Как писать на C++

Лекторы:
Аспирант МФТИ, Шер Артём Владимирович
Аспирант МФТИ, Зингеренко Михаил Владимирович

24 сентября 2024

Эволюция языка C++

- **C++98 (1998):** Первая стандартизация, введение STL (Standard Template Library)
- **C++03 (2003):** Уточнения и исправления стандартов C++98
- **C++11 (2011):** Move-семантика, лямбда-функции, auto, nullptr и constexpr, а также многого другого; существенное расширение STL (умные указатели, chrono, random, thread и т.д.)
- **C++14 (2014):** Уточнения и доработки C++11
- **C++17 (2017):** Различный синтаксический сахар (структурированные привязки, улучшенная работа с шаблонами), constexpr if; расширение STL (optional, variant, filesystem)
- **C++20 (2020):** Концепты, модули, корутины, оператор <=> и многое другое
- **C++23 (2023):** Доработка и расширение C++20 (expected, print и т.д.)
- **C++26 (Разрабатывается)**

Где искать справку по языку?

- ~~Видео на YouTube~~
- ~~Вопросы на StackOverflow~~
- ~~Ответы на StackOverflow~~
- **cppreference.com**
- **Собственно стандарт** (∼1200 страниц не считая приложений)

C++ — Объектно-Ориентированный Язык

В основе ООП лежит использование объектов (классов) и следующих принципов:

- **Инкапсуляция:** Сокрытие деталей реализации, которое позволяет вносить изменения в части программы безболезненно для других её частей.
- **Наследование:** Создание нового класса объектов путём добавления новых элементов (методов) или изменения поведения существующих.
- **Полиморфизм:** Возможность производить операции над различными классами, обладающими (в некотором смысле) общим интерфейсом.

Инкапсуляция в C++

Инкапсуляция — это ключевой принцип ООП, который заключается в сокрытии деталей реализации и контроле доступа к данным и методам объекта.

- **Модификаторы области видимости:**
  - `private` — доступ к данным и методам ограничен только классом.
  - `protected` — доступ к данным и методам ограничен классом и его потомками.
  - `public` — доступ к данным и методам открыт для всех.
- **Инкапсуляция и наследование:**
  - Наследование может изменять уровень доступа: базовые классы могут быть унаследованы как `public`, `protected` или `private`.
  - Инкапсуляция позволяет скрыть внутреннюю реализацию базового класса, предоставляя потомкам только необходимые интерфейсы.

Пример инкапсуляции и наследования в C++ (1/2)

