

# SOLID

SOLID (Single responsibility, Open–closed, Liskov substitution, Interface segregation и Dependency inversion) в программировании — акроним, для первых пяти принципов, названных Робертом Мартином в начале 2000-х, которые означали 5 основных принципов проектирования.

По своей сути ООП строится на принципах SOLID.

Главная цель этих принципов — повысить гибкость вашей архитектуры, уменьшить связанность между её компонентами и облегчить повторное использование кода.

Но, соблюдение этих принципов имеет свою цену, которая выражается в усложнении кода программы.

В реальной жизни, нет таких решений, в которых бы соблюдались все эти принципы сразу.

## SRP

**Single Responsibility Principle** гласит:

У программной сущности должна быть только *одна причина для изменения*.

Грубо говоря, если у нас есть две причины для изменения класса, нам нужно разделить функциональность на два класса.

*Зачем нам следовать этому принципу?*

Принцип единственной ответственности предназначен для борьбы со сложностью. Если класс делает слишком много вещей сразу, то приходится изменять его каждый раз, когда одна из этих вещей изменяется, при этом есть риск разрушить остальные части класса. Хорошо иметь возможность сосредоточиться на сложных аспектах системы по отдельности.

Как проверить удовлетворение SRP:

1. Задайте себе вопрос — что делает этот класс/метод/модуль/сервис. Вы должны ответить на него простым определением. Если в вашем

объяснении есть союз, например "класс получает данные из БД *и* составляет на их основе графики", то, очевидно, SRP нарушается.

2. Фикс некоторого бага или добавление новой фичи затрагивает минимальное количество файлов/классов. В идеале — один.

## ОСР

**Open/Closed Principle** гласит:

программные сущности должны быть *открыты для расширения, но закрыты для модификации*.

Суть заключается в том, что мы не должны модифицировать код класса, так как для остальной системы поведение станет неожиданным, но в то же время мы можем добавлять новый функционал.

Самый частый пример нарушения принципа открытости/закрытости — использование конкретных объектов/переменных без абстракций.

## LSP

**Liskov Substitution Principle** гласит:

функции/методы, которые используют базовый тип, должны иметь *возможность использовать подтипы базового типа*, не зная об этом.

Более простыми словами - производный класс должен быть взаимозаменяем с родительским классом. Наследники класса могут использоваться везде, где используется базовый класс. Без всяких неожиданностей.

Если нужно добавить какое-то ограничение в переопределенный метод, и этого ограничения не существует в базовой реализации, то, нарушается принцип подстановки Liskov. Наследуемый объект может заменить родительское *пред-условие* на такое же или более слабое(контрвариантность) и родительское *пост-условие* на такое же или более сильное(ковариантность).

*Пред-условия* — это требования подпрограммы, т.е. то, что обязано быть истинным для выполнения подпрограммы. Если данные предусловия нарушены, то подпрограмма не должна вызываться ни в коем случае. Вся

ответственность за передачу «правильных» данных лежит на вызывающей программе.

*Пост-условия* выражают состояния «окружающего мира» на момент выполнения подпрограммы. Т.е. это условия, которые гарантируются самой подпрограммой. Кроме того, наличие постусловия в подпрограмме гарантирует ее завершение (т.е. не будет бесконечного цикла, например).

Говорят, что условие P1 *сильнее*, чем P2, а P2 *слабее*, чем P1, если выполнение условия *P1 влечет за собой выполнение условия P2*, но они не эквивалентны

## ISP

**Interface Segregation Principle** гласит:

Программные сущности не должны зависеть от частей интерфейса, которые они не используют (и знать о них тоже не должны).

Более простым языком, клиенты не должны зависеть от методов, которые они не используют.

Большинство объектных языков программирования позволяют классам реализовывать сразу несколько интерфейсов, поэтому нет нужды снабжать ваш интерфейс большим количеством различных поведений, чем он того требует.

Всегда можно присвоить классу сразу несколько интерфейсов поменьше.

## DIP

**Dependency Inversion Principle** гласит:

1. модули верхних уровней не должны импортировать сущности из модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
2. абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Классы нижнего уровня реализуют базовые операции вроде работы с диском, передачи данных по сети, подключения к базе данных и прочее. Классы высокого уровня содержат сложную бизнес-логику программы,

которая опирается на классы низкого уровня для осуществления более простых операций.

