1. **Что такое парадигма программирования?**

Совокупность идей и понятий, определяющих стиль написания компьютерных программ.
Способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.
Не определяется однозначно языком программирования. Имеют пересечения в деталях.

2. **Что такое идиома программирования?**

Идиома программирования — устойчивый способ выражения некоторой составной конструкции в одном или нескольких языках программирования. Идиома является шаблоном решения задачи, записи алгоритма или структуры данных путём комбинирования встроенных элементов языка. Можно считать самым низкоуровневым шаблоном проектирования, применяемым на стыке проектирования и кодирования. Одна и та же идиома может выглядеть по разному в разных языках, либо в ней может не быть надобности в некоторых из них.

3. **Какую парадигму реализует язык С?**

Мультипарадигменное:
Процедурное программирование
Структурное
ООП

4. **Какую парадигму реализует язык С++?** На С++ можно писать программы в рамках нескольких парадигм программирования.

Мультипарадигменное язык:
Процедурное программирование
(код “в стиле С”)
Объектно-ориентированное программирование
(классы, наследование, виртуальные функции, . . .)
Обобщённое программирование
(шаблоны функций и классов)
Функциональное программирование
(функторы, безымянные функции, замыкания)
Генеративное программирование
(метапрограммирование на шаблонах)

5. **Язык С++ считается низкоуровневым или высокоуровневым?**

Низкоуровневый

6. **Что такое ООП?**

Наиболее распространенная и востребованная парадигма в настоящее время.
При использовании ООП программа представляет собой совокупность взаимодействующих между собой данных – объектов.
Функциональную возможность и структуру объектов задают классы – типы данных, определенные пользователем.

7. **Что подразумевает абстракция в ООП?**

Основная идея – реализация понятия «абстракция».
Сущность произвольной сложности можно рассматривать как единое целое, выделяя только необходимый функционал, не вдаваясь в детали внутреннего построения и функционирования.
Один из основных способов абстракции – использование концепции иерархической классификации (сложные системы разбиваются на более простые фрагменты).

8. **Что такое инкапсуляция?**

Механизм, связывающий вместе код и данные, которыми этот код манипулирует.
Защищает данные от произвольного доступа со стороны внешнего кода.
Основа инкапсуляции – класс.
Позволяет скрыть от пользователя детали реализации класса

9. **Что такое наследование?**

Механизм, с помощью которого один объект приобретает свойства другого объекта.
Позволяет объекту наследовать от своего родителя общие атрибуты, а самому определять только те характеристики, которые делают его уникальным.
Поддерживает понятие иерархической классификации.
Уменьшает дублирование кода, повышает скорость написания программ за счет расширения существующих компонентов.

10. **Что такое полиморфизм?**

Механизм, позволяющий использовать один и тот же интерфейс для общего класса действий.

Один интерфейс – множество реализаций.
Выбор конкретной реализации возлагается на компилятор.

11. Какие существуют виды полиморфизма?

статический - реализуется на этапе компиляции с помощью перегрузки функций и операций
динамический - реализуется во время выполнения программы с помощью механизма виртуальных функций
параметрический - реализация на этапе компиляции с использованием механизма шаблонов

12. В чем отличие компилируемых и интерпретируемых языков программирования?

Компилируемый язык — это такой язык, что программа, будучи скомпилированной, содержит инструкции целевой машины; этот машинный код непонятен людям.
В отличие от компилируемых языков, интерпретируемым для исполнения программы не нужен машинный код; вместо этого программу построчно исполнят интерпретаторы.
Интерпретируемый же язык — это такой, в котором инструкции не исполняются целевой машиной, а считываются и исполняются другой программой (которая обычно написана на языке целевой машины).
Интерпретатор читает исходный код программы и выполняет его. Преобразование исходного кода в бинарный и выполенение выполняется построчно.
Компиляторы же, полностью переобразовывают исходный код программы в бинарный (а не построчно, как в случае с интрепретаторами), который ОС может выполнять самостоятельно. То есть, для запуска программы иметь компилятор нет необходимости.
Языки программирования принято разделять на компилируемые и интерпретируемые в силу типичных различий:

- скорость выполнения программы, скомпилированной в машинный код, превосходит скорость интерпретируемой программы, как правило, в десятки и сотни раз;
- в случае использования компилятора, при внесении изменений в исходный код программы, прежде чем эти изменения можно будет увидеть в работе программы, необходимо выполнить компиляцию исходного текста.

13. Что такое статическая и динамическая типизация?