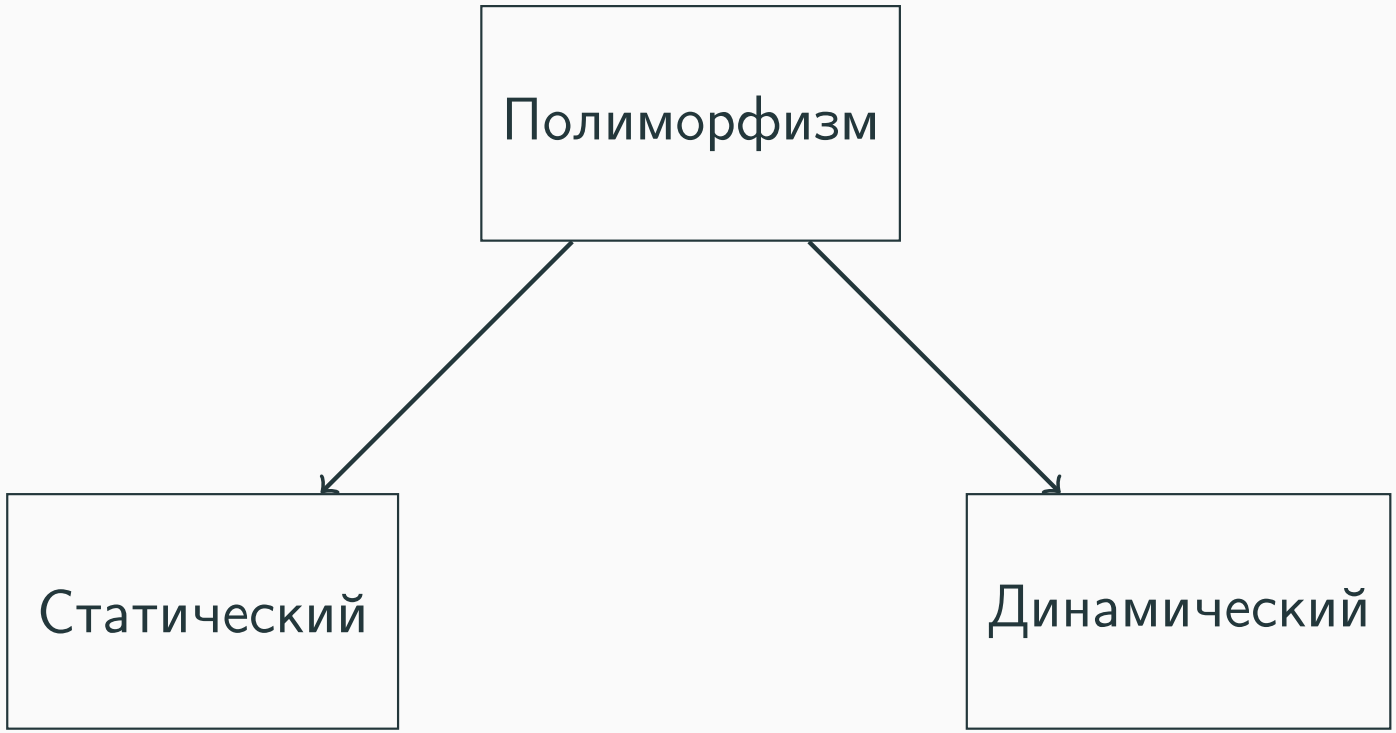
```cpp
class A {
public:
    void foo_a1() {
        std::cout << "Public in " << name << std::endl;
    }
protected:
    void foo_a2() {
        std::cout << "Protected in " << name << std::endl;
    }
private:
    const std::string name = "A";
};

class B : public A {
public:
    void foo_b() {
        foo_a1(); // Accessible: foo_a1() is public in class A
        foo_a2(); // Accessible: foo_a2() is protected in class A,
                  // available in derived class B
        // std::cout << name << std::endl; // Error: name is private in class A
    }
};
```

Пример инкапсуляции и наследования в C++ (2/2)

```
1  class C : private A {
2  public:
3      void foo_c() {
4          foo_a1(); // Accessible: foo_a1() is public in A,
5                    // but becomes private in C due to private inheritance
6          foo_a2(); // Accessible: foo_a2() is protected in A,
7                    // but becomes private in C
8      }
9      using A::foo_a1; // Makes foo_a1() public in C
10 };
11
12 int main() {
13     B b;
14     b.foo_b(); // Works correctly
15     C c;
16     c.foo_a1(); // Works due to the using declaration
17     c.foo_c(); // Works correctly
18     return 0;
19 }
```

Полиморфизм в C++



Шаблонные функции и классы

Виртуальные методы классов
Полиморфизм типов (как в C)
reinterpret_cast, dynamic_cast

Шаблонные функции в C++

```cpp
#include <cstring>
#include <iostream>

// Template function example
template <typename T, typename S>
T my_max(T a, S b) {
    return (a > b) ? a : b;
}

template <> // Explicit specialization for const char*
const char* my_max<const char*, const char*>(const char* a, const char* b) {
    return (std::strcmp(a, b) > 0) ? a : b;
}

// ERROR: Function template partial specialization is not allowed
// template <typename T>
// const T my_max<T, void*>(T a, void* b) {
//     return a;
// }
int main() {
    std::cout << my_max(1, 5) << std::endl; // Ok: 5
    // instantiating my_max<int, int>
    std::cout << my_max(101.3, 5) << std::endl; // Ok: 101.3
    // instantiating my_max<double, int>
    std::cout << my_max("hello", "world") << std::endl;  // Ok: world
    // std::cout << my_max("hello", nullptr) << std::endl;
    // ERROR: invalid operands to binary expression ('const char *' and '
     nullptr_t')
    return 0;
}
```

