Лекция 10 — ООП

Содержание

”Я придумал термин «объектно-ориентированный», и могу сказать, что я не имел в виду C++.”
— Алан Кей

ООП как парадигма

Мир состоит из объектов, и сам мир — объект. При исследовании природы вещей возникает естественное желание такие объекты классифицировать и объединять в более сложные структуры, которые учитывали бы взаимодействия между ними и их внутреннюю иерархию. Эта идея положена в основу концепции объектно-ориентированного программирования.

В широком смысле ООП предлагает взгляд на программирование как на процесс моделирования информационных объектов. Так на него смотрели ещё его первопроходцы — создатели языка Simula, который был разработан в рамках проекта по программному моделированию метода Монте-Карло. И хотя может показаться, что такая парадигма появилась в первую очередь благодаря тому, что самое понятие «объект» естественным образом возникает при решении слишком большого числа задач, это не совсем так.

Дело в том, что в 60-х годах прошлого века не было стандартных форматов или полноценных ОС в привычном понимании этих явлений. Разные способы представления информации требовали разного кода для доступа к ней, что ощутимо затрудняло процесс её передачи. Для решения этой проблемы Алан Кей — один из отцов-основателей ООП — использовал специальные микропрограммы, которые отвечали за развертку данных на других машинах. Позже он осознал важность этой идеи и масштабировал её до целой системы модулей, которые объединяли в себе данные и методы их обработки и имели

возможность взаимодействовать между собой через определенные пользователем интерфейсы, при этом не имея представления о том, что происходит во внешнем мире. Так родился первый чистый объектно-ориентированный язык программирования — Smalltalk.

В наше время ООП ориентировано на крупные программные комплексы, разрабатываемые большими командами, и потому практически преследует две основные цели:

1. структурировать информацию с точки зрения управления и иерархии, сделав работу с кодовой базой максимально простой и понятной для пользователя;
2. разделить ответственность отдельных частей программы.

Введение понятия «объект» решает именно эти задачи. Оно позволяет абстрагироваться от деталей реализации конкретных модулей и работать с достаточно высокоуровневыми интерфейсами, взаимодействие с которыми осуществляется в доступном человеку формате. Когда мы оперируем абстракциями, избыточность данных сводится к минимуму, причем их целостность сохраняется сама собой, что позволяет эффективно и безболезненно расширять функционал программы. При этом изоляция внутренней структуры объекта от внешней среды и однозначность межобъектного взаимодействия позволяют поддерживать отдельные модули обособленно друг от друга, не нарушая работоспособности кода.

Основные положения ООП

1. Абстракция

> ⓘ **Определение**
>
> **Абстракция** — придание объекту характеристик, которые чётко определяют его концептуальные границы, отличая от всех других объектов.

Как уже было сказано, одна из главных задач ООП заключается в определении интерфейса взаимодействия между пользователем и модулем программы — именно в этом и заключается процесс абстракции. Фундаментальная идея состоит в том, чтобы разделить несущественные детали реализации подпрограммы и существенные для её использования характеристики.

Каждый объект характеризуется двумя основными понятиями: *состоянием*, то есть непосредственно хранящимися данными, и *поведением*, то есть механизмами работы с этими данными. Здесь же возникает понятие *инварианта*:

Определение

Инвариант — внутреннее непротиворечивое состояние объекта.

На этапе проектирования объекта нам критически важно закрепить те его свойства, которые необходимы для корректного взаимодействия с ним, а также гарантировать, что эти свойства не будут нарушаться ни во внутренних, ни во внешних процессах, в которых он участвует.

---

2. Инкапсуляция

Задача инкапсуляции

Мы уже знаем, что ООП основывается на объединении данных и методов их обработки в единую логическую единицу. При этом традиционно сами данные, как и весь внутренний объектный интерфейс, остаются сокрыты от пользователя. Причина этого в том, что когда пользователь получает доступ к тем частям интерфейса, которые не были для него предназначены, происходит две вещи:

1. нарушаются границы абстракции;
2. возникает риск нарушения инварианта.

Ни то, ни то, очевидно, не является желательным, поэтому в современном ООП к задаче объединения данных всегда относят также задачу их сокрытия. Обе из них можно обозначить единым термином:

Определение

Инкапсуляция — размещение данных и методов работы с ними в едином компоненте, а также разграничение доступа объектов ко внутренним механизмам друг друга.

Стоит отметить, что хотя в C++ понятия инкапсуляции и сокрытия связаны настолько тесно, что их можно отождествить, это относится не ко всем языкам, поэтому глобально эти явления следует разделять.

Понятие класса

Определение

Класс — универсальный комплексный тип данных, состоящий из семантически единого набора переменных более элементарных типов и функций для работы с этими переменными.

Практически говоря, класс — это и есть модель информационного объекта, о которой говорилось ранее. В классе описывается структура объектов определенного типа, а также внешние и внтуренние интерфейсы для оперирования их содержимым.

В C++ для объявления класса используется ключевое слово `class`:

```
class CPerson;
```

Из этой записи, кстати, сразу можно заметить, что классы поддерживают предварительные объявления. Их также можно объявлять локально, то есть внутри вложенных областей видимости (в том числе внутри других классов), что повлияет на них так же, как на переменные или функции.

Поля и методы

Данные, которые хранит объект класса, называются *полями*, а механизмы их обработки — *методами*. И поля, и методы объявляются внутри класса согласно обычному синтаксису объявления переменных и функций соответственно.

Если при этом не объявлен пользовательский конструктор (*см. Специальные методы*), имя поля может совпадать с именем класса:

```
class MyClass {
    int MyClass = 238; // OK!
};
```

Если в этом случае внутри какого-либо метода будет создаваться объект этого же класса, перед соответствующим объявлением необходимо будет указать ключевое слово `class`, поскольку само имя класса теперь будет восприниматься как имя переменной.

Методы класса могут обращаться к объекту, из которого они вызываются, при помощи указателя `this`:

```
class CPerson {
    int age_ = 51;
    void GrowOlder() {
        this->age++;
```

```
    }
};
```

Впрочем, им рекомендуется не злоупотреблять. Хотя в некоторых языках программирования обращение к полям класса из его методов традиционно реализуется именно при помощи `this`, C++ позволяет его не указывать, и этим стоит пользоваться. Испльзовать `this` следует только тогда, когда это необходимо с точки зрения логики метода.

Методы класса также могут быть *константными*:

```
class CPerson {
    std::string name_;
    void PrintHello() const {
        std::cout << "Hello, my name is " << name_ << "!" << std::endl;
    }
}
```

Указание квалификатора `const` на уровне компиляции гарантирует, что внутри этого метода состояние объекта не изменяется. В частности, у константных объектов можно вызывать только константные методы.

Если мы хотим гарантировать только семантическую константность — то есть неизменность состояния объекта с точки зрения пользовательского взаимодействия, — но всё ещё хотим иметь возможность изменять фактическое состояние объекта (например, при работе с метаинформацией), влияние квалификатора `const` можно обойти, пометив нужные поля как `mutable`:

```
class MyClass {
  private:
    mutable int access_counter_ = 0;
    // ...

  public:
    void MyMethod() const {
        access_counter_++; // OK!
        // ...
    }
}
```

Модификаторы доступа

На каждое объявление внутри класса действует *модификатор доступа*, который отвечает за сокрытие данных, регулируя их доступность для внешних частей программы. В C++ выделяется три модификатора доступа:

`public`	Данные, доступные из любой части программы
`private`	Данные, доступные только внутри методов текущего класса
`protected`	Данные, доступные только внутри методов текущего или унаследованного класса (*см. Наследование*)

Если модификатор доступа не указывается явно, поля и методы класса по умолчанию устанавливаются приватными. Обращение к полю или методу с модификатором `private` или `protected` приводит к ошибке компиляции.

Так, наш класс `CPerson` теперь можно определить следующим образом:

```
class CPerson {
  private:
    std::string name_;

  public:
    void PrintHello() const {
        std::cout << "Hello, my name is " << name_ << "!" << std::endl;
    }
};
```

Периодически также бывает необходимо узнавать значения, которые хранятся в приватных полях объекта, при этом всё ещё не имея возможности никак на них воздействовать. Для этого семантически выделяют отдельную категорию методов — *геттеры*:

```
class Vector {
  private:
    int size_;
// ...
```

```
  public:
    int Size() const {
        return size_;
    }
// ...
}
```

Статические члены класса

При объявлении члена класса возможно указание ключевого слова `static`.

Статические поля фактически представляют собой статические переменные в области видимости класса. Они привязаны к самому классу и существуют в единственном виде, сколько бы объектов ни было создано. Они обязаны быть определены глобально с указанием квалифицированного имени:

```
class CPerson {
  public:
    static int population;
}

int CPerson::population = 0;
```

Отсутствие такого определения приводит к ошибке линковки, потому что при объявлении статических полей не происходит инициализации по умолчанию. Это сделано из соображений соответствия ODR. Впрочем, это перестало быть проблемой после появления инлайн-переменных в C++17:

```
class CPerson {
  public:
    inline static int population = 0; // OK!
};
```

Примечательно, что статические числовые константны можно объявлять прямо в теле класса и без указания ключевого слова `inline`:

```
class Math {
  public:
    static const int Pi = 3; // OK!
};
```

...но только пока мы не попытаемся, например, получить их адрес: [1]

```
class Math {
  public:
    static const int Pi = 3;
};

int main() {
    std::cout << &Math::Pi; // Undefined reference to Math::Pi
}
```

В этом случае компилятор вообще не создаёт переменных, а просто подставляет соответствующее значение в нужные места в коде. Отсутствие физического воплощения константной сущности и приводит к ошибке линковки — компоновщик не находит адреса объекта, потому что объекта просто нет.

Статические методы практически являются просто функциями, привязанными к классу как к пространству имён. Они имеют доступ только к статическим полям класса, не имеют указателя `this`, не могут быть виртуальными (*см. далее*), константными или иметь ссылочный квалификатор.