Если каждое изменение в низкоуровневом классе может затронуть классы бизнес-логики, которые его используют. Вопрос только в том, как класс, зависимый от другого класса, может не быть зависимым от изменений в последнем? Всё просто — нужно заставить низкоуровневый класс следовать интерфейсу, а зависимый класс завязать на этом самом интерфейсе. В таком случае высокоуровневый класс будет зависеть не от конкретной реализации, а от интерфейса, а низкоуровневый класс будет обязан следовать "контракту" интерфейса.

## Паттерны

Паттерны описывают **примерный каркас** решения для распространённых проблем. Под примерным каркасом может подразумеваться связь классов и/или их внутреннее устройство. Здесь стоит понимать, что использовать паттерны нужно не всегда, а если поняли, что в этом есть необходимость, то помните, что решение паттерна не универсальное и его нужно подгонять под свой конкретный случай.

Лучший **сайт по паттернам**(нужен прокси/впн).

## Порождающие

*Порождающие паттерны* - отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.

### Singleton

**Singleton**(одиночка) — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Чтобы реализовать паттерн, необходимо:

1. Добавить приватное статическое свойство, хранящее объект
2. Добавить статический метод для получения объекта
3. Сделать конструктор, операторы присваивания и копирования приватными

Лучшая реализация(можно и через указатели, но там нужно управлять освобождением памяти на указатель). Проще запомнить эту:

```
class Singleton
{
public:
    static Singleton& getInstance()
    {
        static Singleton instance;
        return instance;
    }

private:
    Singleton() { } // конструктор недоступен
    ~Singleton() { } // и деструктор
    // необходимо также запретить копирование
    Singleton(Singleton const&); // реализация не нужна
    Singleton& operator= (Singleton const&); // и тут
};

int main(int argc, char** argv) {
    //new Singleton(); // ошибка
    Singleton& instance = Singleton::getInstance();
    return 0;
}
```

Если возник вопрос, как работает строка `static Singleton instance;`, то при первом вызове `getInstance()` для инициализации статического объекта `instance` будет вызван конструктор класса. Механизм называется *отложенной/ленивой инициализацией*.

Статическая функция-член `getInstance()` возвращает не указатель, а *ссылку* на этот объект.

Плюс реализации заключается в том, что стандарт языка программирования C++ гарантирует автоматическое уничтожение статических объектов при завершении программы, то есть мы не должны думать об этом и освобождать память вручную.

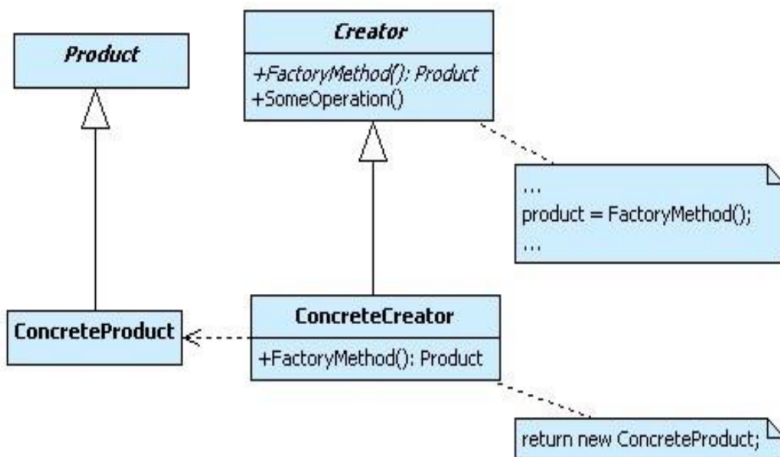
У реализации Мэйерса(так называется реализация класса выше) есть недостатки: сложности создания объектов производных классов и невозможность безопасного доступа нескольких клиентов к единственному объекту в многопоточной среде.

## Factory Method

**Factory Method** (также известен как Виртуальный конструктор, Фабричный метод) — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Создающие классы называются *производителями*, производимые классы - *продуктами*.

Реализовать данный паттерн крайне просто — в родительском классе производителя создаём метод (который должен возвращать некий продукт) и завязываем возвращаемый тип на интерфейс продуктов. В каждом из подклассов фабрики создаётся конкретный тип объекта. Схема выглядит следующим образом:



## Abstract Factory

**Abstract Factory** (Абстрактная фабрика) — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Реализация заключается в следующих шагах:

1. Делим все продукты по семействам, а семейства по их типам.