Шаблонные классы в C++

```cpp
#include <iostream>

template <typename T, typename S>
class MyPair {
private:
    T a; S b;
public:
    MyPair(T first, S second) : a(first), b(second) {}
    T getMax() const {return (a > b)? a : static_cast<T>(b);}
};
// Partial specialization of class is OK
template <typename T>
class MyPair<T, const std::nulptr_t> {
private:
    T a;
public:
    MyPair(T first, const  std::nulptr_t /*second*/) : a(first) {}
    T getMax() const {return a;}
};

int main() {
    MyPair<int, int> intPair(100, 75);
    std::cout  << intPair.getMax() << std::endl; // Output: 100
    // ERROR: invalid operands to binary expression
    //   ('const int' and 'const std::__cxx11::basic_string<char>')
    // MyPair<const int, const std::string&> intstringPair(123, "apple");
    MyPair<const char*, const void*> voidPair("apple", nullptr);
    std::cout << voidPair.getMax() << std::endl; // Output: apple
    return 0;
}
```

Виртуальность в C++

```cpp
// Base class with a pure virtual function
class A {
public:
    std::string name() const {
        return name_impl(); // Calls the derived class's implementation
    }
private:
    virtual std::string name_impl() const = 0; // Pure virtual function
};
class A1 : public A { // Derived class 1
private:
    std::string name_impl() const override { return "A1";}
};
class A2 : public A { // Derived class 2
private:
    std::string name_impl() const override {return "A2";}
};

void print_name(const A &a) {
    std::cout << a.name() << std::endl; // Polymorphic call
}
int main() {
    // A a; // ERROR: variable type 'A' is an abstract class
    A1 a1;
    A2 a2;
    print_name(a1); // Output: A1
    print_name(a2); // Output: A2
    return 0;
}
```

Не Виртуальные Интерфейсы в C++

- Когда виртуальные функции должны быть public, protected или private?
  - Рекомендация: **Предпочтительно делайте виртуальные функции private**.
- **Зачем?**
  - Разделение интерфейса и реализации.
  - Контроль за выполнением предусловий и постусловий базовым классом.
  - Меньшая хрупкость класса при изменениях.
- Что насчёт деструкторов?
  - Рекомендация: **Деструктор базового класса должен быть публичным и виртуальным (почти всегда).**

Подробнее здесь

Какой полиморфизм выбрать? (1/6)

```cpp
// Enum for operations
enum class Operation { Add, Mul, MaxAbs };
constexpr int MODULE = 1000*1000*1000 + 9;
// Templated class ReductorTpl
template <Operation Op>
class ReductorTpl {
public:
    void load(const std::vector<int>& data) { this->data = data; }

    int reduce() const {
        int r = 0;
        for (int x : data) { r = operator_(r, x);}
        return r;
    }
private:
    int operator_(int r, int x);
    std::vector<int> data;
};
template<>
int ReductorTpl<Operation::Add>operator_(int r, int x) {return r + x;}
template<>
int ReductorTpl<Operation::Mul>operator_(int r, int x) {
    return (r * x) % MODULE;
}
template<>
int ReductorTpl<Operation::Mul>operator_(int r, int x) {
    return std::max(r, std::abs(x));
}
```

Какой полиморфизм выбрать? (2/6)

```
1  class ReductorVirtual {
2  public:
3      void load(const std::vector<int>& data) {data_ = data;}
4      int reduce() const {
5          int r = init_();
6          for (int x : data_) { r = operator_(r, x);}
7          return r;
8      }
9      virtual ~ReductorVirtual() = default;
10 private:
11     virtual int init_() const {return 0;}
12     virtual int operator_(int r, int x) const = 0;
13     std::vector<int> data_;
14 };
15
16 class ReductorVirtualAdd : public ReductorVirtual {
17 public:
18     int operator_(int r, int x) const override {return r + x;}
19 };
20 class ReductorVirtualMul : public ReductorVirtual {
21 public:
22     int init_() const override {return 1;}
23     int operator_(int r, int x) const override {return (r * x) % MODULE;}
24 };
25 class ReductorVirtualMaxAbs : public ReductorVirtual {
26 public:
27     int operator_(int r, int x) const override {
28         return std::max(r, std::abs(x));
29     }
30 };
```