·C++ является статически типизированным языком:
1.Каждая сущность в программе (переменная, функция и пр.) имеет свой тип,
2.Этот тип определяется на момент компиляции.

·Это нужно для того чтобы:
1.Вычислить размер памяти, который будет занимать каждая переменная в программе,
2.Определить, какая функция будет вызываться в каждом конкретном месте.

·Всё это определяется на момент компиляции и “зашивается” в скомпилированную программу. В машинном коде никаких типов нет — там идёт работа с последовательностями байт.
Динами́ческая типиза́ция — приём, широко используемый в языках программирования и языках спецификации, при котором переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов.

14. Что является единицей трансляции в C++?

В языках программирования C++ единица трансляции — подаваемый на вход компилятора исходный текст (файл с расширением .cpp) со всеми включёнными в него файлами.

15. Для чего и на какие файлы производится разбиение программы на C++?

Зачем разбивать программу на файлы?

·С небольшими файлами удобнее работать.

·Разбиение на файлы структурирует код.

·Позволяет нескольким программистам разрабатывать приложение одновременно.

·Ускорение повторной компиляции при небольших изменениях в отдельных частях программы.

Файлы с кодом на C++ бывают двух типов:

1.Файлы с исходным кодом (расширение .cpp);

•Может содержать как определения так и объявления;

•Объявления будут локальны для данного файла;

•Должен содержать директиву включения заголовочного файла;

•Не должен содержать объявлений, дублирующих объявления в соответствующем заголовочном файле.

2.Заголовочные файлы (расширение .hpp).

•Может содержать только объявление;

•Не должен содержать определения;

•Должен иметь механизм защиты от повторного включения:

```
#ifndef SYMBOL
#define SYMBOL
// Набор объявлений
#endif
```

16. Что такое union в C++, когда оно может быть применимо?

Union-объединение

Состоит из нескольких переменных, которые разделяют одну и ту же область памяти.

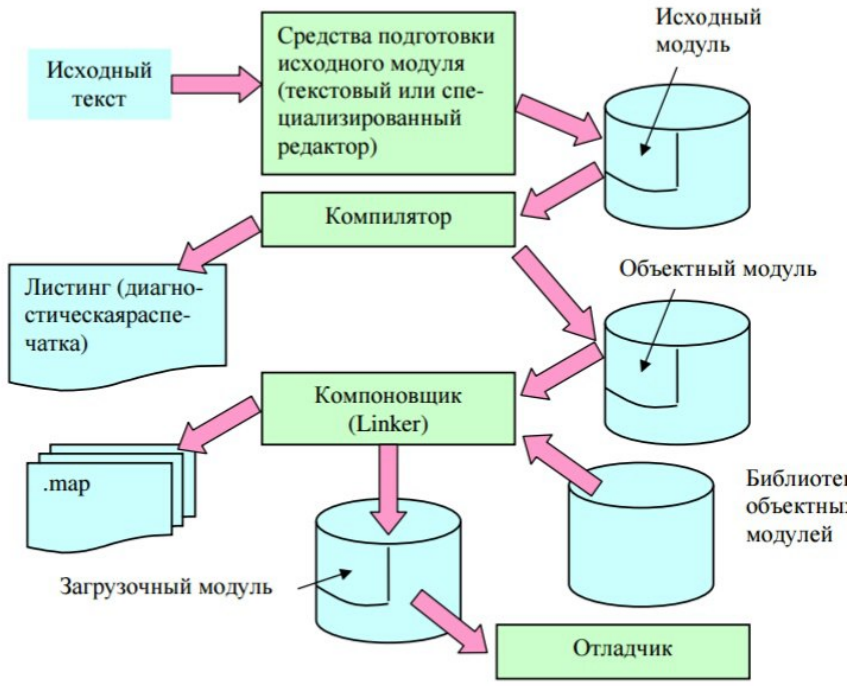
Обеспечивает низкоуровневую поддержку принципов полиморфизма.

структура способна хранить все свои элементы одновременно, а объединение в один момент времени может интерпретироваться только как один из своих элементов.

```
union Integer {
   int value;
   short half[2];
};
int main() {
   Integer integer;
   integer.value = 0xFFFF0000;
   cout << integer.half[0] << " " << integer.half[1] << endl;
};
;
```

17. Опишите процесс преобразования исходного кода в исполняемый файл.

Создание исполняемого файла издавна производилось в три этапа: (1) обработка исходного кода препроцессором, (2) компиляция в объектный код и (3) компоновка объектных модулей, включая модули из объектных библиотек, в исполняемый файл.