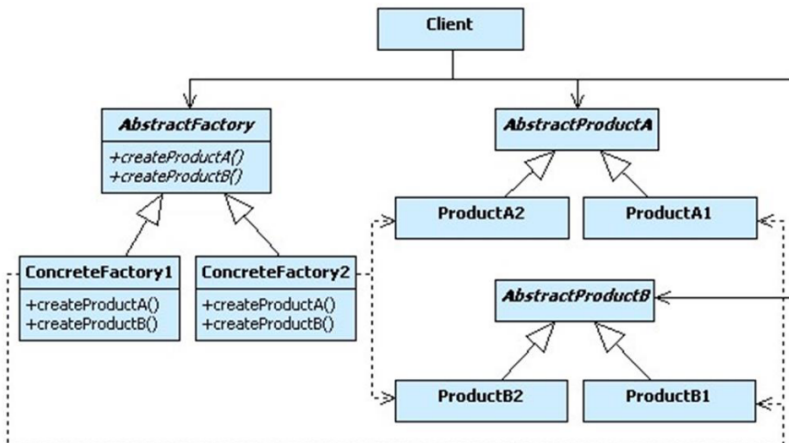
Например, нужно создавать геометрические фигуры разного цвета.

Здесь фигуры (квадраты, круги и тд) - семейства, их цвета (синий, красный и тд) - типы. Или наоборот, всё зависит от семантики. Проще

определить, задумавшись о том, что должны производить фабрики - фигуры или цвета. В нашем примере, очевидно, фигуры.

2. Сведите все типы продуктов к общим интерфейсам. В примере выше интерфейсами будут `BlueColorInterface`, `RedColorInterface`, которые в свою очередь наследуют `ColorInterface`.
3. Определите интерфейс абстрактной фабрики. Он должен иметь фабричные методы для создания каждого из типов продуктов. То есть `AbstractFactory` будет иметь виртуальные методы `createBlue()`, `createRed()` и тд
4. Создайте классы конкретных фабрик, реализовав интерфейс абстрактной фабрики. Этих классов должно быть столько же, сколько и вариаций семейств продуктов. В приведённом примере нам потребуется создать `SquareFactory`, `CircleFactory` и тд.

Схема:



## Поведенческие

**Поведенческие** паттерны описывают как организовать взаимодействие между объектами программы.

## Strategy

**Strategy**(стратегия) определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы

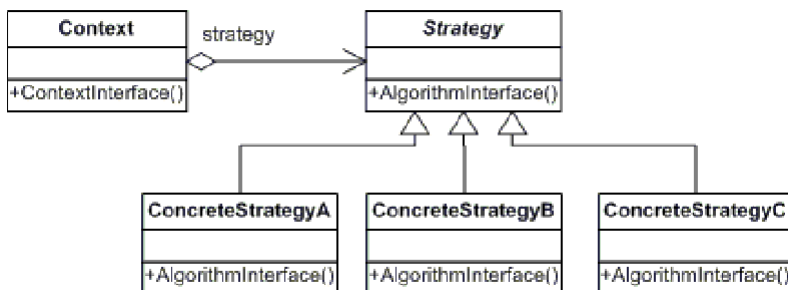
можно взаимозаменять прямо во время исполнения программы.

Реализация крайне простая: есть некий клиент(контекст стратегии), который через **агрегацию** принимает некий объект алгоритма(через общий интерфейс алгоритмов) и в некотором методе использует его.

Если более подробно, то шаги реализации следующие:

1. Определите алгоритм, который **подвержен частым изменениям**. Также подойдёт алгоритм, имеющий несколько вариаций, которые **выбираются во время выполнения программы**.
2. Создайте интерфейс стратегий, описывающий этот алгоритм. Он должен быть общим для всех вариантов алгоритма.
3. Реализуйте интерфейс алгоритма в требуемых вариациях.
4. В классе контекста создайте поле для хранения ссылки на текущий объект-стратегию и возможность "прокидывать" этот объект в класс.

Схема(обращаем внимание, что определённая стратегия может быть передана в контекст как через конструктор, так и через специальный метод, агрегация ничего не говорит про то, как это должно происходить - решайте исходя из задачи):



## Observer

**Observer**(наблюдатель) — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

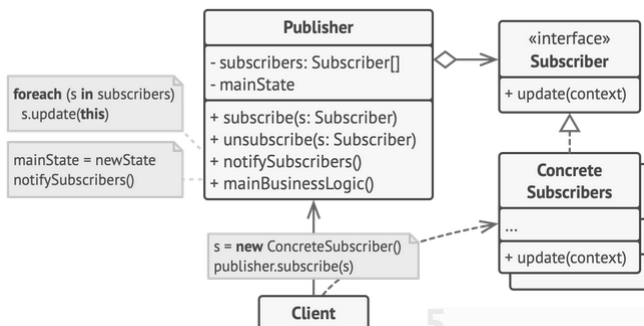
Шаги реализации:

1. Делим систему на два класса: издатель и подписчики(наблюдатели).