Специальные методы

Конструкторы

Общие представления

> ⓘ **Определение**
>
> **Конструктор** — специальный нестатический метод класса, использующийся для инициализации объектов.

Конструкторы автоматически вызываются компилятором при осуществлении того или иного вида инициализации. Семантически их задача — обеспечить соблюдение инварианта при создании новых объектов.

Конструктор можно объявить как функцию без возвращаемого значения, имя которой совпадает с именем класса:

```
class CPerson {
  private:
    std::string name_;

  public:
```

```
    CPerson(const std::string& name) {
        name_ = name;
    }
}
```

Конструктор, не принимающий аргументов, называется *конструктором по умолчанию* и вызывается при инициализации по умолчанию.
Конструктор, принимающий в качестве аргумента объект своего же класса, называется *конструктором копирования* и вызывается при копирующей инициализации.
Перегруженный оператор `=` называется *оператором копирующего присваивания* и вызывается при копировании уже инициализированных объектов:

```
class A {
  public:
    A() {
        std::cout << "A()\n";
    }

    void operator = (const A& other) {
        std::cout << "void operator = (const A& other)\n";
    }
};

int main() {
    A a1;
    A a2;
    a2 = a1;
}

/* Вывод:
A()
A()
void operator = (const A& other) */
```

Конструкторы, принимающие произвольный набор аргументов, вызываются при прямой и универсальной инициализации.

Глобально процесс инициализации объекта пользовательского класса происходит следующим образом:

1. Если класс не является базовым ни для какого другого класса, все виртуальные базовые классы инициализируются в порядке DFS-обхода, начиная с первого указанного при объявлении. (*см. Наследование*)

2. Все базовые классы инициализируются, начиная с первого указанного при объявлении. (*см. Наследование*)
3. Все нестатические поля инициализируются в порядке объявления в теле класса.
4. Выполняется тело конструктора.

Список инициализации членов

Как только что было отмечено, все поля нового объекта инициализируются до начала выполнения тела конструктора:

```
class Member {
  public:
    Member() {
        std::cout << "Member()" << std::endl;
    }
};

class Main {
  private:
    Member field;

  public:
    Main() {
        std::cout << "Main()" << std::endl;
    }
} obj;

/* Вывод:
Member()
Main() */
```

Этот процесс можно контролировать при помощи *списка инициализации членов*, который располагается до тела конструктора и позволяет указать конструкторы не по умолчанию, которые следует вызывать при инициализации полей класса:

```
class Member {
  private:
    int value_;

  public:
    Member(int x) {
        std::cout << "Member(int x)" << std::endl;
    }
};
```

```
class Main {
  private:
    Member field;

  public:
    Main(int x)
        : field(x)
    {
        std::cout << "Main(int x)" << std::endl;
    }
} obj(238);

/* Вывод:
Member(int x)
Main(int x)
Вместо конструктора по умолчанию вызывается конструктор от int! */
```

Начиная с C++11, если в списке инициализации членов встречается имя класса, то список должен состоять только из этого единственного вхождения. В этом случае конструктор будет называться *делегирующим*, а единственная запись в списке инициализации воспримется компилятором как *целевой* конструктор, которому перенаправляется запрос, после чего управление возвращается к исходному конструктору:

```
class Main {
  public:
    Main(int x, char c) {
        std::cout << "Main(int x, char c)" << std::endl;
    }

    Main(int x)
        : Main(x, 'o')
    {
        std::cout << "Main(int x)" << std::endl;
    }
} obj(238);

/* Вывод:
Main(int x, char c)
Main(int x) */
```

Делегирующие конструкторы не могут быть рекурсивными.

Преобразующие конструкторы

Конструкторы могут быть использованы компилятором для неявного приведения типов:

```
class CPerson {
  private:
    std::string name_;

  public:

    std::string Name() const {
        return name_;
    }

    CPerson(const char* name)
        : name_(name)
    {
        std::cout << "CPerson(const std::string& name)" << std::endl;
    }
};

void PrintHello(const CPerson& person) {
    std::cout << "Hello, my name is " << person.Name() << "!" << std::endl;
}

int main() {
    PrintHello("Petya");
}

/* Вывод:
CPerson(const std::string& name)
Hello, my name is Petya! */
```

В этом примере компилятор успешно вызвал функцию от `CPerson`, приведя строковый литерал к нужному типу путем вызова конструктора.

Для обратного преобразования используются *операторы приведения типов*. Они объявляются как функции без аргументов и без возвращаемого значения, имя которых совпадает с именем результирующего типа:

```
class CRational {
  private:
    int numerator;
    int denominator;

  public:
    operator float() {
```

```
        return (float) numerator / denominator;
    }
}
```

Возвращаемый тип совпадает с результирующим. Начиная с C++14, в качестве результирующего типа также можно указывать ключевое слово `auto`.

Там, где мы получаем возможность достаточно удобно приводить определенные нами типы к стандартным и наоборот, компилятор получает возможность достаточно удобно помешать нам наслаждаться жизнью. На эту тему можно придумать бесчисленное множество примеров, но читатель, скорее всего, и сам прекрасно знаком с тем, насколько сильно иногда разбивают сердце неявные преобразования. Чтобы их избежать, в C++ имеется возможность указать ключевое слово `explicit`, которое запрещает неявную конвертацию типов как в преобразующих конструкторах, так и в операторах приведения:

```
class CPerson {
  private:
    std::string name_;

  public:

    std::string Name() const {
        return name_;
    }

    explicit CPerson(const char* name)
        : name_(name)
    {
        std::cout << "CPerson(const std::string& name)" << std::endl;
    }
};

void PrintHello(const CPerson& person) {
    std::cout << "Hello, my name is " << person.Name() << "!" << std::endl;
}

int main() {
        PrintHello("Petya"); // no matching function for call to
'PrintHello'
}
```

Деструктор

> **Определение**
>
> **Деструктор** — специальный нестатический метод класса, использующийся для освобождения памяти при уничтожении объекта.

Деструктор автоматически вызывается компилятором в момент окончания времени жизни объекта. Их также можно вызвать явно.

Деструктор объявляется аналогично конструктору за исключением двух пунктов: он не может принимать аргументы и перед его именем указывается символ `~`:

```
class CPerson {
  public:
    ~CPerson() {}
}
```

Порядок действий в деструкторе обратен порядку действий в конструкторе:

1. Выполняется тело деструктора.
2. Вызываются деструкторы всех нестатических членов класса, начиная с последнего указанного в объявлении.
3. Вызываются деструкторы всех базовых классов, начиная с последнего указанного в объявлении. (*см. Наследование*)
4. Вызываются деструкторы всех виртуальных базовых классов, начиная с последнего указанного в объявлении. (*см. Наследование*)

Это означает, что даже когда деструктор вызывается явно, он не передает контроль вызывающему фрагменту программы сразу после ключевого слова `return` — перед этим выполняются все описанные выше шаги.

Стандартные специальные методы

Хотя ранее шла речь только о конструкторах и деструкторах, объявленных (*user-declared*) и определённых (*user-provided*) пользователем, при определенных обстоятельствах специальные методы могут также автоматически создаваться компилятором. В случаях, когда это возможно, этот процесс также можно явно спровоцировать при помощи конструкции `= default`:

```
class A {
  public:
```

```
    A() = default; // компилятор сам создаст конструктор!
}
```

Компилятор может самостоятельно генерировать следующие методы:

> **Неявный конструктор по умолчанию**
>
> Если пользователь не объявил *ни один* конструктор, компилятор добавит в класс объявление соответствующего метода, причем всегда с модификаторами `public` и `inline`.
> Если при этом происходит обращение к нему или берётся его адрес, компилятор автоматически его определит. Определенный компилятором стандартный конструктор имеет такой же эффект, как конструктор с пустым телом и пустым списком инициализации членов.

> **Неявный конструктор копирования**
>
> Если пользователь не объявил конструктор копирования, компилятор добавит в класс объявление соответствующего метода, причем всегда с модификаторами `public` и `inline` и без модификатора `explicit`. Если это возможно, то есть если все нестатические поля и базовые классы имеют конструкторы копирования с параметрами вида `const M&`, то объявленный конструктор копирования будет иметь вид `T::T(const T&)`; в противном случае он будет иметь вид `T::T(T&)`.
> Если при этом происходит обращение к нему или берётся его адрес, компилятор автоматически его определит. Такой конструктор поэлементно скопирует все нестатические поля, в том числе непосредственных базовых классов, используя прямую инициализацию. При этом все поля ссылочного типа будут указывать на те же сущности, на которые указывали поля копируемого объекта.

> **Неявный деструктор**
>
> Если пользователь не объявил деструктор, компилятор добавит в класс объявление соответствующего метода, причём всегда с модификаторами `public` и `inline`.
> Если при этом происходит обращение к нему или берётся его адрес, компилятор автоматически его определит. Определенный компилятором стандартный деструктор имеет такой же эффект, как деструктор с пустым телом.

NB! Следует обратить отдельное внимание на слова «ни один» при описании неявного конструктора по умолчанию. Если хотя бы один конструктор был объявлен (возможно, даже не определен), автоматическая генерация конструктора по умолчанию сразу же отключается.

Удалённые специальные методы

Начиная с C++11, существует возможность запрещать вызов функций с определенной сигнатурой путём указания конструкции `= delete`:

```
void foo(double);
void foo(int) = delete;
```

В отличие от функций, которые не были объявлены, удалённые функции наравне с обычными участвуют в разрешении перегрузок. Если функция, которую компилятор выбрал наилучшим кандидатом для вызова, помечена как удалённая, он прервёт процесс компиляции. Так, в представленном примере вместо того, чтобы неявно привести `int` к `double`, он выведет в консоль ошибку `call to deleted function 'foo'`, поскольку именно функция `foo(int)` будет выбрана наиболее подходящей.