Какой полиморфизм выбрать? (3/6)

```
1  template <Operation op>
2  std::pair<double, int> measure_reduce_time_tpl_run(const std::vector<int> &data)
       {
3      ReductorTpl<op> reductor;
4      reductor.load(data);
5      auto start = std::chrono::high_resolution_clock::now();
6      int r = reductor.reduce();
7      auto end = std::chrono::high_resolution_clock::now();
8      return {1e-6 * std::chrono::duration_cast<std::chrono::nanoseconds>(
9                                  end - start).count(), r};
10 }
11
12
13 std::pair<double, int> measure_reduce_time_tpl(const std::vector<int> &data,
14                                                 Operation op) {
15     if (Operation::Add == op) {
16         return measure_reduce_time_tpl_run<Operation::Add>(data);
17     } else if (Operation::Mul == op) {
18         return measure_reduce_time_tpl_run<Operation::Mul>(data);
19     } else if (Operation::MaxAbs == op) {
20         return measure_reduce_time_tpl_run<Operation::MaxAbs>(data);
21     }
22     throw std::runtime_error("incorrect opperator");
23 }
```

Какой полиморфизм выбрать? (4/6)

```
std::pair<double, int> measure_reduce_time_virt(const std::vector<int> &data,
                                                Operation op) {
    std::unique_ptr<ReductorVirtual> reductor;
    if (Operation::Add == op) {
        reductor = std::unique_ptr<ReductorVirtual>(new ReductorVirtualAdd);
    } else if (Operation::Mul == op) {
        reductor = std::unique_ptr<ReductorVirtual>(new ReductorVirtualMul);
    } else if (Operation::MaxAbs == op) {
        reductor = std::unique_ptr<ReductorVirtual>(new ReductorVirtualMaxAbs);
    }
    if (reductor.get() == nullptr) {
        throw std::runtime_error("incorrect opperator");
    }
    reductor->load(data);
    auto start = std::chrono::high_resolution_clock::now();
    int r = reductor->reduce();
    auto end = std::chrono::high_resolution_clock::now();
    return {1e-6 * std::chrono::duration_cast<std::chrono::nanoseconds>(
                                        end - start).count(), r};
}
```

Какой полиморфизм выбрать? (5/6)

В нашем примере статический быстрее динамического

- в 7.8 раз для ADD
- без изменений для MUL
- в 4.7 раз для MaxAbs

Вывод: в случае "простых"и маленьких вызовов динамический полиморфизм ограничивает эффективность.

Но что если обернуть в виртуальную функцию не вызов отдельной операции но весь цикл reduce?

- Тогда скорости работы совпадут!

Какой полиморфизм выбрать? (6/6)

Рекомендации:

- **Статический полиморфизм:**
  - Быстрее при частых вызовах и простой логике.
  - Оптимален при известном типе операции на этапе компиляции.
  - Истанциируется в каждой единице трансляции (с.м. далее) и для каждой специализации что может существенно увеличить размер исполняемого файла.
- **Динамический полиморфизм:**
  - Гибкость на этапе выполнения.
  - Незначительная разница в скорости при сложных и редких операциях.
- **Вывод:**
  - Статический полиморфизм для простых, частых операций.
  - Динамический для сложных и изменяющихся сценариев.
  - В любом случае, если сомневаетесь: посоветуйтесь с коллегами.

Зачем разделять заголовки и реализацию?

- **Инкапсуляция деталей реализации:**
  - Заголовочные файлы содержат только интерфейс (объявления функций, классов и т.д.), что позволяет скрыть детали реализации от пользователей.
  - Это упрощает изменение реализации без изменения интерфейса.
- **Ускорение компиляции:**
  - Разделение на заголовки и реализацию позволяет компилятору пересобирать только измененные компоненты.
- **Модульность и повторное использование кода:**
  - Заголовочные файлы можно подключать в разных местах проекта.
  - Реализация содержится в одном месте и компилируется один раз, что предотвращает дублирование кода и облегчает его поддержку.