2. Подписчиков реализуем через общий интерфейс, в котором лишь метод для обновления. Каждый из подписчиков реагирует на обновление по-своему.
3. Классу издателя даруем свойство, где будут храниться текущие подписчики, метод для добавления/удаления подписчиков(через интерфейс) и метод для оповещения всех подписчиков.
4. Добавляем клиент, который наследует издателя и при определённых событиях оповещает подписчиков.

Схема:



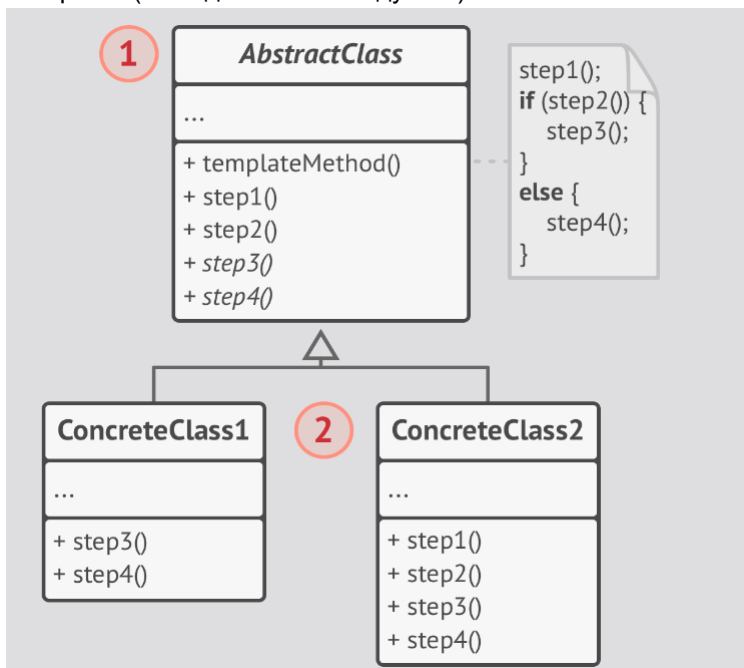
## Template Method

**Template Method**(шаблонный метод) определяет скелет алгоритма, перекидывая ответственность за некоторые его шаги на подклассы.

Чтобы реализовать паттерн, необходимо выделить некий алгоритм, который состоит из шагов. В базовом классе определяем метод алгоритма, который вызывает виртуальные методы. Последние могут быть реализованы по-разному в дочерних классах, но сам каркас(порядок и условия вызова виртуальных методов) не меняется. Так как шаги *могут* переопределяться, но это не обязательно, можно некоторые или даже все виртуальные методы реализовать в абстрактном классе.

Проще всего рассмотреть схему. Из неё видно, что в базовом классе есть некий шаблон для вызова виртуальных методов. Наследующие же классы переопределяют все или некоторые шаги алгоритма и имеют метод

алгоритма(он ведь тоже наследуется):



## Inversion of Control

**Inversion of Control (инверсия управления)** — это некий абстрактный принцип, набор рекомендаций для написания слабо связанного кода. Суть которого в том, что каждый компонент системы должен быть как можно более изолированным от других, не полагаясь в своей работе на детали конкретной реализации других компонентов.

Если *кратко*, то:

1. Всё построено на принципе IoC, который является очень *широким определением*, как всё должно быть, но не говорит как этого добиться.
2. Чтобы избавиться от необходимости получать зависимости для классов вручную(то есть это лишь один из аспектов принципа инверсии контроля), был придуман IoC-container. Это уже *конкретная идея*, как делегировать создание объектов стороннему сервису.
3. Контейнер в свою очередь используется в паттернах, которые имеют разные подходы к использованию ioc-container'a.

Не стоит путать Inversion of Control и **Dependency Inversion Principle**, это не одно и то же. На самом деле, DIP лишь часть IoC, который говорит, как управлять зависимостями между классами, но сама инверсия контроля куда шире и охватывает, например, то же делегирование создания объектов(IoC-container) и предназначена для уменьшения coupling'a в целом, а не только про то, как организовывать классы и интерфейсы для взаимодействия друг с другом.

- В лекциях Андреевой эти вещи считаются одним и тем же, будьте аккуратны

## IoC-container, DI, SL

Принцип используется для создания **IoC-контейнера**. Сам контейнер ответственен за **получение необходимых зависимостей**. Тут речь не только про создание объектов, но и получение инициализированных некоторым значением переменных, и даже Callable(из других языков: нечто, что может быть вызвано как функция; в C++ такого нет, в плюсах Callable - объект, который может быть вызван как функция, то есть просто перегрузка оператора `()`).