В общем случае удаление функций служит мощным инструментом управления процессом разрешения перегрузок, однако на практике чаще всего используется именно при работе со специальными методами классов. В частности, удаление конструкторов позволяет запрещать те или иные виды инициализации объектов, закрепляя их семантические свойства прямо на уровне компиляции:

```
class CPerson {
  private:
    std::string name_;

  public:
    CPerson(const std::string& name)
        : name_(name)
    {}

    CPerson(CPerson& other) = delete; // каждый человек уникален —
                                      // его нельзя скопировать!
}
```

В определенных случаях компилятор может самостоятельно пометить созданный им специальный метод как удаленный:

Конструктор по умолчанию удаляется, если

1. В классе есть не определённое поле ссылочного или константного типа.
2. Конструктор по умолчанию или деструктор одного из полей или базовых классов недоступен (является приватным, удалён или не

	может быть однозначно определён при перегрузке).
Конструктор копирования удаляется, если	1. Объявлен конструктор перемещения. 2. Конструктор копирования или деструктор одного из полей недоступен.
Деструктор удаляется, если	1. Деструктор одного из полей или базовых классов недоступен. 2. Деструктор виртуальный (*см. Наследование*), и поиск функции деаллокации безуспешен.

Объекты с удалённым деструктором нельзя создавать как обычные локальные переменные на стеке. Их можно создавать динамически, но, очевидно, не удалять, поскольку `delete` (не путать с `= delete` — здесь речь про динамическую память!) также вызывает деструктор. Удалять деструктор вообще кажется плохой идеей.

Классы и структуры

Особенно внимательный читатель может заметить, что классы по своей сути очень похожи на другой знакомый нам концепт — структуры. И он окажется прав, потому что с точки зрения языка одно из двух отличий структуры от класса в том, что всё её поля и методы по умолчанию публичны. Тем не менее, настоятельно рекомендуется использовать структуры в классическом смысле, то есть для объединения данных, которые не предусматривают интерфейсов взаимодействия с ними. В случаях, когда такой интерфейс возникает, следует работать с классами.

---

3. Наследование

Концепция наследования

Как отмечалось ранее, при работе с достаточно большими моделями, в которых объекты тесно связаны друг с другом с точки зрения внешних интерфейсов, возникает желание организовывать их в определенную иерархическую структуру, а в идеале — иметь возможность переиспользовать общую для них логику, не копируя её для каждого класса.

Мы, например, прекрасно знаем, что квадрат, треугольник и круг — это двумерные геометрические фигуры, для которых можно посчитать площадь и периметр, и нам хотелось бы, чтобы это знание было закреплено в некотором общем классе «фигура», а сами фигуры являлись бы надстройкой над ним. Эту задачу решает механизм *наследования*.

> ⓘ **Определение**
>
> **Наследование** — объявление новых классов, перенимающих функционал существующих и, возможно, расширяющих его.

Уже из определения видно, какие две главные задачи решает наследование: *расширение* кода и его *переиспользование*. Со структурной точки зрения оно также позволяет организовать иерархию классов, о которой говорилось выше.

В простейшем своём виде наследование очень похоже на *композицию* — архитектурный подход, при котором объекты одних классов вкладываются в другие как поля. Композиция устанавливает отношение «содержит» (*has-a relationship*) и также позволяет переиспользовать код других классов:

```
class Engine;

class Car {
  private:
    Engine engine_; // car has an engine
}
```

В качестве преимущества наследования здесь можно выделить бо̀льшую глубину доступа с точки зрения инкапсуляции (*см. Модификаторы доступа*), хотя это и не является основной причиной прибегать к нему — она будет рассмотрена в следующем разделе.

Класс, от которого происходит наследование, называется *родительским*, *базовым* или *суперклассом*. Наследуемый класс называется *потомком* или *наследником*, а также *дочерним* или *производным* классом.

Чтобы унаследовать один класс от другого, необходимо после имени потомка через двоеточие указать имя базового класса:

```
class Base {
  protected:
    int x = 238;
```

```
}

class Derived : Base {
  public:
    void Print() {
        std::cout << x << std::endl;
    }
}
```

Наследование может производиться более чем от одного класса. В таком случае все базовые классы перечисляются через запятую:

```
class BiologicalCreature;

class SocialCreature;

class Human : private BiologicalCreature, public SocialCreature {};
```

Наследование от текущего класса можно запретить, указав при его объявлении ключевое слово `final`:

```
class Base final;

class Derived : Base {}; // CE: base 'Base' is marked 'final'
```

Как ранее было отмечено, базовый класс всегда инициализируется до наследника, а разрушается — после. По этой причине при конструировании потомка обязательно нужно вызывать конструктор родителя, если для последнего не реализован конструктор по умолчанию:

```
class Base {
  private:
    int x_;

  public:
    Base(int x)
        : x_(x)
    {}
};

class Derived : Base {
  private:
    int y_;
```

```
  public:
    Derived(int x, int y)
        : Base(x) // если не указать, получаем CE:
                  // «constructor for 'Derived' must explicitly initialize
                  // the base class 'Base' which doest not have a default
                  // constructor»
        , y_(y)
    {}
};
```

Модификаторы доступа

Так же, как и к полям класса, к базовым классам применяются модификаторы доступа, которые в случае наследования имеют несколько отличное значение:

`public`	Все унаследованные поля и методы сохраняют те же модификаторы доступа, которые имели в базовом классе.
`protected`	Все унаследованные публичные поля становятся защищёнными.
`private` (по умолчанию)	Все унаследованные поля становятся приватными.

Публичное наследование встречается наиболее часто и прозрачно расширяет интерфейс базового класса, не нарушая его и сохраняя все инварианты. Оно устанавливает отношение «является» (*is-a relationship*) — например, студент является человеком:

```
class CPerson;
class CStudent : CPerson;
```

Наследование структур по умолчанию именно публичное — это второе и последнее их отличие от классов.

Защищённое наследование встречается реже. Внешние части программы не видят, что класс был унаследован, однако это известно как самому наследнику, так и всем его потомкам.

Приватное наследование чаще служит скорее техническим инструментом. Интерфейс базы скрывается от всех частей программы, кроме самого наследника. Практически, приватное наследование отвечает только за аспект переиспользования кода — расширение интерфейсов с его помощью не достигается.

Затемнение имён

При наследовании имеет место явление затемнения имён, схожее с тем, которое возникает в случае полей видимости:

```
class Base {
  public:
    void foo(int x) {
        std::cout << "foo(int x)\n";
    }
};

class Derived : public Base {
  public:
    void foo() {
        std::cout << "foo()\n";
    }
};

int main() {
    Derived obj;
    obj.foo(5); // CE: too many arguments to function call
}
```

Практически, при указании любого имени в теле наследника все поля и методы с этим же именем, принадлежащие базовому классу, будут игнорироваться компилятором. Если нам необходимо обратиться к члену базового класса, мы должны явно указывать его имя:

```
int main() {
    Derived obj;
    obj.Base::foo(5); // OK!
}
```

Техническое устройство наследования

Практически, наследник представляет собой не что иное, как конкатенацию собственных членов с членами своих базовых классов, причём порядок этой конкатенации не

закреплён, хотя почти всегда базовый класс идёт раньше, чтобы указатели на него и на его потомка совпадали.

Эта структура наследованных классов имеет ключевое значение при их приведении к базовым. В процессе такого преобразования вся дополнительная информация, которую несёт в себе наследник, просто отсекается:

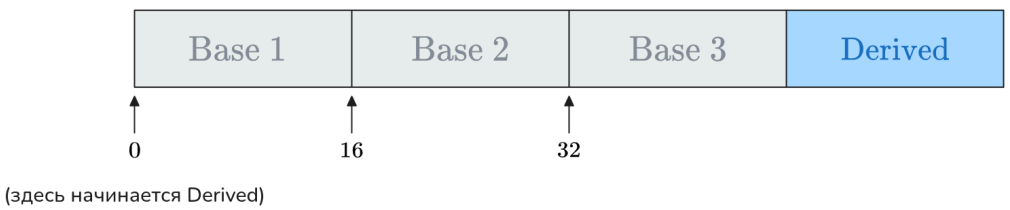

```
Derived obj;
```


```
Base base_obj = obj;
```

Этот процесс называется *slicing*. Он вызван тем, что конструкторы базового класса ничего не знают о полях и методах, которые были доопредлены в его потомках. Пренебрежение этим обстоятельством может приводить к критическим ошибкам при работе с функциями, принимающие базовые классы по значению, или с коллекциями полиморфных объектов (*см. Полиморфизм*).

Язык гарантирует, что базовый класс остаётся в памяти неимзменным. Как и с любой другой гарантией в C++, не до конца ясно, как именно это поддерживается — современные компиляторы могут применять различные оптимизации (вроде *tail padding reuse*, которая позволяет переиспользовать байты выравнивания базовых классов и размещать в них поля наследников)[2] для уменьшения размеров производных классов, и эта тема достаточно глубока, чтобы не поднимать её.

Множественное наследование реализуется аналогично обыкновенному, однако является, конечно, более сложной задачей. Расположение базовых классов в памяти при множественном наследовании не регламентируется. В этой ситуации также имеют место дополнительные траты, связанные с приведением к более старшим типам, поскольку в процессе становится необходимо сдвигать указатели:



Эта тонкость особенно влияет на работу со встроенными преобразованиями. Например, тогда как `static_cast` осуществляет такой сдвиг, `reinterpret_cast` — нет, из-за чего его нельзя использовать для подобных конверсий.