18. В чем отличие ссылки от указателя?

Основное назначение указателя – это организация динамических объектов, то есть размер, которых может меняться (увеличиваться или уменьшаться). Тогда как ссылки предназначены для организации прямого доступа к тому, или иному объекту. Главное отличие состоит во внутреннем механизме работы. Указатели ссылаются на участок в памяти, используя его адрес. А ссылки ссылаются на объект, по его имени (тоже своего рода адрес).

19. Что такое указатель на функцию и как он может быть использован?

Хотя функция - это не переменная, она по-прежнему имеет физическое положение в памяти, которое может быть присвоено указателю. Адрес, присвоенный указателю, является входной точкой в функцию. Указатель может использоваться вместо имени функции. Он также позволяет передавать функции как обычные аргументы в другие функции.

Пример:

Реализация callback function
Типо нажимаешь на кнопку в интерфейсе
И вызывается функция
На которую указывает указатель
Допустим в каком-нибудь калькуляторе
apply(x, y, &add)
Типо сложить 2 числа
apply(x, y, &mult)
apply(x, y, &div)

- используется в таблице виртуальных фугнкций,которая представляет собой массив указателей на функциб

20. Какие способы группировки данных в C++ вам известны?

Структуры, классы, union

21. Для чего предназначены структуры?

Структура – способ синтаксически (и физически) сгруппировать логически связанные данные.

22. Где может быть определена структура?

Везде

23. Допустимо ли использование указателей/ссылок/массивов структур?

Структура объявляет новый тип данных – можно использовать массивы этого типа.

По аналогии с массивами, можно объявлять указатели на структуры.

```
struct Node {
  struct Holder {  <- Вложенное объявление структуры
    int value;
  };
  Holder holder;
  Node *next;
};
int main() {
  Node second = {20, NULL};
  Node first = {10, &second};

  cout << first.next->holder.value << endl;
};
```

Для объектов структур имеется возможность передачи параметра функции *по ссылке*.

```
void shift(Point2D &point, double dx, double dy) {
  point.x += dx;
  point.y += dy;
}
int main() {
  Point2D point = {0, 0};
  shift(point, 1, 2);
  cout << point.x << " " << point.y << endl;
};
```

24. Какие существуют способы передачи параметров в функцию?

по ссылке, по значению, через указатель

Передача по ссылке, если мы не возвращаем значение, но нужно чтобы оно изменилось.

Указатель обращается к адресу, для использования надо разыменовывать.

25. Для чего предназначены классы, в чем их отличие от структур?

Класс определяет новый тип данных, который задает формат объекта(связывает данные с кодом-инкапсулирование)

поля структуры по умолчанию - public

поля класса по умолчанию - private

26. Что такое инвариант класса?

Инвариант класса - утверждение, которое(должно быть) истинно применительно к любому объекту данного класса в любой момент времени(за исключением переходных процессов в методах объекта)

27. В чем отличие функций от методов?

Методы организуются в классе/объекте, для выполнения блока кода с обращением к нему по имени класса/объекта и имени метода с возможностью передачи параметров как локальных, так и глобальных.

Функция имеет более простой синтаксис нежели метод и отличается несколькими признаками, такими как:

Функция не может находится в объекте/классе.

Функция не может иметь разграничение к ней доступа.

Функцию не нужно вызывать из под класса/объекта.

28. В каких случаях используются значения по умолчанию в функциях?

При обращении к функции, можно опускать некоторые её аргументы, но для этого необходимо при объявлении прототипа данной функции проинициализировать её параметры какими-то значениями, эти значения и будут использоваться в функции по умолчанию. Аргументы по умолчанию должны быть заданы в прототипе функции. Если в функции несколько параметров, то параметры, которые опускаются должны находиться правее остальных. Таким образом, если опускается самый первый параметр функции, то все остальные параметры тоже должны быть опущены. Если опускается какой-то другой параметр, то все параметры, расположенные перед ним могут не опускаться, но после него они должны быть опущены.

29. Что такое публичный интерфейс?

Публичный интерфейс – набор методов, доступный внешним пользователям класса.

30. Какие существуют модификаторы доступа, для чего они используются?

•Public – доступ открыт всем, кто видит определение данного класса;

•Protected – доступ открыт классам, производным от данного;

•Private – доступ открыт самому классу (т.е. функциям-членам данного класса) и друзьям (friend) данного класса, как функциям, так и классам.

