.

2. Компилируйте без warnings

```
#include <string>
#include <iostream>

using namespace std;

string foo() {
  cout << "Something is wrong..." << endl;
}


int main() {
  string res = foo();
  cout << res << endl;
}
```

2. Компилируйте без warnings

```
alex@pc:/opt/prog$ g++ -std=c++11 -o test ./test.cpp
alex@pc:/opt/prog$ ./test
Something is wrong...
Segmentation fault (core dumped)
```

```
alex@pc:/opt/prog$ g++ -std=c++11 -o test ./test.cpp -Wall
./test.cpp: In function ‘std::__cxx11::string foo()’:
./test.cpp:8:1: warning: no return statement in function returning non-void
[-Wreturn-type]
 }
 ^
```

3. Не оптимизируйте преждевременно

- Преждевременная оптимизация – усложнения архитектуры или кода во имя производительности, когда это не подкреплено доказанной необходимостью.
- Семь раз **отмерь**, один раз отрежь: 80% времени тратится на выполнение 20% программы, поэтому, прежде чем тратить время на оптимизацию убедитесь, что это именно те 20%.
- Пока у вас нет проваленных требований к производительности программы – пишите **код для людей**.
- Передача параметров по ссылке и другие принятые в C++ идиомы не являются преждевременной оптимизацией.

```
class Player  {
public:
    std::string get_name() const { return m_name; }  // probably inline method
private:
    std::string m_name;
};
```

4. Не пессимизируйте преждевременно

- Пессимизация – упрощение и без того читабельного кода по цене его производительности.
- Если не страдает сложность и читабельность кода, то всегда используйте эффективные паттерны проектирования и идиомы C++.
- Передача параметров по значению (если подходит по ссылке), использование постфиксных ++/-- вместо префиксных операторов, создание излишних временных объектов – это все примеры пессимизации.

```
const std::string keyword = …;
…
const char* str = …;
…
if (keyword == std::string(str)) { … }  // there is no need in std::string() here
```

5. Соблюдайте RAII

- **RAII** - Resource acquisition is initialization. Идиома программирования, когда время жизни ресурса жестко привязано к времени жизни объекта (владение ресурсом – инвариант объекта): захват ресурса происходит во время создания объекта (в конструкторе), освобождение – во время уничтожения (в деструкторе).
- RAII существует чтобы позволить коду соблюдать **exception guarantee** (basic, strong, no-throw).
- Избегайте самописных RAII-объектов, используйте библиотечные (<memory>, <fstream> и т.д.).
- При создании RAII-объекта **явно определяйте или запрещайте** конструктор копирования/перемещения, оператор копирования/перемещения присваиванием, деструктор (**the rule of three, the rule of five**). При реализации используйте **copy-and-swap** идиому.
- Передавайте **владение** ресурсом через RAII-объект. Если передача владения не требуется, а нужно просто использовать ресурс – передавайте “голый” ресурс.
- Захват ресурса и передача его владеющему объекту всегда должны находиться в **отдельном выражении**.

6. Избегайте зависимости от порядка инициализации глобальных объектов

- Порядок инициализации объектов уровня пространства имен не определен. Кроме того, такие объекты статически инициализируются нулями, что усложняет отладку, так как объект кажется валидным, но пустым, хотя ваш код инициализации еще так и не был выполнен.
- Избежать этой зависимости помогает паттерн Singleton, но будьте осторожны – у него много своих недостатков, лучше просто не использовать глобальные объекты.

```
// first.cpp
std::unique_ptr<DataProvider> g_data_provider = new DataProvider;

// second.cpp
std::unique_ptr<DataProxy> g_data_proxy = new DataProxy(*g_data_provider);
```

7. Никогда не перегружайте операторы &&, ||, ,

- При перегрузке операторов необходимо сохранять их первоначальную семантику. Кроме того, не стоит писать код, зависящий от порядка инициализации аргументов функции. При перегрузке этих трех операторов вы нарушаете сразу два этих правила.

- Порядок инициализации аргументов функции не определен.
- Перед вызовом функции всегда вычисляются все аргументы.

```
if (ptr && ptr->connected()) { … }
```

```
f(i++), g(i++);
```

8. Не наследуйте от классов, не предназначенных быть базовыми

Проектирование базового класса отличается от проектирования value-класса. При наследовании от не-базового класса появляются следующие **проблемы**:

- В не-базовом классе деструктор не объявлен как виртуальный, что при удалении объекта по указателю на базовый класс не вызовет деструктор наследника, не почистит его память и приведет к **утечке**.
- При попытке переопределения функций, не объявленных как virtual в базовом классе, происходит их **сокрытие**, из-за чего метод для вызова будет определяться статически и интерфейсы родителя и наследника могут разниться.
- В существующем пользовательском коде объекты базового класса использовались как объекты-значения, что приведет к **срезке** при замене на объект класса наследника.

Для расширения value-классов используйте композицию или функции не-члены.

9. Публичное наследование моделирует отношение “является”

- Наследуйте публично чтобы **быть использованным**, а не чтобы **использовать**. Если необходимо реализовать функционал посредством другого класса, то есть использовать, применяйте **композицию**, реже – приватное наследование.
- Публичное наследование означает, что наследник соблюдает все **гарантии** базового класса, предусловия и постусловия всех переопределенных виртуальных функций совпадают, так же как и списки параметров.
- Классический пример: прямоугольник и квадрат.

10. Определяйте и инициализируйте поля класса в одинаковом порядке

- Члены-данные поля класса всегда инициализируются в порядке объявления в определении класса.
- Порядок, в котором они указаны в списке инициализации в конструкторе, не влияет на сам порядок инициализации.

11. Никогда не вызывайте виртуальные функции в конструкторах и деструкторах

- В C++ динамический тип объекта во время создания совпадает с классом, конструктор которого сейчас выполняется. То же верно про уничтожение и деструкторы.
- Вызов виртуальной функции в конструкторе или деструкторе всегда будет **статически связан**, и может привести к UB.
- При необходимости виртуального конструирования используйте post-initialization либо lazy-post-initalization.

12. Ошибайтесь правильно

- Бросайте исключения **по значению**, ловите **по константной ссылке**. Выделять объект исключения динамически либо невозможно, в случае ошибки “out of memory”, либо некорректно, потому что перехватывающей стороне придется решить, кто освободит эту память. Ловить нужно по ссылке, чтобы предотвратить срезку объекта исключения.
- При переброске исключения используйте просто throw без указания типа дабы избежать срезки.
- Не позволяйте исключениям покидать границ модуля.

13. Не используйте C-style cast

C-style cast имеет следующие недостатки:

- Единый синтаксис для выполнения различных преобразований, потенциально некорректных.
- Зависит от наличия определения типов в области видимости.
- Сложно найти в коде.

C++ casts:

- static_cast – преобразование времени компиляции совместимых типов.
- reinterpret_cast - преобразование времени компиляции несовместимых типов.
- const_cast – преобразование времени компиляции, отменяющее модификатор const у типа данных.
- dynamic_cast – преобразование времени выполнения указателя вниз по иерархии.

Вопросы?

Литература:

- H. Sutter, A. Alexandrescu “C++ Coding Standards. 101 Rules, Guidelines, and Best Pracitces”