Проблемы множественного наследования

Хотя множественное наследование может потребоваться при решении ряда задач, достаточно легко догадаться, что этот подход скорее вызывает проблемы, чем решает. Самая очевидная из них — неоднозначность обращения при совпадении имён:

```
class Base1 {
  public:
    int var;
}

class Base2 {
  public:
    int var;
}

class Derived : public Base1, public Base2 {};
```

При непосредственном обращении к `var` ни компилятору, ни даже нам не очень ясно, какое именно поле нас интересует. Конечно, это приводит к ошибке компиляции.
Для разрешения этой неурядицы необходимо, как и в случае с сокрытием имён, явно указывать, какому базовому классу принадлежит нужный член:

```
class Base1 {
  public:
    int var = 238;
}

class Base2 {
  public:
    int var = 3104;
}

class Derived : public Base1, public Base2 {};

int main() {
    Derived obj;
    std::cout << obj.Base1::var; // OK!
}
```

Если мы хотим однозначно определить, какой из одноимённых членов мы хотим использовать в наследнике, можно использовать using-декларацию:

```
class Derived: public Base1, public Base2 {
  public:
    using Base2::var;
}

int main() {
    Derived obj;
    std::cout << obj.var; // OK! Программа обратится к var в Base2
}
```

Тем не менее, наиболее практичным советом для решения этой проблемы будет не допускать её возникновения.

Другой известная сложность множественного наследования — проблема *ромбовидного наследования* (*diamond problem*):

```
class CPerson {
  public:
    int age;
};

class CEmployee : public CPerson {};

class CStudent : public CPerson {};

class CIntern : public CStudent, public CEmployee {};
```

Здесь `CIntern` наследуется от `CStudent` и `CEmployee`, которые имеют общего предка — `CPerson`. Первое и наименее проблемное, с чем придется столкнуться — перерасход памяти, поскольку `CIntern` будет содержать две копии базового класса `CPerson`. Немедленно вытекающая из этого факта вторая загвоздка заключается в том, что обращение к `age` внезапно также окажется неоднозначным, поскольку и `CEmployee`, и `CStudent` хранят каждый по своей копии этого поля. Смысла это, конечно, несёт мало.

Для решения этой проблемы в C++ существует механизм *виртуального наследования*. При указании спецификатора `virtual` компилятор гарантирует, что все наследники, в том числе и непрямые, будут обладать только одной копией базового класса:

```
class CPerson {
  public:
    int age = 04;
};
```

```
class CEmployee : virtual public CPerson {};

class CStudent : virtual public CPerson {};

class CIntern : public CStudent, public CEmployee {};

int main() {
    CIntern intern;
    std::cout << intern.age; // OK! Только одна копия age
}
```

Стоит отметить, что виртуальное наследование не совсем «бесплатное» или, скорее, совсем не бесплатное. Его техническая реализация будет подробнее рассмотрена в разделе *Виртуальные функции в памяти*.

Виртуальное наследование требует некоторой предусмотрительности, поскольку ключевое слово `virtual` указывается не для самогò проблемного наследника, а для его базовых классов. По этой (и не только) причине ромбовидного наследования лучше в принципе не допускать — его всегда возможно избежать, несколько пересмотрев архитектуру программы.

Как видно, множественное наследование — вещь в целом не очень приятная. Каждый раз, когда оно возникает в вашем коде, рекомендуется вспоминать, что языки вроде Java или C# вообще от него отказались. Возможно, не просто так.

---

4. Полиморфизм

Наводящие соображения

«Родственность» классов, участвующих в наследовании, помимо прочего означает, что в некоторых контекстах они не отличимы друг от друга в смысле разделения общих интерфейсов и одинакового взаимодействия со внешней средой. Например, рассмотренные ранее классы «студент» и «сотрудник» обладают общей характеристикой «возраст», причем доступ к этой информации вообще не зависит от того, какой именно класс мы рассматриваем, поскольку оба они публично унаследованы от одного предка.

Это замечание приводит к идее «обобщённых» функций, реализующих одинаковые механизмы обработки данных сходных объектов вне зависимости от конкретных типов, к которым они относятся:

```
class CPerson {
  private:
```

```
    int age_;

  public:
    CPerson(int age)
        : age_(age)
    {}

    int Age() const {
        return age_;
    }
};

class CEmployee : public CPerson {
  public:
    CEmployee(int age)
        : CPerson(age)
    {}
};

class CStudent : public CPerson {
  public:
    CStudent(int age)
        : CPerson(age)
    {}
};

void PrintAge(const CPerson& person) {
    std::cout << "I'm " << person.Age() << " years old!" << std::endl;
}

int main() {
    CEmployee employee(31);
    CStudent student(04);
    PrintAge(employee);
    PrintAge(student);
}
```

Возможность работы с такими функциями неразрывно связана с явлением, которое носит название *полиморфизм* (от лат. *πολύς* -— «много, многочисленный» и *μορφή* — «форма, вид»).

Понятие полиморфизма

Определение

Полиморфизм — свойство системы типов, позволяющее использовать сходные механизмы для обработки различных данных.

Функцию, способную работать с данными разных типов, будем называть *полиморфной*.

Полиморфизм — многогранное понятие. На основании различных критериев различают полиморфизм подтипов, параметрический и ad hoc; статический и динамический (также ограниченный и неограниченный, но эти категории рассмотрены не будут). В языке C++ встречаются различные комбинации этих свойств.

Ad hoc полиморфизм, также иногда называемый *специальным*, на практике является полиморфизмом только на уровне интерфейса. Этим термином обозначаются явления перегрузки функций и приведения типов, которые предоставляют лишь единый синтаксис для работы с разнородными данными, но не сам механизм.

Параметрический полиморфизм является «истинным» в том смысле, что он в точности соответствует самой полиморфической семантике. В языке Си параметрический полиморфизм возможно частично реализовать, используя указатели типа `void*`. В C++ для реализации параметрического полиморфизма введена система шаблонов.

Как специальный, так и параметрический полиморфизм являются *статическими* в том смысле, что связывание интерфейсов в этих случаях происходит на этапе компиляции.

Полиморфизм подтипов

Полиморфизм подтипов подразумевает способность взаимодействовать с объектами различных типов через единый интерфейс базового класса. Он позволяет дочерним типам заменять собой родительские, не влияя при этом на функционал и работоспособность подпрограмм, которые с ними работают.

Полиморфизм подтипов является *динамическим*. Это означает, что связывание интерфейсов происходит на этапе выполнения программы, поскольку компилятору заведомо неизвестно, какой именно метод будет вызываться в теле полиморфной функции (*см. Виртуальные методы*).

В контексте ООП всегда рассматривается именно полиморфизм подтипов. Он тесно связан с наследованием (вплоть до того, что эти понятия могут отождествляться) и, по сути, является главной мотивацией к нему прибегать. В современном ООП полиморфизм подтипов является одним из главных положений, поскольку предоставляет огромные возможности для создания расширяемой архитектуры, обеспечивая при этом максимальное переиспользование кода.

Виртуальные методы

Пример функции `PrintAge`, описанный в предисловии, очень косвенно относится к полиморфизму подтипов. Его настоящая сила раскрывается, когда мы позволяем производным классам не просто переиспользовать код родителя, но и изменять само поведение общей функции, оставаясь в рамаках единого интерфейса.

Рассмотрим случай двумерных геометрических фигур и функции для вычисления площади:

```
class Figure {
  public:
    float GetArea() const {
        return 0;
    }
};

class Square : public Figure {
  private:
    float a_;

  public:
    Square(int a)
        : a_(a)
    {}

    float GetArea() const {
        return a_ * a_;
    }
};

float GetArea(const Figure& figure) {
    return figure.GetArea();
}

int main() {
    Square sqr(5);
    std::cout << GetArea(sqr);
}
```

Идея здесь совершенно ясна: мы хотим, чтобы функция `GetArea` вызывала соответствующий метод, определённый в базовом классе и перегруженный в производном, который и передаётся в качестве аргумента. Вместе с этим ясна и проблема этого подхода: в теле функции `GetArea` компилятор воспринимает `figure` как

ссылку на объект базового класса, который по умолчанию ничего не знает про то, как его интерфейс был переопределён наследниками. Приведённая программа ожидаемо выведет 0.

Для того, чтобы организовать корректную работу функций с обозначенной семантикой, используются *виртуальные методы*. Ключевое слово `virtual` при объявлении функции-члена базового класса указывает компилятору, что она может быть переопределена в каком-то из наследников, и выбор конкретной реализации для вызова следует осуществлять в зависимости от фактического типа объекта в момент выполнения программы. Иначе говоря, виртуальные функции служат инструментом для осуществления *динамического* (или *позднего*) связывания.

Через вирутальные методы базовый класс задаёт интерфейс, который непосредственно реализуется уже его наследниками:

```
class Figure {
  public:
    virtual float GetArea() const {
        return 0;
    }
};

class Square : public Figure {
// ...
};

float GetArea(const Figure& figure) {
    return figure.GetArea();
}

int main() {
    Square sqr(5);
    std::cout << GetArea(sqr); // Вывод: 25
}
```

В C++ работа с полиморфными объектами всегда осуществляется при помощи указателей или ссылок. Это ограничение более чем понятно: при приведении к базовому классу вся информация, связанная с переопределёнными методами, теряется в результате слайсинга, в то время как при работе с ссылками и указателями эти данные остаются неизменными.

Языковые нюансы

Синтаксические требования

Для того, чтобы метод производного класса считался переопределяющим метод базового, их сигнатуры должны *полностью* совпадать. Это значит, что даже случайное опущение квалификатора `const` или `volatile` в методе дочернего класса не позволит компилятору сопоставить два этих объявления. В таком случае функция-член наследника будет затемнять базовый метод и не будет рассматриваться как кандидат для виртуального вызова.

Из этого правила существует единственное исключение. Если методы базового и производного класса возвращают ссылку или указатель на классы `A` и `B` соответственно, и при этом `B` является наследником (прямым или косвенным) `A`, то объявление в наследнике всё ещё будет считаться переопределяющим:

```
class A {};

class B : public A {};

class Base {
    virtual A& foo();
};

class Derived : public Base {
    B& foo() override; // OK!
};
```

В стандарте это явление носит имя *covariant return types*.