Сделав метод закрытым, совсем не обязательно его сохранять при переходе к другой реализации. Такой метод труднее реализовать, а возможно, он окажется вообще ненужным, если изменится представление данных, что, в общем, несущественно. Важнее другое: до тех пор, пока метод является закрытым (private), разработчики класса могут быть уверены в том, что он никогда не будет использован в операциях, выполняемых за пределами класса, а следовательно, они могут просто удалить его. Если же метод является открытым (public), его нельзя просто так опустить,поскольку от него может зависеть другой код.

31. Что такое геттеры и сеттеры?

Обращения к private членам в классе get-возвращает значение, set-изменяет значение

32. Что такое inline-функции?

это встраиваемая функция, небольшая по объему, код которой поставляется вместо ее вызова(тем самым время работы программы сокращается, так как функцию вызывать более затратно)

Следует отметить, что inline - лишь рекомендация, а не команда компилятору заменять вызов функции ее телом. Он может подсчитать встраивание нецелесообразным и просто проигнорировать модификатор inline и трактовать функцию как обычную. Так что как говорится, на все воля компилятора...

Скорость:

- *быстрее* - внедрение кода функции в код программы поможет избежать использования лишних инструкций (связанных с вызовом функции и возвратом из нее)
- *медленнее* - слишком частое использование встраиваемых функций, к тому же больших, приведет к разрастанию кода (помимо этого, компилятор иногда вынужден использовать дополнительные временные переменные, чтобы сохранить семантику), что может привести в пробуксовке, т.е. в процессе работы программы операционной системе постоянно потребуется производить подкачку новых страниц

Размер исполняемого файла:

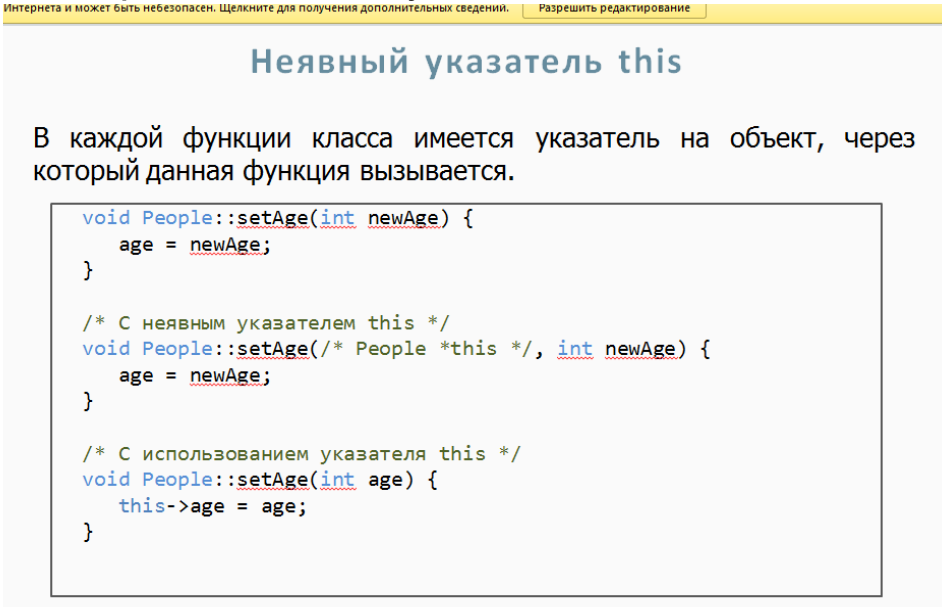
- *увеличить* - как уже упоминалось, слишком частое использование встраиваемых функций, к тому же больших, приведет к разрастанию кода и соответственно увеличению размера исполняемого файла
- *уменьшить* - благодаря оптимизирующему компилятору размер исполняемого файла может и уменьшиться при использовании очень маленьких встраиваемых функций, так как компилятору при этом не нужно будет создавать "лишние" инструкции для вызова функции и выхода из нее, помещению аргументов в стек и обратно. Так же во время внедрения большой встраиваемой функции в код программы оптимизирующий компилятор может удалить избыточный код, что опять же может уменьшить размер файла.

Пробуксовка (thrashing):

- *увеличить* - слишком частое использование встраиваемых функций, к тому же больших, приведет к разрастанию кода, что может привести в пробуксовке, т.е. в процессе работы программы операционной системе постоянно потребуется производить подкачку новых страниц
- *уменьшить* - может уменьшить количество страниц виртуальной памяти, которые должны быть в оперативной памяти одновременно. Например, когда функция f() вызывает функцию g(), коды функций, скорее всего, находятся на разных страницах виртуальной памяти,

при внедрении компилятором кода функции g() в код функции f(), часто код обеих функций находится на одной странице

33. Где применяется неявный указатель this?