Путь от исходного кода до программы

- Текст программы на C++ хранится в *исходных файлах*.
- *Исходные файлы* проходят несколько этапов обработки, чтобы стать *единицами трансляции*. Упрощенно, этот процесс можно представить так:
  1. *Лексический анализатор* удаляет комментарии, лишние пробелы и проверяет корректность файла.
  2. *Препроцессор* подставляет заголовочные файлы по директивам `#include`.
  3. *Препроцессор* удаляет неиспользуемый код по директивам `#if`, `#ifdef`, `#ifndef` и т.д.
  4. *Препроцессор* исполняет макросы по директивам `#define`.
  5. Происходит *компиляция*, включающая *синтаксический* и *семантический* анализ.
  6. Происходит инстанциация шаблонов.
- Программа на C++ может состоять из нескольких единиц трансляции. Единицы трансляции транслируются отдельно, а потом линкуются в одну программу.

Подробнее тут: cppreference.com

Единицы трансляции: Важные аспекты

- *Единица трансляциии* формируется из одного исходного файла и всех заголовков, подключенных к нему, включая заголовки, подключенные через другие заголовки.
- Если *заголовок* содержит код (не только определения), он будет продублирован в каждой единице трансляции, к которой подключен этот заголовок.
- Шаблоны инстанцируются отдельно в каждой единице трансляции.
- Если шаблонная функция (или метод класса) объявлена в заголовке, ее инстанциации в разных единицах трансляции будут считаться разными функциями.
- Если определение шаблонной функции находится в исходном файле, другие единицы трансляции могут использовать только те реализации шаблона, которые были инстанцированы при компиляции этого файла.
- **НИКОГДА не разносите реализацию шаблонной функции/метода/класса и их специализацию в разные единицы трансляции! Потенциальное IFNDR.**

One Definition Rule (Неформально)

Неформально

Любая сущность в программе должна быть **определена** только один раз.

Но есть нюансы:

- Это правило про **определения** (definitions), не путать с *объявлениями* (declarations).
- Не все *объявления* являются **определениями** (см. тут).
- Как и любое уважающее себя правило, оно содержит исключения, и именно они оказываются важны для написания сложных программ.

Разберемся с правилом по частям.

One Definition Rule (часть 1)

Из стандарта

No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, or template.

С этим просто: *если вы один раз определили переменную, функцию, класс или шаблон, а потом определили еще раз (при этом в той же единице трансляции) — возникает конфликт имен, компилятор не понимает, что ему делать и выдает ошибку.*

Все помнят про include guard-ы?

Тык и тык

One Definition Rule (часть 2)

Из стандарта

Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program; no diagnostic required... An inline function shall be defined in every translation unit in which it is odr-used.

Что такое odr-used?

- Объект *odr-used*, если его значение читается (кроме constexpr), записывается, берется его адрес, или на него ссылается ссылка.
- Ссылка *odr-used*, если она используется и ее объект не известен на этапе компиляции.
- Функция *odr-used*, если она вызывается или берется ее адрес.

Если объект, ссылка или функция являются *odr-used*, их определение должно существовать где-то в программе (и если функция не inline – один раз). Нарушение этого правила обычно приводит к ошибке на этапе линковки.

Что такое inline-функция?

- **Inline-функция** — это функция, объявленная с ключевым словом `inline`.
- **Inline-функции** могут иметь несколько определений в разных единицах трансляции, не вызывая ошибок линковки, что позволяет определять их в заголовочных файлах.
- Однако все определения **inline-функции** должны быть идентичны. Иначе **IFNDR**.
- `inline` — это единственный способ безопасно определять функции (включая явную специализацию шаблонов) в заголовочных файлах без проблем с линковкой.
- Изначально ключевое слово `inline` использовалось (и до сих пор используется в языке C) для указания компилятору, что встраивание функции предпочтительнее вызова, чтобы уменьшить накладные расходы. Сейчас компиляторы могут игнорировать это указание и самостоятельно принимать решение о встраивании.

А если очень хочется попросить компилятор встраивать функцию?