Ключевое слово `override` явно указывает компилятору, что объявляемый метод является переопределением какого-то виртуального базового метода. Безуспешный поиск соответствующей сигнатуры в теле базового класса приводит к ошибке компиляции:

```
class Figure {
  public:
    virtual float GetArea() const {
        return 0;
    }
};

class Square : public Figure {
  private:
    float a_;

  public:
    Square(int a)
        : a_(a)
```

```
    {}

    float GetArea() override { // Ошибка — не указан const
        return a_ * a_;
    }
};
```

Переопределение виртуальных методов можно запретить путём указания спецификатора `final`:

```
class Figure {
  public:
    virtual float GetArea() final {
        return 0;
    }
};

class Square : public Figure {
  private:
    float a_;

  public:
    Square(int a)
        : a_(a)
    {}

    float GetArea() const { // CE: declaration of 'GetArea'
                            //     overrides a 'final' function
        return a_ * a_;
    }
};
```

Ключевое слово `final` здесь несет ту же семантику, что и в случае с наследованием, и может быть полезно при работе со сложными иерархиями классов в ситуациях, когда требуется закрепить какое-то окончательное состояние метода для будущих потомков.

Аргументы по умолчанию

Виртуальные методы не совсем очевидным образом ведут себя при работе с аргументами по умолчанию. Дело в том, что даже несмотря на динамическое связывание самих вызовов функций, аргументы по умолчанию определяются статически согласно типу указателя, что порой может откровенно вводить в ступор. Так, следующий код

```
class Base {
  public:
    virtual void Log(const std::string& msg = "Base") {
        std::cout << "Base::Log " << msg << std::endl;
    }
};

class Derived : public Base {
  public:
    void Log(const std::string& msg = "Derived") override {
        std::cout << "Derived::Log " << msg << std::endl;
    }
};

int main() {
    Derived obj;
    Base& base_obj = obj;
    base_obj.Log();
}
```

совершенно неожиданным образом выведет в консоль `Derived::Log Base`. К этому стоит быть готовым — в C++ стоит быть готовым вообще ко всему.

Конструкторы и деструкторы

Краткость — сестра таланта, а виртуальные методы в конструкторах и деструкторах — зло.

Проблема с ними достаточно понятна: мы можем попытаться вызвать функцию, переопределённую в классе, которого ещё не существует. Контролировать этот процесс откровенно невозможно, поэтому стандарт разводит руками и просто отключает полиморфизм, переставая поддерживать какие-либо связанные гарантии, хотя на практике это, скорее всего, будет приводить к ошибкам линковки. Словом, не надо так делать. Здесь даже пример не нужен.

Абстрактные классы

Спецификатор `virtual` подразумевает, что функция *может* быть переопределена, и в случае, когда этого не происходит, компилятор просто исполняет тело базового метода. Если же мы хотим, чтобы функция *обязана* была быть переопределена, мы можем заменить её тело конструкцией `= 0`, сделав её *чисто виртуальной*:

```
class Base {
    virtual void foo() = 0;
```

```
}
```

Обращение к чисто виртуальной функции приводит к ошибке компиляции. Это значит, что она обязательно должна быть переопределена в наследуемом классе и при этом доступна для вызова.

Как ни странно, мы всё ещё можем объявить тело чисто виртуальной функции за пределами класса:

```
class Base {
    virtual void foo() = 0;
}

void Base::foo() {
    std::cout << "I'm a pure virtual function!" << std::endl;
}
```

Это ни в коем случае не лишает наследников необходимости переопределить соответствующий метод, однако позволяет вызывать его базовую реализацию из производных классов.

> ⓘ **Определение**
>
> Класс, в котором объявлен или унаследован хотя бы один чисто виртуальный метод, называется **абстрактным**.

Абстрактные классы служат чисто техническим инструментом, позволяющим определить фундаментальный для группы наследуемых классов функционал. Пользователь не может создавать объекты таких классов — это мотивируется как их семантикой, так и определением чисто виртуальной функции, — однако может объявлять указатели и ссылки на них.

Виртуальный деструктор

Как уже было отмечено, в случае полиморфизма мы всегда имеем дело только с указателями и ссылками. Это несколько усложняет процесс взаимодействия с объектами и, в частности, вызывает определенные проблемы, связанные с контролем ресурсов:

```
class Base {
  public:
    Base() {
        std::cout << "Base()" << std::endl;
```

```
    }

    ~Base() {
        std::cout << "~Base()" << std::endl;
    }
};

class Derived : public Base {
  public:
    Derived() {
        std::cout << "Derived()" << std::endl;
    }

    ~Derived() {
        std::cout << "~Derived()" << std::endl;
    }
};

int main() {
    Base* obj_ptr = new Derived;
    delete obj_ptr;
}

/* Вывод:
Base()
Derived()
~Base() */
```

В этом примере мы динамически выделяем объект класса `Derived`, но используем для этого указатель на базовый класс `Base` — такая ситуация возникает, например, при работе с коллекциями полиморфных объектов. Когда же мы деаллоцируем выделенную память, компилятор честно вызывает деструктор базового класса и на этом считает свою работу выполненной. И он, в целом, нигде нас не обманывает: мы отдали ему команду «удали объект по этому указателю», а поскольку указатель имеет тип `Base*`, то и деструктор вызывается только у класса `Base`. Если при этом в конструкторе класса `Derived` выделялся ресурс с динамическим временем жизни, такое поведение приведёт к утечке памяти.

Даже несмотря на то, что деструкторы не наследуются, их можно (а в подобных ситуациях *нужно*) объявлять виртуальными. В результате такого объявления компилятор в момент уничтожения объекта по указателю на базовый класс передаст управление деструктору того класса, которому реально принадлежит деаллоцируемый объект — так же, как это происходит при вызове виртуальных методов:

```cpp
class Base {
  public:
    // ...
    virtual ~Base() {
        std::cout << "~Base()" << std::endl;
    }
};

class Derived : public Base {
    // ...
};

int main() {
    Base* obj_ptr = new Derived;
    delete obj_ptr;
}

/* Вывод:
Base()
Derived()
~Derived()
~Base() */
```

> ⚠ **Важно!**
>
> Деструктор абстрактного класса следует всегда делать виртуальным.

Если мы по какой-то причине хотим запретить создание объектов класса, который не является абстрактным с точки зрения интерфейса, мы также можем сделать деструктор *чисто виртуальным*, хотя формально такой класс всё равно автоматически станет абстрактным.

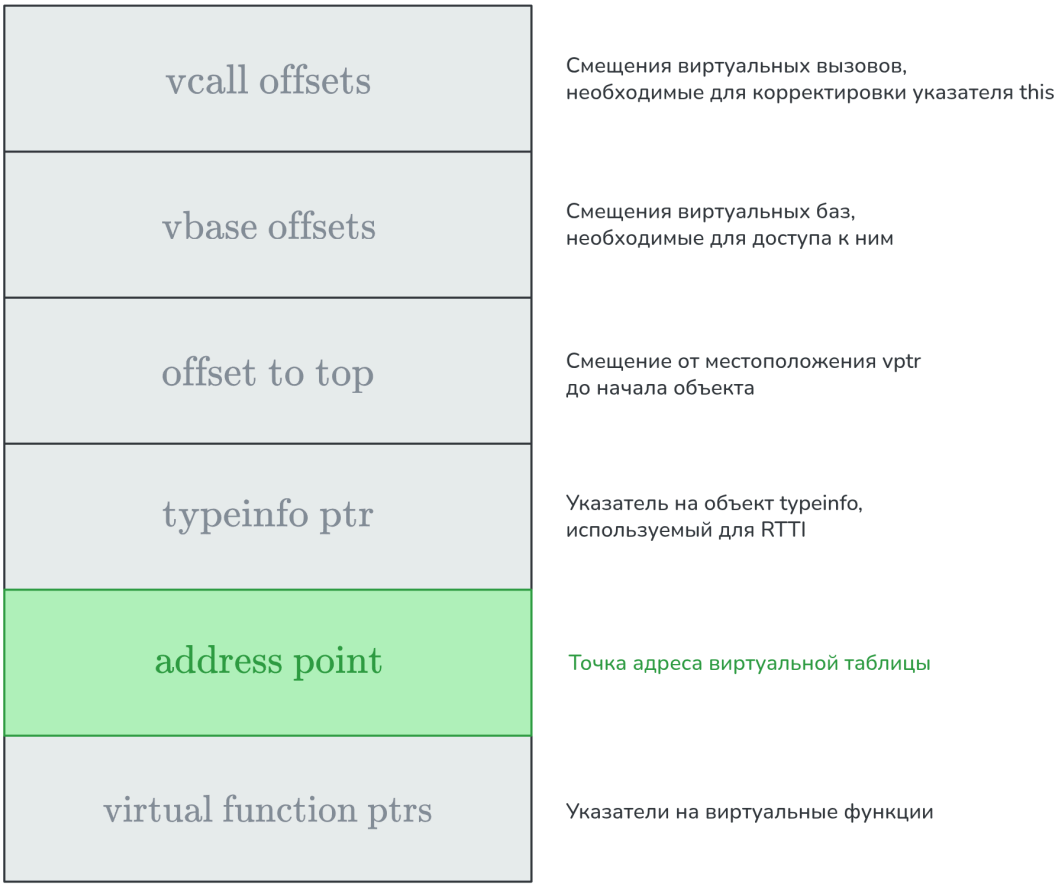
Виртуальные функции в памяти

Хотя техническая реализация виртуальных методов, очевидно, никак не закрепляется Стандартом, на практике из соображений совместимости почти все современные компиляторы используют для этого один механизм — *виртуальную таблицу*.

> ⓘ **Определение**
>
> **Виртуальная таблица** — абстрактная структура данных с произвольным доступом, которая хранит в себе всю информацию, необходимую для:

- вызова виртуальных функций;
- доступа к виртуальным базам;
- идентификации типа в момент выполнения программы (*RTTI — runtime type identification*)

Согласно Itanium C++ ABI (*см. Источники*), виртуальные таблицы имеют следующую структуру:



Кратко поясним предназначение каждого из этих полей.