Этот контейнер используется в паттернах проектирования **Dependency Injection**(DI, внедрение зависимостей) и **Service Locator**, то есть оба они нужны для получения параметров, притом отличия в паттернах, по большей части, не в том, как они устроены, а в том, **как они используются**. Какой паттерн мы бы ни использовали, мы должны задать правила, по которым контейнер будет знать, как создать тот или иной объект (продвинутые библиотеки/фреймворки в некоторых случаях не требуют даже этого).

- **Service Locator**: в коде есть обращение напрямую из зависимого класса к контейнеру, чтобы получить необходимый объект/переменную/значение. Пример: `container.get("SomeClass")`. Можно также указывать интерфейс, если задано правило, которое говорит контейнеру, какой конкретно класс необходим.
- **DI**: фреймворк/библиотека сама прокидывает все зависимости в конструктор/через сеттер(так себе вариант, но тоже существует) и мы получаем готовый к работе объект со всеми зависимостями, забывая о трудностях его инициализации.

Зачем вообще нужен этот контейнер и почему бы не создавать всё вручную?

Рассмотрим следующий пример, где мы пытаемся воссоздать настоящую структуру компьютера. Тогда класс компьютера зависит от процессора, видеокарты, ..., материнской платы, материнская плата в свою очередь зависит от чипсета, всяких контроллеров, и т.д.

Создание этого класса выглядело бы примерно так:

```
new Computer(new CPU(...), new GPU(...), ..., new Motherboard(new
Chipset(...), new Controller1(...), new Controller2))
```

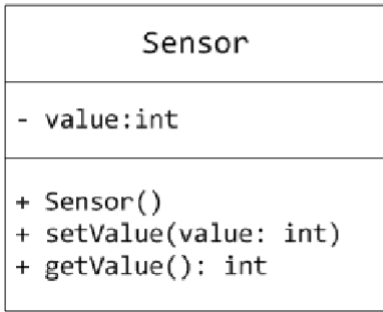
## UML

**UML** – унифицированный язык моделирования (Unified Modeling Language) – это система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования. Если коротко, мы используем UML, чтобы отобразить сигнатуры классов и связи между последними без кода.









Основные правила UML языка:

1. Делим каждый класс, и, соответственно, UML схему этого класса на три части, идущих последовательно:
  1. Имя класса(*абстрактные* обозначаются курсивом)
  2. Свойства класса
  3. Методы класса(*абстрактные* обозначаются курсивом, *статические* обозначаются подчёркиванием)
- Модификаторы доступа у свойств и методов класса могут быть обозначены:
  - — (private)
  - # (protected)
  - + (public)

Пример класса на UML-диаграмме:



2. Связи между классами обозначаются стрелкой:

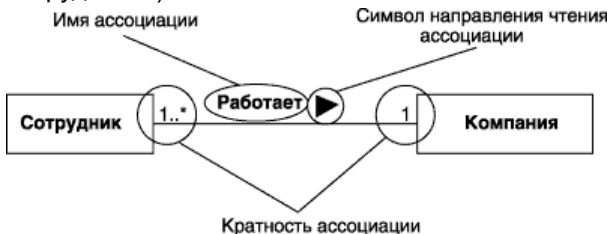
1. **Зависимость** (зависимый  независимый) показывает, что один класс зависит от другого, чтобы сказать нам о том, что при изменении независимого класса, зависимый от него может поменять своё поведение. Самая распространённая связь, в качестве примера — класс  использует в своём методе метод класса , то есть  зависит от .
2. **Ассоциации** показывают, что объекты одного класса связаны с объектами другого класса. Существует три частных случая ассоциации:
  1. **Агрегация** (часть  целое) — объекты составляют часть объекта-контейнера, который их содержит, но они не зависят от него, то есть *объект-контейнер не управляет их временем жизни* (если контейнер будет уничтожен, то его содержимое — нет).
  2. **Композиция** (часть  целое) — это по сути включение класса, внутрь другого класса с помощью **создания объекта внутри этого класса**. На содержащийся объект может ссылаться только содержащий его объект-контейнер и первый должен быть удален при удалении объекта-контейнера, то есть при композиции *объект-контейнер управляет временем жизни содержащихся объектов*.
  3. **Обобщение** (дочерний класс/подкласс  родительский класс/суперклассом/базовый класс) — выражается наследованием, то есть в этой связи дочерний

класс является более конкретным(уточнением/расширением) по отношению к родительскому классу.