34. Для чего используется ключевое слово const?

Позволяет определить типизированные константы, попытка изменить константные данные приводит к неопределенному поведению

Типизированные константы представляют собой фактически переменные с начальным значением, которые инициализируются (принимают указанное в описании значение) при запуске программы.*В отличие от значений локальных переменных, которые теряются при выходе из подпрограммы, типизированные константы сохраняют свои значения между вызовами подпрограммы.*Это связано с тем, что типизированные константы размещаются в памяти так же, как и глобальные переменные программы. Локальные же переменные располагаются во временной области памяти, называемой стеком.

35. Что такое константные ссылки/указатели, указатели/ссылки на константу?

«Слово const делает неизменяемым тип слева от него».

Константный указатель и указатель на константу

```
int a = 10;
const int *firstConstPointer = &a;  // Указатель на константу.
const int *secondConstPointer = &a; // Указатель на константу.
*firstConstPointer = 10;       <- Недопустимо изменения содержания.
secondConstPointer = NULL;          // Допустимо изменение указателя.
int * const pointerToConst = &a;    // Константный указатель.
*pointerToConst = 20;               // Допустимо изменения содержания.
pointerToConst = NULL;         <- Недопустимо изменение указателя.
// Константный указатель на константу.
int const * const constPointerToConst = &a;
*constPointerToConst = 30;     <- Недопустимо изменения содержания.
constPointerToConst = NULL;    <- Недопустимо изменение указателя.

int a = 10;
int *pointer = &a;
// Указатель на указатель на константу int.
int const **pointerToPointerToConst = &pointer;
// Указатель на константный указатель на int.
```

```
int * const *pointerToConstPointer = &pointer;
// Константный указатель на указатель на int.
int ** const constPointerToPointer = &pointer;
```

Константные ссылки

Сама по себе является неизменяемой.

```
int a = 10;
int & const reference = a;  <- Ошибка компиляции.
int const & constReference = a;  // Ссылка на константу.
```

Позволяет избежать копирование объектов при передаче в функцию.

```
Point midpoint(Segment const & segment);
```

36. В чем отличие синтаксической и логической константности методов?

Синтаксическая константность – константные методы не могут менять поля (обеспечивается компилятором).(короче здесь всё подтверждено кодом, то есть мы указали, что метод конст)

Логическая константность – запрещено изменение данных, определяющих состояние объекта в константных методах.(а тут просто подразумеваем, что метод ничего не изменяет, но не указываем это явно)

37. Для чего используется ключевое слово mutable?

позволяет определить поля, которые можно изменять внутри константного метода

38. Что такое конструктор?

Специальные функции, объявляемые в классе.

Имя функции совпадает с именем класса.

Не имеют возвращаемого значения.

Предназначены для инициализации создаваемых объектов класса.

39. В каких случаях используется перегрузка конструкторов?

В классе может присутствовать несколько конструкторов с разным количеством или типом параметров.

```
class Date {
  int year;
  int month;
  int day;
public:
  Date(int day, int month, int year);
  Date(int day, int month);
  Date(int day);
  Date();
```

+ если создается библиотека классов, которую будет использовать кто-то кроме ее автора, то, возможно, имеет смысл снабдить ее конструкторами для всех наиболее употребительных форм инициализации, позволяя программисту выбрать наиболее подходящую форму для его конкретного приложения.

40. Какую цель может преследовать создание приватного конструктора?

Запрет копирования объектов

41. Каким образом и в какой последовательности происходит инициализация полей объекта?

списки инициализации позволяют проинициализировать поля до входа в конструктор.

Инициализация полей происходит в порядке объявления полей в классе, а не в порядке ихследования в списке инициализации.

Объекты создаются «снизу-вверх» - от базовых к производным. Порядок вызовов конструкторов:

1.Конструкторы виртуальных базовых классов (в порядке объявления в списке наследования);

2.Конструкторы прямых базовых классов (в порядке объявления в списке наследования);

3.Конструкторы полей (в порядке объявления в классе);

Конструктор класса (вызванный)

42. Для чего используется ключевое слово explicit?