1. `vcall offsets`

   Здесь хранятся смещения, используемые для корректировки указателя `this` внутри функций, вызываемых через виртуальную базу. Эти данные необходимы по той причине, что местоположение виртуальных (в отличие от обычных) базовых классов относительно начала объекта не зафиксировано — обычно они располагаются в памяти последними, а не первыми, из-за чего их реальный адрес может варьироваться в зависимости от уровней иерархии наследования.

2. `vbase offsets`

> Здесь хранятся смещения, необходимые, чтобы получить доступ к виртуальному базовому классу. Мотивация хранения этих данных такая же, как и в предыдущем пункте.

> 3. `offset to top`
>
> Это значение необходимо для преобразований типов в случае множественного наследования — в частности, для `dynamic_cast<void*>`. Если мы имеем указатель на одну из старших баз в составе сложного объекта, компилятору необходимо знать, где этот объект на самом деле начинается.

> 4. `typeinfo ptr`
>
> Это указатель на объект `typeinfo`, используемый для определения типа в момент выполнения программы. В случае полиморфных объектов там всегда хранится их реальный тип.

> 5. `address point`
>
> Это «точка входа» виртуальной таблицы. Она не совпадает с реальным началом данных. Такое решение необходимо, чтобы вспомогательную информацию (все четыре предыдущие категории) можно было разделять от основной (сами указатели на виртуальные функции), определяя направление смещения указателя (отрицательное или положительное).

> 6. `virtual function ptrs`
>
> Здесь хранятся непосредственно адреса функций, с которыми динамически связываются вызовы виртуального метода базового класса.

Все эти поля заполняются различным образом в зависимости от структуры объекта и его места в иерархии. Механизм взаимодействия с виртуальными таблицами вообще очень сложен, и поэтому не будет рассмотрен. Хотя компиляторы предпринимают достаточно эффективные меры оптимизации при работе с полиморфными объектами, стоит помнить: виртуальные функции, ровно как и виртуальное наследование, совсем не бесплатные, а их совмещение — совсем платное.

Помимо того, что нам, вообще-то, надо хранить кучу сложных структур данных для каждого класса, нам еще и нужно уметь получать к ним доступ. Для этого каждый объект хранит указатель `vptr` на виртуальную таблицу, ассоциированную с его типом. Этот указатель представляет собой скрытое (обычно, самое первое) поле, которое инициализируется адресом `vtable` класса в момент создания объекта. Это поясняет, помимо прочего, в чём состоит беда с виртуальными функциями в конструкторах: в

момент инициализации `vptr` указывает на виртуальную таблицу базового класса, а не текущего.

Убедиться в существовании мистических таблиц и указателей на них очень просто:

```
class NormalBase {
    void foo();
};

class AbstractBase {
    virtual void foo();
};

class NormalDerived : NormalBase {};

class VirtualDerived : virtual NormalBase {};

class AbstractDerived : AbstractBase {};

int main() {
    std::cout << "NormalBase size is " << sizeof(NormalBase) << '\n';
    std::cout << "AbstractBase size is " << sizeof(AbstractBase) << '\n';
    std::cout << "NormalDerived size is " << sizeof(NormalDerived) << '\n';
    std::cout << "VirtualDerived size is " << sizeof(VirtualDerived) <<
'\n';
    std::cout << "AbstractDerived size is " << sizeof(AbstractDerived) <<
'\n';
}

/* Вывод:
NormalBase size is 1
AbstractBase size is 8
NormalDerived size is 1
VirtualDerived size is 8
AbstractDerived size is 8 */
```

Практически, вызов виртуальной функции разворачивается компилятором примерно в следующий код:

```
ptr->foo(3104); // Исходный вызов

(*ptr->vptr[0])(3104) // Обращение к виртуальной таблице
```

или, если быть точнее, в следующие ассемблерные инструкции (clang 21.1.0):

```
mov rdi, qword ptr [rbp - 8]
mov rax, qword ptr [rdi]
call qword ptr [rax]
```

Происходит здесь примерно то же самое, что и в псевдокоде: первой инструкцией берётся адрес объекта, второй — адрес виртуальной таблицы, а третьей происходит одновременно поиск функции в таблице и её вызов.

Важно помнить, что виртуальные функции — не бескомпромиссное решение, и при работе с ними мы всегда платим производительностью как с точки зрения времени работы, так и с точки зрения памяти, пусть и незначительно.

Проектирование классов

Мотивация

Как и любой другой принцип программирования, ООП предлагает принципиально новый взгляд на процесс написания и организации кода, предоставляя при этом внушительное количество инструментов для реализации своих идей. Число этих инструментов и особенно тонкостей при работе с ними если не пугает (пугать должно скорее то, на каком языке мы работаем), то по меньшей мере дезориентирует.

Этот блок несет своей целью представить читателю несколько фундаментальных идей, которые позволяют организовать полученные знания об объектно-ориентированной парадигме и которых следует придерживаться при работе с ней, чтобы гарантировать написание понятного, эффективного и устойчивого кода.

SOLID

SOLID — набор из пяти ключевых принципов проектировния классов в парадигме ООП. Само слово SOLID является мнемонической, т.е. предназначенной для облегчённого запоминания, аббревиатурой от названий положений.

Цель SOLID, как было обозначено абзацем выше — упростить понимание, написание и поддержку программ. Из этого напрямую следует, что эти принципы не универсальны, однако их всегда следует держать в уме. Стоит также отметить, что SOLID, вообще-то, мало кто понимает, и почти все трактуют его положения по-своему (чего стоит только авторитетное мнение хабровцев в комментариях под статьями на эту тему), поэтому не стоит слепо верить примерам, которые будут приведены далее — они носят *строго* иллюстративный и ориентировочный характер.

Итак, в состав SOLID входят:

I. **Single Responsibility Principle,**

или **принцип единственной ответственности**.

> У класса никогда не должно быть более одной причины изменяться или дополняться. Иными словами, каждый класс должен иметь одну единственную зону ответственности. Изменение должно минимально затрагивать код. Функциональность большого класса следует разбивать на более мелкие части, отвечающие за конкретные задачи.

Несколько сложным, но точным примером будет парсинг DSL-файлов. Предметно-ориентированные языки часто подразумевают описание объектов — сущностей, событий или более сложных конструкций, на которые удобно смотреть через призму ООП. Когда мы считываем данные этих объектов, их, конечно, следует создать, прежде чем производить их дальнейшую обработку. Может возникнуть идея делать это прямо в парсере:

```
class Object {
    // ...
};

class Parser {
  private:
    void ReadObject(Object* obj) {
        // ...
    }
    std::vector<Object*> objects;

  public:
    void Parse() {
        // ...
        Object* obj = new Object;
        ReadObject(obj);
        objects.push_back(obj);
        // ...
    }

    ~Parser() {
        for (auto& obj : objects) {
            delete obj;
        }
```

```
    }
};
```

Это, во-первых, обязывает нас пристально следить за утечками памяти и, во-вторых, грубо нарушает принцип единственной ответственности. Да и здравую логику, в общем-то, тоже: достаточно задаться вопросом, почему парсер вообще внезапно должен следить за временем жизни каких-то объектов, когда его единственная задача — осознать содержимое файла.

Решением здесь будет передать управление памятью специальному классу:

```
class Object {
    // ...
};

class Storage {
  private:
    std::vector<Object*> objects_;

  public:
    void CreateObject() {
        objects_.push_back(new Object);
    }

    Object* Back() const {
        return objects_.back();
    }

    ~Storage() {
        for (auto& obj : objects_) {
            delete obj;
        }
    }
};

class Parser {
  private:
    void ReadObject(Object* obj) {
        // ...
    }
    Storage storage_;

  public:
    void Parse() {
        // ...
```

```
        storage_.CreateObject();
        ReadObject(storage_.Back());
        // ...
    }
}
```

II. **Open-Closed Principle,**

или **принцип открытости-закрытости**.

> Каждый класс должен быть закрыт для изменения, но открыт для расширения. Классы следует проектировать так, чтобы другие сущности могли свободно расширять их функционал, при этом не изменяя его.

Рассмотрим уже знакомый пример с геометрическими фигурами. Пусть модель изначально была спроектирована с использованием общего класса «фигура», но на начальном этапе поддерживала только объекты типа «квадрат»:

```
class Figure {
  public:
    void Identify() {
        std::cout << "I'm a square!" << std::endl;
    }
};

class Square : public Figure {
    // ...
}
```

Добавить поддержку объектов типа «треугольник» можно следующим образом:

```
class Figure {
  private:
    FigureType type; // enum

  public:
    void Identify() {
        if (type == kSquare) std::cout << "I'm a square!" << std::endl;
        if (type == kTriangle) std::cout << << "I'm a triangle!" <<
std::endl;
    }
}
```

Это будет нарушением принципа открытости-закрытости, поскольку мы расширили функционал класса путём его непосредственного изменения. Даже без понимания этого принципа достаточно понятно, почему это плохое решение:

- мы безосновательно усложняем логику класса `Figure`;
- мы добавляем классу `Figure` функционал, которым он не должен обладать (управление реестром всех наследников), нарушая принцип единственной ответственности;
- каждое такое изменение может привести к необходимости изменять все зависимые части программы.

Решением служит уже знакомая нам механика виртуальных методов:

```
class Figure {
  public:
    virtual void Identify() = 0; // Figure — абстрактный класс
};

class Square : public Figure {
  public:
    void Identify() {
        std::cout << "I'm a square!" << std::endl;
    }
};

class Triangle : public Figure {
  public:
    void Identify() {
        std::cout << "I'm a triangle!" << std::endl;
    }
};
```

III. **Liskov Substitution Principle,**

или **принцип подстановки Барбары Ли́сков**.

> Функции, использующие указатели и ссылки на базовые классы, должны иметь возможность использовать указатели и ссылки на производные классы, не зная об этом. Наследники всегда должны корректно реализовывать логику своих предков.