4. **Реализация**(реализующий класс -----> интерфейс/ абстрактный класс)

- Для композиции и агрегации можно указать количество объектов с каждой из сторон:
  - 1 — ровно один объект;
  - 0..1 — от нуля до одного объекта;
  - 0..\* — ноль или больше объектов;
  - \* — много объектов.

Пример(сотрудник работает только в одной компании, в каждой компании может быть от одного и больше сотрудников):



## C++

## Память

Память в плюсах бывает двух типов:

- **Статическая** — память выделяется только один раз во время компиляции. Размер выделенной памяти есть фиксированным и неизменным до конца выполнения программы. Примером такого выделения может служить объявление массива из 10 целых чисел(`int x[10]`)
- **Динамическая** — память можно выделять и освобождать в рантайме. Например, можно выделить память для массива, размер которого заведомо неизвестен. В этом случае используется комбинация операторов `new` и `delete`. Оператор `new` выделяет память для переменной (массива) в специальной области памяти,

которая называется «куча» (`heap`). Оператор `delete` освобождает выделенную память. Каждому оператору `new` должен соответствовать свой оператор `delete`.

## Ссылки и указатели

Ссылки и указатели предназначены для одного и того же действия — обращения к существующему в памяти объекту, но отличаются по возможностям, где указатели имеют куда больший функционал.

Кратко базовые действия с ссылками и указателями:

```
int x;  
int *y = &x; // От любой переменной можно взять адрес при помощи  
операции взятия адреса "&". Эта операция возвращает указатель  
int z = *y; // Указатель можно разыменовать при помощи операции  
разыменовывания "*". Это операция возвращает тот объект, на который  
указывает указатель
```

## Ссылки

**Ссылки** — это самостоятельный тип данных, который позволяет обращаться к уже существующему в памяти объекту под другим именем. Рассмотрим пример:

```
int x = 42;  
int& ref = x; // ссылка на x  
  
++x;  
  
std::cout << ref << "\n"; // 43
```

В частности, ссылка позволяет дать дополнительное имя переменной и передавать в функции сами переменные, а не значения переменных. Что нам это даёт? Это позволяет менять значение переменной в функции не только в её области видимости, но и в той, откуда она пришла:

```
void incrementValue(int &x) {  
    x++;  
}  
  
int main()  
{
```

```
int x = 10;
incrementValue(x);
cout << x; // 11
}
```

Что стоит знать про ссылки:

1. Само значение ссылки — **константа**, это просто адрес памяти переменной, на которую мы ссылаемся, поэтому объявлять её `const` не нужно
2. Ссылка должна быть **проинициализирована сразу** в момент объявления.
3. Ссылка привязана к одному и тому же объекту со своего рождения, **переназначить её нельзя**.

## Указатели

**Указатели** представляют собой объекты, значением которых служат адреса других объектов (переменных, констант, указателей) или функций. Как и ссылки, указатели применяются для косвенного доступа к объекту. Однако в отличие от ссылок указатели обладают большими возможностями. Например, их можно переназначать, получать по ним. Указатели можно явным образом инициализировать нулем, используя специальную константу `nullptr`.

С помощью оператора `&` можно получить адрес некоторого объекта, например, адрес переменной. Затем этот адрес можно присвоить указателю:

```
int number {25};
int *pnumber {&number}; // указатель pnumber хранит адрес переменной
number
```

Что важно, переменная `number` имеет тип `int`, и указатель, который указывает на ее адрес, тоже имеет тип `int`. То есть должно быть соответствие по типу. Однако также можно использовать ключевое слово `auto` при объявлении указателя.

**Разыменованием** указателя называют обращение к значению по адресу, хранимому указателем. Так как указатель хранит адрес, то мы можем по



этому адресу получить хранящееся там значение, то есть значение переменной `number`. Для этого применяется оператор `*`. Результатом этой операции всегда является объект, на который указывает указатель. И также используя указатель, мы можем менять значение по адресу, который хранится в указателе:

```
int x = 10;
int *px = &x;
*px = 45;
std::cout << "x = " << x << std::endl;    // 45
```

## Умные указатели

Умные указатели - это классы-обертки для обычных указателей C++. Они позволяют забыть о ручном освобождении памяти с помощью `delete`. Сам класс-обертка является локальной переменной, поэтому при выходе из области видимости ресурс (память в куче) автоматически освобождается.

Для использования умных указателей необходимо подключить библиотеку работы с памятью (`#include <memory>`).