Рассмотрим следующий класс:

```
class MyClass {
 int i;
 public:
   MyClass(int j)
     {i = j;}
   // ...
};
```

Объекты этого класса могут быть объявлены двумя способами:

```
MyClass ob1(1);
MyClass ob2 = 10;
```

В данном случае инструкция:

```
MyClass ob2 = 10;
```

автоматически конвертируется в следующую форму:

```
MyClass ob2(10);
```

Однако, если объявить конструктор MyClass с ключевым словом explicit, то это автоматическое конвертирование не будет выполняться. Ниже объявление класса MyClass показано с использованием ключевого слова explicit при объявлении конструктора:

```
class MyClass {

 int i;
 public:
   explicit MyClass(int j)
     {i = j;}
   // ...
};
```

Теперь допустимыми являются только конструкции следующего вида:

```
MyClass ob (110);
```

43. В чем заключается предназначение конструктора по умолчанию?

Конструкторы по умолчанию не имеют параметров и подчиняются несколько иным правилам:
Конструкторы по умолчанию являются специальными функциями-членами. Если в классе не объявлены конструкторы, компилятор создает конструктор по умолчанию:
Если при вызове конструктора по умолчанию вы пытаетесь использовать скобки, выводится предупреждение:

```
class myclass{};
int main(){
myclass mc();     // warning C4930: prototyped function not called (was a variable definition intended?)
}
```

Это пример проблемы Most Vexing Parse (наиболее неоднозначного анализа). Поскольку выражение примера можно интерпретировать как объявление функции или как вызов конструктора по умолчанию и в связи с тем, что средства синтаксического анализа C++ отдают предпочтение объявлениям перед другими действиями, данное выражение обрабатывается как объявление функции.

```
class Box {
   Box(int width, int length, int height)
      : m_width(width), m_length(length), m_height(height){}
};
private:
```

```
    int m_width;
    int m_length;
    int m_height;

};

int main(){

    Box box1(1, 2, 3);
    Box box2{ 2, 3, 4 };
    Box box4;     // compiler error C2512: no appropriate default constructor available
}
```

Если у класса нет конструктора по умолчанию, массив объектов этого класса не может быть создан только с помощью синтаксиса двух квадратных скобок. Например, в представленном выше блоке кода массив Boxes не может быть объявлен следующим образом:
```
Box boxes[3];    // compiler error C2512: no appropriate default constructor available
```

Однако для инициализации массива Boxes можно использовать набор списков инициализаторов:

```
Box boxes[3]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

44. Что такое деструктор, для чего он используется?
Специальные функции, объявляемые в классе.
Имя функции совпадает с именем класса, плюс знак ~(тильда) в начале.
Не имеют возвращаемого значения и параметров.
Вызывается автоматически при удалении экземпляра структуры.
Предназначены для освобождения используемых ресурсов.

45. Каков порядок вызова деструкторов при разрушении объекта?

Деструкторы переменных на стеке вызываются в обратном порядке (по отношению к порядку вызова конструкторов).
Объекты разрушаются «сверху-вниз» - от производных к базовым. Порядок вызовов деструкторов:

1. *Деструктор класса (вызванный);*

2. *Деструкторы полей (в порядке обратном объявлению в классе);*

3. *Деструкторы прямых базовых классов (в порядке обратном объявлению в списке наследования);*

4. *Деструкторы виртуальных базовых классов (в порядке обратном объявлению в списке наследования).*

При допустимости использования наследования деструктор следует объявлять виртуальным

46. В какой момент вызывается деструктор объекта?
Вызывается автоматически при удалении экземпляра структуры.
Предназначены для освобождения используемых ресурсов.**(из презентации)**

Деструкторы вызываются, когда происходит одно из следующих событий:**(из интернета)**
- Объект, предоставленный с использованием оператора new, можно явно освободить с использованием оператора delete. Если объекты освобождаются с помощью оператора delete, память освобождается для "наиболее производного объекта" или объекта, который является полным, а не вложенным объектом, представляющим базовый класс. Освобождение "самого производного объекта" гарантированно работает только с виртуальными деструкторами. Освобождение может завершиться ошибкой в случае множественного наследования, если сведения о типе не соответствуют базовому типу фактического объекта.

- Локальный (автоматический) объект с областью видимости "блок" выходит за пределы области видимости.
- Время существования временного объекта заканчивается.
- Программа заканчивается, глобальные или статические объекты продолжают существовать.
- Деструктор явно вызываться с использованием полного имени функции деструктора.

47. Каково время жизни объекта?

Время жизни – временной интервал между вызовами конструктора и деструктора.

48. Зачем нужен виртуальный деструктор?

Деструктор интерфейсов или абстрактных классов обычно делают виртуальным. Такой прием позволяет корректно удалять без утечек памяти, имея лишь указатель на базовый класс.

Пусть (на C++) есть тип Father и порождённый от него тип Son:

```
class Father
    {
    public:
      Father() {}
      ~Father() {}
    };

    class Son : public Father
    {
    public:
      int* buffer;
      Son() : Father() { buffer = new int[1024]; }
      ~Son() { delete[] buffer; }
    };
```

Нижеприведённый код является некорректным и приводит к утечке памяти.