На этом принципе держится полиморфизм. Практически он говорит, что не нужно наследоваться от всего подряд.

В качестве примера нарушения принципа подстановки Лисков можно привести классическую *проблему квадрата и прямоугольника*. Ещё со школьной скамьи читателю известно, что любой квадрат является прямоугольником (но не любым), и эту зависимость вполне естественно передать в иерархии классов:

```
class Rectangle {
  protected:
    int a_, b_;

  public:
    Rectangle(int a, int b)
        : a_(a)
        , b_(b)
    {}

    virtual int GetArea() const {
        return a_ * b_;
    }
};

class Square : public Rectangle {
  public:
    Square(int a)
        : a_(a)
    {}

    int GetArea() const override {
        return a_ * a_;
    }
};
```

Казалось бы, жизнь прекрасна, но не тут-то было.
Допустим, мы захотели произвольным образом изменять стороны фигуры и определили для этого соответствующие методы в обоих классах:

```
class Rectangle {
  protected:
    int a_, b_;

  public:
    Rectangle(int a, int b)
        : a_(a)
        , b_(b)
    {}
```

```
    virtual int GetArea() const {
        return a_ * b_;
    }

    virtual void SetWidth(int x) {
        a_ = x;
    }

    virtual void SetHeight(int x) {
        b_ = x;
    }
};

class Square : public Rectangle {
  public:
    Square(int a)
        : Rectangle(a, a)
    {}

    int GetArea() const override {
        return a_ * a_;
    }

    void SetWidth(int x) override {
        a_ = b_ = x;
    }

    void SetHeight(int x) override {
        b_ = a_ = x;
    }
};
```

Логично, что длины сторон прямоугольника мы можем изменять независимо, а квадрата — нет. Именно в этот момент и нарушается принцип подстановки Лисков. Если мы попытаемся завести функцию, которая работает с площадью прямоугольника:

```
void ManipulateArea(Rectangle& rect) {
    rect.SetWidth(5);
    rect.SetHeight(10);
    assert(rect.GetArea() == 50);
}
```

то мы не сможем обеспечить корректную обработку класса `Square`, поскольку его площадь в этом случае будет составлять 100, а не 50 единиц.

Мораль в том, что прямоугольник и квадрат нельзя связать наследованием. Этот пример наглядно иллюстрирует, что не все вещи, работающие в житейской логике, будут работать в ООП, и что к организации иерархии классов следует подходить осознанно и предусмотрительно.

IV. **Interface Segregation Principle,**

или **принцип разделения интерфейсов**.

> Клиент не должен зависеть от интерфейсов, которые он *не* использует. Больши́е интерфейсы следует разбивать на несколько меньших, чтобы обеспечить гибкость взаимодействия и избежать ненужных зависимостей.

Допустим, у нас есть общий интерфейс класса «робот» и производный класс «дрон»:

```
class Robot {
    virtual void Walk() = 0;
    virtual void Speak() = 0;
    virtual void Fly() = 0;
    virtual ~Robot() = default;
};

class Drone : public Robot {
    void Walk() {
        // ?
    }

    void Speak() {
        // ?
    }

    void Fly() {
        // ...
    }
};
```

Несмотря на то, что дроны обычно не умеют ни ходить, ни говорить, в такой архитектуре `Drone` обязан переопределять эти методы, хотя из всего интерфейса ему необходима только логика полёта.

Решением здесь будет разбить интерфейс класса `Robot` на несколько более и наследовать (возможно, множественно) только те логические части, которые нужны конкретному классу:

```
class IWalk {
    virtual void Walk() = 0;
    virtual ~IWalk() = default;
};

class ISpeak {
    virtual void Speak() = 0;
    virtual ~ISpeak() = default;
};

class IFly {
    virtual void Fly() = 0;
    virtual ~IFly() = default;
};

class Drone : public IFly {
    void Fly() override {
        // ...
    }
};
```

V. **Dependency Inversion Principle,**

или **принцип инверсии зависимостей**.

> Зависимости должны относиться к интерфейсам, а не к конкретным классам. Интерфейсы не должны зависеть от реализации, а вот реализация должна зависеть от интерфейсов.

Пусть, например, мы работаем с базами данных, и связываем данные пользователя с классом `MyDatabase`:

```
class MyDatabase {
  public:
    void SaveData(const User& user);
};

class UserService {
  private:
    MyDatabase database_;

  public:
    void AddUser(const User& user) {
        database_.SaveData(user);
        // ...
```

```
    }
};
```

Это нарушает принцип инверсии зависимостей, поскольку в этом случае `UserService` намертво привязан к реализации `MyDatabase`. Если мы хотим перейти на другую базу данных, нам необходимо будет полностью изменять реализацию класса `UserService`.

Решением будет создать «прокладку» в виде абстрактного интерфейса базы данных:

```
class Database {
  public:
    virtual void SaveData(const User& user) = 0;
};

class MyDatabase : public Database {
  public:
    void SaveData(const User& user) override {
        // ...
    }
};

class UserService {
  private:
    Database& database_;

  public:
    UserService(Database& database)
        : database_(database)
    {}

    void AddUser(const User& user) {
        database_.SaveData(user);
    }
};
```

GRASP

GRASP — аббревиатура от выражения General Responsibility Assignment Software Patterns, что в переводе означает «общие шаблоны распределения ответственностей».

GRASP предлагает девять принципов распределения ролей при проектировании объектно-ориентированных систем и свойства объектов, которыми они должны обладать,

чтобы эти роли гармонично исполнять. Они не привносят чего-то принципиально нового, но документируют проверенные временем принципы объектно-ориентированного анализа. Как и в случае с SOLID, не следует воспринимать их как священный грааль.

Примеры в этом блоке практически не будут представлены, поскольку GRASP достаточно абстрактен и вместе с этим достаточно понятен, чтобы это было позволительно.

Итак, GRASP выделяет следующие девять шаблонов:

I. **Информационный эксперт**

> Если объект обладает бо̀льшей частью информации, необходимой для решения задачи, именно ему эту задачу и следует поручить.

Это самый очевидный и важный шаблон из девяти, позволяющий локализовать ответственность, что приводит к укреплению инкапсуляции и ослаблению зависимостей между объектами.

II. **Создатель**

> Классу `B` следует поручить создание объектов класса `A`, если верно хотя бы одно (больше — лучше) из следующего:
>
> - `B` содержит объекты класса `A`;
> - `B` записывает объекты класса `A`, т.е. каким-либо образом фиксирует факт их существования;
> - `B` активно использует объекты класса `A`;
> - `B` обладает данными для инициализации `A`.

Практически говоря, шаблон *Создатель* — это интерпретация шаблона *Информационный эксперт* в контексте создания объектов. Он позволяет не создавать искусственные фабрики там, где это не нужно, хотя в этом случае стоит быть аккуратным в вопросе соблюдения принципа единственной ответственности.

III. **Контроллер**

> Паттерн *контроллер* назначает классу обязанность за обработку внешних событий и координацию работы системы. Контроллер не относится к пользовательскому интерфейсу и при этом не выполняет самостоятельной работы, делегируя её остальным компонентам программы.

Контроллер является «прослойкой» между UI и бизнес-логикой программы. Он представляет собой систему, подсистему, корневой объект или отдельное устройство, и может отвечать как за один, так и за несколько сценариев использования. На этом

основании выделяют *фасадные контроллеры* (вся подсистема) и *контроллеры сценария* (конкретный сценарий).

Основное назначение контроллера — объединение интерфейсов.

IV. **Слабая связанность (Low Coupling)**

> *Связанность* — мера взаимной зависимости модулей.
>
> Сильная связанность рассматривается как серьёзный недостаток, поскольку затрудняет понимание логики модулей, их модификацию, автономное тестирование и переиспользование. Слабая связанность же, наоборот, является признаком хорошо спроектированной системы.

V. **Высокая сплоченность (High Cohesion)**

> *Сплоченность* — мера взаимной зависимости элементов внутри одного модуля.
>
> Высокая сплоченность класса означает, что его элементы тесно связаны и сфокусированы на выполнении одной задачи. Грамотное разделение программы на классы и подсистемы стимулирует высокую сплоченность, которая, в свою очередь, понижает связанность.
>
> Слабая сплоченность означает, что элементы модуля предназначены для решения слишком большого числа разнородных задач. Такие модули трудно понять, переиспользовать и поддерживать как единое целое.

VI. **Полиморфизм**

> Обязанность по определению вариаций поведения в зависимости от типа назначается непосредственно типу, для которого это поведение актуально. Это достигается при помощи полиморфных операций.

Этот принцип был подорбно разобран в разделе *Полиморфизм*.

VII. **Чистая выдумка (Pure Fabrication)**

> *Чистой выдумкой* называется класс, который не отвечает ни за какой концепт в предметной области проблемы, но его использование уменьшает связанность, увеличивает сплочённость и упрощает переиспользование. В парадигме предметно-ориентированного программирования (*Domain-Driven Design*) чистые выдумки носят называние *сервис*.

Этот шаблон описывает искуственно созданные синтетические сущности, которые не возникают естественным образом при решении задачи, но значительно упрощают код,

беря на себя часть технических обязанностей других классов.

VIII. **Перенаправление (Indirection)**

> Паттерн *Перенаправление* поддерживает слабую связанность и высокий потенциал переиспользования в ситуациях, когда прямая связь между двумя классами неизбежна. В этом случае ответственность за взаимодействие объектов следует назначить промежуточному объекту-посреднику.

Это не самый очевидный принцип, поэтому здесь всё же стоит обратиться к примеру. Пусть мы хотим реализовать логику включения лампы по нажатию кнопки. Это можно было бы сделать напрямую:

```
class Lamp {
  private:
    bool turned_on_ = false;

  public:
    bool TurnOn() {
        turned_on_ = true;
    }
};

class Button {
  private:
    Lamp lamp;

  public:
    void Click() {
        lamp.TurnOn();
    }
};
```

В такой реализации интерфейсы сильно связаны: если мы захотим, например, переназначить нажатие кнопки на запуск ядерных боеголовок вместо включения лампочки, нам придется целиком переписывать её класс.