Рассмотрим следующие типы указателей:

- `std::unique_ptr` поддерживает единственность указателя на ресурс. Нельзя создать указатель на тот же объект через присвоение:

```
std::unique_ptr<MyClass> c(func()); // указатель на функцию
std::unique_ptr<MyClass> c2 = c; // Ошибка
std::unique_ptr< MyClass > c2 = std::move(c); // так можно. Притом c
становится пустой, а монопольное управление указателем получает c2
```

- `std::shared_ptr` позволяет иметь несколько указателей на один и тот же объект. В этом случае оба умных указателя в равной мере управляют обычным указателем. Освобождение памяти произойдет в момент, когда последний `shared_ptr`, обладающий общим ресурсом, покинет область видимости.
- `std::weak_ptr` с помощью `shared_ptr` можно создать циклические ссылки, однако `weak_ptr` не участвует в подсчете ссылок. Можно думать о `weak_ptr` как об указателе, позволяющим получить временное владение объектом.

Последний тип умных указателей мы не проходили на лекциях

## Массивы

Обычно компилятор преобразует массив в указатели. С помощью указателей можно манипулировать элементами массива, как и с помощью индексов.

Имя массива по сути является адресом его первого элемента.

Соответственно через операцию разыменования мы можем получить значение по этому адресу:

```
int nums[] {1, 2, 3, 4, 5};
std::cout << "nums[0] address: " << nums << std::endl; //
0x1f1ebffe60
std::cout << "nums[0] value: " << *nums << std::endl; // 1
std::cout << "nums[1] value: " << *(nums + 1) << std::endl; // 2
```

Перебрать значения последовательности(в том числе и массива) можно следующим образом:

```
int nums[] {1, 2, 3, 4, 5};

for (auto n : nums) {
    std::cout << n << std::endl;
}
```

Для многомерных массивов:

```
const int rows = 3, columns = 2; // const здесь обязателен, чтобы
массив знал, что эти значения константные
int numbers[rows][columns] { {1, 2}, {3, 4}, {5, 6} };

for(auto &subnumbers : numbers) // внешний массив хранит ссылки на
внутренние
{
    for(int number : subnumbers) // внутренние массивы хранят числа
    {
        std::cout << number << "\t";
    }
    std::cout << std::endl;
}
```

## Динамические массивы

Размер динамического массива можно изменять во время работы программы(вспоминаем, он хранится в **динамической памяти**). Синтаксис в самом простом случае следующий:

```
int size = 10; // обращаем внимание, здесь не константа
int *arr = new int[size];
delete [] arr; // удаляем всю память под массив, а не только
выделенную под указатель
```

Двумерный массив:

```
// динамическое создание двумерного массива вещественных чисел на
десять элементов
float **ptrarray = new float* [2]; // две строки в массиве
for (int count = 0; count < 2; count++)
    ptrarray[count] = new float [5]; // и пять столбцов(в каждой
из строк)
```

## Наследование

В C++ есть несколько типов наследования:

1. **public** — `public` и `protected` данные наследуются без изменения уровня доступа к ним;
2. **protected** — все унаследованные данные становятся `protected`;
3. **private** — все унаследованные данные становятся `private`.

```
class ClassName : public ParentClass1, protected ParentClass2 //
объявление класса
{
};
```

В C++ **конструкторы и деструкторы не наследуются**.

Однако они вызываются, когда дочерний класс инициализирует свой объект.

- При наследовании сначала конструируется базовая часть класса, затем производная, а при разрушении наоборот — сначала вызывается деструктор производного класса, который по окончании своей работы вызывает по цепочке деструктор базового.

```
class Employee : public Person
{
public:
    Employee() { company = "NoCompany"; }
    Employee(std::string inf, int val, std::string cmp) :
Person(inf, val) { // передаём inf и val в конструктор Person
        company = cmp;
    }
};
```

```
Person *p0 = new Employee («Alice Goodwin", 8, "Microsoft");
p0->Display();
delete p0;
```

В этом примере будет вызван конструктор `Person`, но не `Employee`. Деструктор базового класса не может вызвать деструктор производного, потому что он о нем ничего не знает. В итоге часть памяти, выделенная под производный класс, безвозвратно теряется. Если хотим вызов деструктора `Employee`, в первой строке объявляем указатель `auto` или `Employee` или делаем деструктор класса `Person` *виртуальным*

`final` работает как и везде - класс нельзя будет наследовать:

```
class Employee final : public Person
{
};

class X : public Employee // error
{
};
```

## Виртуальные методы

Виртуальные методы - методы, которые могут быть переопределены в наследующих классах. Важно, что они *могут* быть переопределены. Если нет - вызовется метод родительского класса.