```
Father* object = new Son(); // вызывается Son()
    delete object; // вызывается ~Father()!!
```

Однако, если сделать деструктор Father виртуальным:

```
class Father
    {
    public:
      Father() {}
      virtual ~Father() {}
    };
    class Son : public Father
    {
    private:
      int* buffer;
    public:
      Son() : Father() { buffer = new int[1024]; }
      ~Son() { delete[] buffer; }
    };
```

вызов delete object; приведет к последовательному вызову деструкторов ~Son и ~Father.

В языке программирования C++ деструктор полиморфного базового класса должен объявляться виртуальным. Только так обеспечивается корректное разрушение объекта производного класса через указатель на соответствующий базовый класс.
Когда же следует объявлять деструктор виртуальным? Существует правило - если базовый класс предназначен для полиморфного использования, то его деструктор должен объявляться виртуальным. Для реализации механизма виртуальных функций каждый объект класса хранит указатель на таблицу виртуальных функций vptr, что увеличивает его общий размер. Обычно, при объявлении виртуального деструктора такой класс уже имеет виртуальные функции, и увеличения размера соответствующего объекта не происходит.
Используя виртуальные деструкторы, можно уничтожать объекты, не зная их тип — правильный деструктор для объекта вызывается с помощью механизма виртуальных функций. Обратите внимание, что для абстрактных классов деструкторы также могут объявляться как чисто виртуальные функции.

49. Как осуществляется работа с динамической памятью в C/C++?

- *Cи*

Выделение памяти:

void* malloc(size_t sizemem) – выделяет блок памяти, размером sizemem байт и возвращает указатель на начало блока. Содержание выделенного блока памяти *не инициализируется* (остается с неопределенными значениями).

void* calloc(size_t nmemb, size_t size) – выделяет память для массива размером nmemb, каждый элемент которого равен size байт и возвращает указатель на выделенную память. Память при этом *очищается* (зануляется).

void* realloc(void *ptr, size_t size) – меняет размер блока памяти, на который указывает ptr, на размер, равный size байт. Содержание будет неизменным в пределах наименьшего из старых и новых размеров, а новая распределенная память будет *не инициализирована*.

Освобождение памяти:

void free(void *ptr) – освобождает место в памяти, на который указывает ptr, полученный динамическим выделением памяти. Иначе (если функция free уже вызывалась для этого участка памяти, дальнейший ход событий непредсказуем – undefined behavior). Если ptr == NULL, то не выполняется никаких действий.

Вызов функции realloc с параметром size равным нулю эквивалентен free(ptr).

- *C++*

Для создания объекта в динамической памяти используется оператор new, он отвечает за вызов конструктора.

При вызове оператора delete вызывается деструктор объекта.

Операторы new[] и delete[] работают аналогично только для массивов

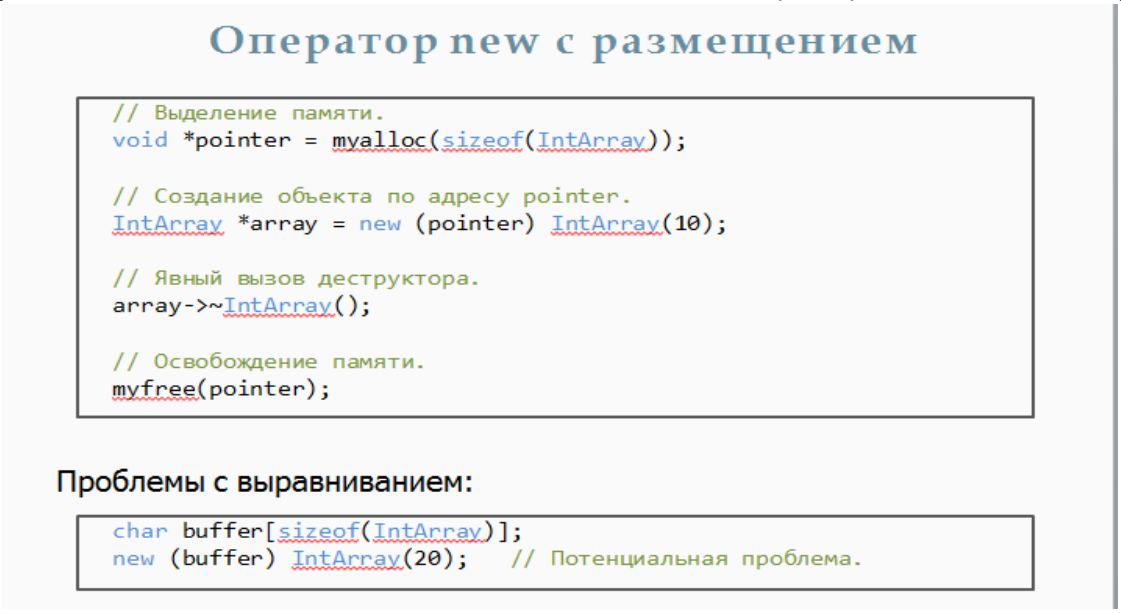
50. В чем различие delete и delete[]?

Работают аналогично с new и delete, но new[] и delete[] предназначены для выделения и очистки памяти для массива .

51. Как работает оператор new с размещением?