Решением будет привнести в цепь взаимодействий посредника в виде абстрактного класса «команда»:

```
class Lamp  {
    // ...
};
```

```
class Command {
  public:
    virtual void Execute() = 0;
};

class Button {
  private:
    Command& cmd_;

  public:
    Button(Command& cmd)
        : cmd_(cmd)
    {}

    void Click() {
        cmd_.Execute();
    }
};
```

Как было отмечено, это решение значительно ослабляет связанность системы и повышает коэффициент переиспользования кода.

IX. Устойчивость к изменениям (Protected Variations)

> Шаблон *Устойчивость к изменениям* защищает одни элемент системы от изменений вдругих путём вынесения взаимодействия в фиксированный интерфейс, через который и только через который оно осуществляется.

Этот шаблон подразумевает определение потенциально нестабильных участков с предсказуемыми изменениями и их «ограждение» стабильными абстракциями. Эти абстракции, впрочем, могут поддерживать полиморфные вариации.

Несколько слов о паттернах

Условно можно обозначить, что принципы SOLID отвечают на вопросы «почему» и «зачем», а шаблоны GRASP — на вопрос «что», и в этом плане они образуют своеобразную иерархию, в которой за более абстрактным следует менее абстрактное.

И на них эта иерархия не заканчивается. На вопрос «как», то есть вопрос реализации всего, что было описано, отвечают шаблоны GoF (*Gang of Four — «банда четырех», в честь авторов*). Это двадцать три порождающих, структурных и поведенческих паттерна программирования, которые предоставляют готовые решения практически всех задач,

которые возникают на этапе проектирования объектно-ориентированных систем. Естественно, что здесь они описаны не будут — по крайней мере, не в ближайшее время ~~(я уже устал итд)~~. Тем не менее, с ними рекомендуется ознакомиться — все материалы по этой теме свободно доступны в сети Интернет.

Правило трёх

> Если в классе был объявлен хотя бы один из трёх следующих специальных методов, то необходимо объявить и оставшиеся:
>
> 1. Деструктор
> 2. Конструктор копирования
> 3. Оператор копирующего присваивания
>
> Иногда также встречается и обратная формулировка: если удалён хотя бы один из методов, необходимо удалить все.

С появлением семантики перемещения в C++11 к этому правилу добавляются также конструкторы перемещения и оператор перемещающего присваивания, расширяя его до *правила пяти*, однако их рассмотрение выходит за рамки текущей темы.

Мотивация у этой идеи крайне простая: если нам понадобился пользовательский деструктор, то почти гарантированно наш класс управляет ресурсом, который не очищается автоматически. В этом случае сгенерированные компилятором конструкторы копирования и присваивания не будут корректно с этим ресурсом обращаться, и нам необходимо самостоятельно переопределить эту логику.

Исходя из принципа единственной ответственности, классы, для которых определен хотя бы один из указанных методов, должны отвечать только за управление памятью. Это наблюдение приводит к формулировке **правила нуля**: все остальные категории классов *не должны* содержать ни одного из отмеченных специальных методов.

Идиомы

Идиома — более узкое понятие, чем паттерн. Она решает менее масштабную, обычно низкоуровневую проблему, учитывая специфику конкретного языка программирования. В C++ идиомы чаще всего направлены на безопасность и эффективность, позволяя, например, избегать тяжело обнаруживаемых утечек памяти или по максимуму использовать оптимизирующие возможности компилятора. В этом небольшом блоке

будет обзорно рассмотрено несколько полезных идиом, прямо или косвенно связанных с ООП.

pImpl (Pointer to implementation)

Идиома pImpl (*указатель на реализацию*) позволяет вынести реализацию класса за пределы его тела, используя для этого отдельную структуру и указатель на неё:

```
class MyClass {
  private:
    struct Impl;
    Impl* pImpl;

  public:
    void foo();
};

struct MyClass::Impl {
    void foo() {
        std::cout << "I'm a function!" << std::endl;
    }
};

void MyClass::foo() {
    pImpl->foo();
}

int main() {
    MyClass obj;
    obj.foo();
}
```

Главное преимущество, которое даёт эта идиома — нам необходимо перекомпилировать код класса лишь в том случае, когда изменяется его интерфейс, а всё, что касается внутренней реализации, теперь компилируется независимо. Это не только ускоряет компиляцию, но и позволяет поддерживать совместимость в определенных сценариях. С другой стороны, эта идиома провоцирует определенные накладные расходы, связанные со временем работы, динамическим выделением памяти, дополнительными файлами и общим усложнением кода.

Non-Virtual Interface

Проверка пред- и постусловий — признанный и полезный приём ООП, особенно на этапе разработки. Она позволяет гарантировать, что инварианты как иерархии классов, так и

абстракции в целом не будут нарушены ни в какой точке.

Если заставить программиста проверять их в каждом производном классе, коих могут быть десятки, он рано или поздно ошибётся. Для обеспечения согласованности и простоты поддержки таких услових их в идеале следует сосредотачивать в одном месте — потенциально, в базовом классе.

Идиома *Non-Virtual Interface* позволяет переопределить только часть базового класса. Она тесно связана с принципами, которые сформулировал Герб Саттер:

1. Отдавайте предпочтение невиртуальным интерфейсам (*прим.* в прямом смысле, а не в смысле идиомы).
2. Отдавайте предпочтение приватным виртуальным функциям.
3. Делайте виртуальную функцию защищённой только в том случае, если производным классам необходимо вызывать её базовую версию.
4. Деструктор базового класса должен быть либо публичным и виртуальным, либо защищённым и невиртуальным.

Следуя им, мы приходим к примерно следующему коду:

```
class Base {
  public:
    void Work() {
        PreWork();
        WorkImpl();
        PostWork();
    }

  protected:
    virtual void WorkImpl() = 0;
};

class Derived : public Base {
  protected:
    void WorkImpl() override {
        cout << "Doing work in Derived" << std::endl;
    }
};
```

Здесь полностью сохраняется логика виртуальных методов, однако вместе с этим часть логики базового класса остаётся неизменной для любого наследника, чего мы и хотели добиться.

Virtual Friend Function

Рассмотрим так горячо полюбившуюся нам иерархию фигур:

```
class Figure {
  public:
    friend std::ostream& operator<<(std::ostream& out, const Figure& f) {
        out << "I'm just a figure!" << std::endl;
        return out;
    }
};
```

Хотя желание переопределить оператор вывода для классов-наследников, раз уж мы реализовали его для базового класса, кажется более чем логичным, непосредственно сделать это невозможно, поскольку он не является виртуальным методом.

Идиома *Virtual Friend Function* позволяет реализовать полиморфное поведение для свободных функций. Решение состоит в том, чтобы использовать такую функцию как обёртку, которая будет вызывать скрытый вирутальный метод:

```
class Figure {
  protected:
    virtual void PrintImpl(std::ostream& out) const {
        out << "I'm just a figure!" << std::endl;
    }

  public:
    friend std::ostream& operator<<(std::ostream& out, const Figure& f) {
        f.PrintImpl(out);
        return out;
    }
};

class Triangle : public Figure {
  protected:
    void PrintImpl(std::ostream& out) const override {
        out << "I'm a triangle!" << std::endl;
    }
};

int main() {
    Triangle tri;
    Figure& tri_ref = tri;
    std::cout << tri_ref;
}
```

Источники

1. А. П. Хвастунов — Лекции по основам программирования на C++, 1 семестр, 2025
2. Википедия — Объектно-ориентированное программирование (https://ru.wikipedia.org/wiki/Объектно-ориентированное_программирование)
3. Сергей Бобровский — История объектно-ориентированного программирования (https://www.computer-museum.ru/histsoft/oophist.htm)
4. cppreference.com — Classes (https://en.cppreference.com/w/cpp/language/classes.html)
5. cppreference.com — Function declaration (https://en.cppreference.com/w/cpp/language/function.html)
6. cppreference.com — User-defined conversion functions (https://en.cppreference.com/w/cpp/language/cast_operator.html)
7. cppreference.com — Derived classes (https://en.cppreference.com/w/cpp/language/derived_class.html)
8. Stanley B. Lippman — Inside the C++ Object Model, §3.4 Inheritance and the Data Member (https://raw.githubusercontent.com/YuxuanLing/books/master/c/C%2B%2B_En_Inside.The.C%2B%2B.Object.Model.pdf)
9. Википедия — Полиморфизм (информатика) (https://ru.wikipedia.org/wiki/Полиморфизм_(информатика))
10. cppreference.com — Virtual function specifier (https://en.cppreference.com/w/cpp/language/virtual.html)
11. Itanium C++ ABI, §2.5 Virtual Table Layout (https://itanium-cxx-abi.github.io/cxx-abi/abi.html)
12. atromone — Принципы SOLID, только понятно (https://habr.com/ru/articles/811305/)
13. Wikipedia — SOLID (https://en.wikipedia.org/wiki/SOLID)
14. cpprefernece.com — The rule of three (https://en.cppreference.com/w/cpp/language/rule_of_three.html)
15. Wikipedia — GRASP (object-oriented design) (https://en.wikipedia.org/wiki/GRASP_(object-oriented_design))
16. cppreference.com — PImpl (https://en.cppreference.com/w/cpp/language/pimpl.html)
17. Wikibooks — Non-Virtual Interface (https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Non-Virtual_Interface)

---

1. См. раздел ODR-use: https://en.cppreference.com/w/cpp/language/definition.html ↩
2. На эту тему есть небольшой хороший блог (https://bryanpendleton.blogspot.com/2011/03/tail-padding-reuse-in-gcc.html) с примерами. Советую поиграться с этим и посмотреть, как разные компиляторы

размещают данные при наследовании и как на это влияет... что угодно, потому что на это влияет почти всё подряд. ↩