C++ style guidelines

v0.01

yuyoyuppe

Для удобства в конце приведён пример кода, аггрегирующий большинство правил и позволяющий быстро вспомнить правила без избыточного чтения.

Глава 1

TDD

Юнит-тесты и TDD

- Каждый тест содержит только одну assert-конструкцию
- Лёгкость читаемости тестов должна быть сопоставима с обычным текстом; приоритет, как и всегда, отдаётся наилучшему выражению intention'а
- Тесты не могут быть длиннее нескольких строк
- Тесты не могут содержать условные конструкции — для предотвращения можно воспользоваться подобным кодом:

```
#define if(a) ERROR__cannot_use_conditional_statement_in_a_test_file;
```

- Тесты должны быть в форме Arrange-Act-Assert[1], т.е. подготовка состояния - выполнение теста - проверка утверждения
- Предпочтительные библиотеки: mocking — fakeit, DI — boost.DI, Testing — boost.test
- Тесты пишутся **до** кода, при этом каждый новый тест изначально должен проваливаться; после создания **минимально** работающей реализации для прохождения этого теста проводится рефакторинг кода[2]
- Каждый тест служит документацией к классу, т.к. характеризует отдельную feature, и, благодаря написанию тестов до кода, мы автоматически покрываем все features, получая 100% code coverage[3]
- Все тесты должны выполняться не более, чем за пару секунд, иначе возникнет соблазн частично их отключать

1. иногда зовётся given-when-then

2. эти три фазы составляют канонический TDD-цикл, длящийся ~10 минут

3. придерживаясь данной практики, надобность в специальных утилитах, позволяющих численно оценить code coverage, отпадает, т.к. не нужно полагаться на малоосмысленные цифры

Глава 2

Форматирование

- Максимальная длина строки - 110[1] символов
- Пробелы вместо табуляций
- Ширина отступа - 4 пробела
- Открывающая фигурная скобка всегда идёт на отдельной строке
- Конструкция switch-case выглядит так:

```
switch(condition)
{
        case 1:
        // ...
        break;

        case 2:
        // ...
        break;

        default:
        // ...
        break;
}
```

- Допускается использование однострочных блоков if-else без фигурных скобок для повышения читаемости
- Комментарии пишутся предыдущей строкой(а не продолжением) перед релевантным кодом на одном с ним уровне отступа
- Одна строка - одно объявление
- Если аргументы не помещаются в вызов/объявление, то нужен рефакторинг, но в крайнем случае это должно выглядеть так:

1. аргумент «80 символов на консоли» неактуален, т.к. её обычно расширяют, а 100 символов Dejavu Mono 12 занимают ~1050 пикселей, что «должно хватить всем»

```
very_long_variable_name.draw_line(
        Point{p1_x, p1_y},
        Point{p2_x, p2_y});
```

Глава 3

Исключения и ресурсы

- Слухи о низкой скорости исключений преувеличены, однако их частое использование накладывает нежелательный отпечаток на стиль программирования
- Исключения используются в местах, где игнорирование ошибки может привести к необратимому краху системы
- Такими ошибками обычно являются: некорректное взаимодействие с внешним[1] ресурсом, неверное использование API-компонентов системы
- Однако, поводом для исключения не являются такие вещи, как: несоблюдение (пост/пре)условий при использовании внутреннего для системы компонента; также, эти условия не нужно задавать assert'ами внутри компонентов, но использовать concepts[2] и систему типов(см. след. раздел о типах)
- throws/noexcept не используются, т.к. для большинства систем его польза сомнительна, тяжело поддерживать консистентность, создаёт визуальный шум
- RAII где только возможно
- new/delete запрещены
- Не кидать исключения в деструкторах и т.п., а также являясь onwer'ом объекта/ресурса