Рассмотрим пример наследования(пример, кстати, демонстрирует паттерн *шаблонный метод*):

```
#include <cstdlib>
#include <iostream>
```

```

using std::cout;
using std::endl;

class Cat
{
public:
    void askForFood() const
    {
        speak();
        eat();
    }
    virtual void speak() const { cout << "Meow! "; }
    virtual void eat() const { cout << "**champing*" << endl; }
};

class CheshireCat : public Cat
{
public:
    virtual void speak() const { cout << "WTF?! Where\'s my milk? ="
"; }
};

int main()
{
    Cat * cats[] = { new Cat, new CheshireCat };

    cout << "Ordinary Cat: "; cats[0]->askForFood();
    cout << "Cheshire Cat: "; cats[1]->askForFood();

    delete cats[0]; delete cats[1];
    return EXIT_SUCCESS;
}

```

Здесь конструкция `speak()` в методе `askForFood()` эквивалента `this->speak()`, то есть вызов происходит через указатель, а значит — будет использовано позднее связывание. Вот почему при вызове метода `askForFood()` через указатель на `CheshireCat` мы видим то, что и хотели: механизм виртуальных методов работает исправно даже несмотря на то, что вызов непосредственно виртуального метода происходит внутри другого метода класса.

## Виртуальный деструктор

Основное правило: *если у вас в классе присутствует хотя бы одна виртуальный метод, деструктор также следует сделать*

*виртуальным.*

Однако! Если мы запишем вызов любого виртуального метода в виртуальный деструктор базового класса, будет вызван метод базового класса, независимо от того, переопределён ли метод или нет. Это происходит, потому что при вызове виртуальных методов из деструктора компилятор использует не позднее, а раннее связывание. Все помнят, что конструирование объекта происходит, начиная с базового класса, а разрушение идет в строго обратном порядке. Если же мы захотим внутри деструктора `~Cat()` совершить виртуальный вызов метода `sayGoodbye()`, то фактически попытаемся обратиться к той части объекта, которая уже была разрушена.

## static

Обычно статический метод в классе используется, когда необходимо реализовать некоторое действие, которое относится к классу в целом, а не к конкретному объекту класса. Статические методы не имеют указателя `this` и могут вызываться без создания класса.

- Статические свойства класса инициализируются вне класса, для того чтобы была выделенная память компилятором в глобальном участке:

```
class SomeObject
{
public:
    SomeObject(){ count++;}
    ~SomeObject(){ count--;}
    static unsigned getCount();
private:
    static unsigned count;
};

unsigned SomeObject::count = 0;

unsigned SomeObject::getCount()
{
    return count;
}
```

## const

**const** указывает, что значение переменной является константой, то есть его нельзя изменить.

`const` с указателями работает немного иначе:

1. `int * const p` — указатель после инициализации менять нельзя(присвоить переменной другой указатель нельзя) , однако само число изменить можно.
  2. `int const* p` или `const int* p` — указатель изменить можно, а вот число — нет(как правило, используется вариант объявления `const int`)
  3. `const int* const p` — нельзя поменять ни указатель, ни значение, на которое ссылается указатель
- Во всех случаях, если изменить *переменную*, на которую ссылается указатель, *не через указатель*(а через доступ к ней самой), **ошибки не будет** — значение указателя и переменной изменится!

`const` особенно *удобен*, если нужно передать параметры в функцию/метод, но при этом надо обязательно знать, что *переданный параметр не будет изменен*.

Значения `const` свойств класса задаются один раз и навсегда в конструкторе, а значения `static const` свойств класса типа `enum`, `int`, `char` прямо в объявлении класса.

## В методах

Метод трактует `this` как указатель на текущий объект. Т.е `this` в методе класса `X` будет `X*`.

Но если метод класса объявлен как `const`, то тип `this` будет `const X*`. В таких методах не может быть ничего присвоено переменным класса, которые не объявлены как `static` или как `mutable`.

Также `const`-методы не могут возвращать не `const` ссылки и указатели на свойства класса и не могут вызывать не `const` методы класса.

`const`-функции иногда называют инспекторами (*inspector*), а остальные мутаторами (*mutator*).

Для константного объекта(`Class const ClassVar(2)`) при вызове методов будут вызываться константные методы, если же последние не определены будет ошибка. Если же определены только `const`-методы, но объект создаётся не константный, то будут использоваться `const`-методы.

Притом при объявлении класса таким образом все данные, не объявленные как `const`, начинают трактоваться как `const: int` становится `int const`, `int *` становится `int * const`, `const int *` становится `int const *const` и так далее.

- Конструкторы и деструкторы не могут быть объявлены как `const`.
- Также не бывает `static const` методов.