- Это нельзя сделать средствами самого языка C++.
- Но некоторые компиляторы позволяют (с помощью атрибутов компилятора), к сожалению, не единообразно.
- Для этого в GCC используется ключ
  inline __attribute__((always_inline))
- Если inline (в смысле атрибута) функция используется в разных единицах трансляции ее **определение** должно быть вынесено в заголовочный файл.

Библиотеки в C++

- Header-only и компилируемые
- Статические и разделяемые
- Работа с библиотеками с помощью CMake

Что такое библиотека?

- **Библиотека** – набор неких готовых ресурсов (исходного кода и данных) предназначенный для использования программами.
- Библиотеки позволяют повторно использовать код, уменьшая дублирование и ускоряя процесс разработки.
- Библиотеки облегчают управление большими проектами и позволяют интегрировать внешние решения в свой код.

Header-only и компилируемые библиотеки

- **Header-only библиотеки:**
  - Состоят только из заголовочных файлов, которые содержат как интерфейс, так и реализацию.
  - Код библиотеки компилируется вместе с пользовательской программой в каждой единице трансляции, где подключена библиотека.
  - Примеры: `Eigen` или вот.
- **Компилируемые библиотеки:**
  - Состоят из заголовочных файлов и файлов реализации (`.cpp`), которые компилируются отдельно.
  - Могут поставляться в виде набора заголовочных файлов и скомпилированного объектного файла самой библиотеки.
  - Требует линковки объектных файлов или бинарных библиотек во время сборки программы.
  - Примеры: стандартная библиотека C++ и многое другое.

Преимущества и недостатки Header-only библиотек

- **Преимущества:**
  - **Простота использования:** Не требуется отдельная компиляция, упаковка или установка. Достаточно `#include` нужные файлы в коде.
  - **Оптимизация:** Компилятор имеет доступ ко всему исходному коду библиотеки, что позволяет лучше оптимизировать выполнение.
  - **Отсутствие проблем с опциями компиляции:** Поскольку библиотека компилируется вместе с проектом, нет проблем с различиями в настройках компиляции или кроссплатформенной сборкой.
- **Недостатки:**
  - **Долгое время компиляции:** Каждый раз, когда библиотека включается, компилятор должен обрабатывать весь код.
  - **Избыточный код:** Многократное включение кода может привести к увеличению объема машинного кода из-за избыточного инлайнинга.
  - **Отсутствие инкапсуляции:** Сложнее разделить интерфейс и реализацию, что делает код менее понятным и управляемым.
  - **Уязвимость к изменениям:** Любое изменение в библиотеке требует перекомпиляции всех единиц трансляции, которые её используют.

Мы обычно не пишем hedear-only библиотеки как часть наших проектов.

Динамические и статические библиотеки

- **Статические библиотеки:**
  - **Статическая линковка:** Происходит во время создания исполняемого файла или другого объектного файла. Все необходимые модули связываются и включаются в исполняемый файл.
  - **Статическая библиотека:** Файл, предназначенный для статической линковки и часто называемый архивом и набор заголовков.
  - **Автономное выполнение:** Исполняемый файл, созданный с использованием статической линковки, не зависит от внешних библиотек во время выполнения.
- **Динамические (разделяемые) библиотеки:**
  - **Динамическая линковка:** Происходит во время загрузки или выполнения программы. Модули загружаются из отдельных файлов, называемых разделяемыми объектами или динамическими библиотеками.
  - **Разделяемая библиотека:** Файл, который может использоваться одновременно несколькими исполняемыми файлами или другими разделяемыми объектами.
  - **Отложенная линковка:** Линковка может быть отложена до момента, когда программа начнет выполнение, что уменьшает размер исполняемого файла.

Преимущества и недостатки

	Статические библиотеки	**Динамические библиотеки**
+	• Нет зависимости от внешних библиотек при запуске. • Упрощает переносимость исполняемого файла.	• Экономия памяти за счет разделения ресурсов. • Возможность обновления библиотек без перекомпиляции. • Меньший размер исполняемого файла.
−	• Увеличение размера исполняемого файла. • Необходимость повторной линковки при изменении модулей. • Запредельный объем оперативной памяти, при параллельной линковке.	• Проблемы с совместимостью версий библиотек. • Зависимость от внешних файлов при запуске.

До следующей лекции!