1. независимым от нашей системы
2. будут введены в стандарте C++17 или позднее

Глава 4

Типы

- Предпочтение compile-time проверок исключениям
- T * t, T & t
- C-style массивы запрещены
- const везде, где только возможно[1]
- typedef не используется вообще в угоду читаемому using
- Maybe<T> вместо nullable, return false или исключения для индикации большинства ошибок
- отсутствие «голых типов» в аргументах функций у API-методов и предпочтение им типизированных user-defined literals, к примеру:

```
class KMs
{
public:
        explicit KMs(unsigned long long int value)
        {
                // ...
        }

        operator unsigned long long int()
        {
        }
}
```

- Приведение типов используется только для ублажения компилятора при работе с 3rd party libs; если возможно, этот кошмар помещается в адаптер
- dynamic_cast нежелателен, т.к. зачастую означает code smell — рекомендуется пересмотреть дизайн

1. – в привычку должна войти манера «убирать» препятствующие логике const-квалификаторы

- Злоупотребление умными указателями отодвигает на задний план семантику передачи/разделения ownership и неоправданно[2] снижает производительность; однако голые указатели в параметрах нужно оборачивать в обёртки, например:

```
not_null<T*> check(T* p) { if (p) return not_null<T*>{p}; throw Unexpected_
```

2. проблемы с висячими указателями обнаруживаются компилятором и статическим анализатором

Глава 5

Классы и функции

- Single responsibility principle как основная движущая сила рефакторинга
- Длина функции/метода не более 25 строк
- Общая длина определения класса не более 150 строк[1]
- Синглтоны в любом виде и mutable глобальные переменные запрещены
- Классы, содержащие в названии туманные термины, ассоциирующиеся с программной архитектурой, такие как Interface, Model System(за исключением тех, что общаются с ОС и устройствами), Manager, Director, Factory, Controller и т.п. запрещены[2]
- Имена классов начинаются с прописной буквы и используют camelcase: MessageReceiver, ArtificalObject и т.п.
- Имена функций и переменных начинаются со строчной буквы и используют snake_case: updated_message, start_recording() и т.п.
- Имена классовых переменных заканчиваются подчёркиванием: storage_, id_ и т.п.
- Квалификаторы доступа в классе не повторяются, идут сверху вниз в порядке public-proteced-private
- Внутри квалифицированных блоков сначала идут типы и спец. методы, потом методы, затем поля
- Методы, не изменяющие состояния класса, должны отмечаться как const
- Не стоит бояться virtual, однако предпочитать tag dispatch, CRTP и т.п.
- Иногда public метод можно заменить helper'ом
- Rule of zero, либо объявляем все специальные функции, либо =delete все специальные функции
- In-class member initializers улучшают читаемость

1. соблюдение этого правила очень важно, т.к. оно способствует читаемости и чистоте дизайна
2. да, в системных проектах тоже

- Тривиальные (г/с)еттеры кошмарны
- Для симметричных операторов типа operator== используются free функции

Глава 6

Наследование

- Разрешено наследование одной реализации[1] и любого числа интерфейсов
- Composition over inheritance
- virtual метод не должен иметь default аргументов
- Все базовые классы должны иметь virtual деструктор[2]

1. но лучше обходиться без него вообще; вертикальные иерархии - code smell
2. иначе undefined behavior, конечно

Глава 7

Шаблоны

- Трюки с шаблонами должны повышать читаемость и экспрессивность, а не заглушать intention из-за premature optimization[1]
- Шаблонные классы, определяющие новые virtual методы, запрещены
- tuple предпочтительнее pair из-за редкой нужды в обозначении семантики последней
- Шаблонные типы должны иметь максимум ограничений
- Все методы и типы класса, зависящие не от всех шаблонных аргументов, должны помещаться в базовый класс без шаблонов

1. в идеале она и не возникнет, т.к. её появление — сигнал о том, что TDD оказался заброшен, либо используются слишком большие шаги в TDD-цикле

Глава 8

Агрегированный пример

```
class Person
{
public:
        void say(const std::string & phrase)
        {
                //...
        }

        const std::string name_;
private:
        void think()
        {
                //...
        }

        std::unordered_map<int, int> thoughts_;
}
```