Существует особая форма оператора new, называемая Placement new. Данный оператор не выделяет память, а получает своим аргументом адрес на уже выделенную каким-либо образом память (например, на стеке или через malloc()). Происходит размещение (инициализация) объекта путём вызова конструктора, и объект создается в памяти по указанному адресу. Часто такой метод применяют, когда у класса нет конструктора по умолчанию и при этом нужно создать массив объектов. Звучит просто, однако нужно помнить, что платим мы высокую цену: заранее

указывая максимальный размер буфера.



52. Что подразумевается под идиомой RAII?

Resource Acquisition Is Initialization (получение ресурса есть инициализация).

Программная идиома объектно-ориентированного программирования.

Основная идея – с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение – с уничтожением объекта.

Типичный способ реализации – организация получения доступа к ресурсу в конструкторе, а освобождения – в деструкторе.

Применяется для:
•Выделения памяти
•Открытия файлов/устройств/каналов
•Мьютексов/критических секций/других механизмов блокировки

53. Перечислите основные подходы к обработке ошибок.
•Остановка программы:
Все ОЧЕНЬ плохо.
•Возврат кода ошибки из вызванного метода:
int getValueFromDB();
Конструктор.
•Сообщение об ошибке через внешнюю переменную (errno):
Необходимо постоянно проверять значение;
Плохо работает в многопоточной среде.
•Использование функций-обработчиков сразу:
Обработчик ошибки должен решить её в месте вызова;
Должен быть универсален.

54. Для чего предназначен механизм обработки исключительных ситуаций?

Является альтернативой традиционным техникам, когда они недостаточны, плохо читаемы, неэффективны.

Позволяет программистам разделить код, обрабатывающий исключительные ситуации, от кода, реализующего естественное поведение программы.

Предоставляет единый стиль обработки ошибок.

Если функция не может обработать ошибку – она «бросает» исключение, рассчитывая на то, что вызывающая сторона сможет решить проблему.

Функция, принимающая на себя обязательства по решению проблемы «ловит» возникающие исключения.

55. Что такое исключение?

Это аномальное поведение функции во время выполнения, которое программа может обнаружить Исключения используются тогда, когда код решения проблемы находится не рядом с местом её возникновения.

Предназначены для доставки информации из точки, где обнаружена ошибка в точку, где она может быть обработана.

Хорошо подходят для библиотек, т.к. ответственность за обработку некоторых видов ошибок библиотека не должна брать на себя. В таком случае, она должна предоставить вызывающему коду возможность разрешить возникшую проблему.

56. Какие типы данных допустимы для использования в качестве объектов exception?

Объекты исключений могут быть любого типа, допускающего копирование.
Строго рекомендуется использовать специально разработанные типы для объектов, которые «бросаются» в исключительных ситуациях. Это минимизирует шансы на пересечение с другими типами ошибок от сторонних компонентов программы

57. Как происходит возбуждение исключения?

Исключение возбуждается путем создания объекта-исключения и «пробрасывания» его вверх по стеку вызовов с использованием оператора throw.

Объект-исключение, который будет описывать возникшую исключительную ситуацию должен содержать конструктор копирования или перемещения.

58. Кто отвечает за обработку возникших исключительных ситуаций?

Инструкции, которые могут возбуждать исключения должны быть заключены в try-блок.
После блока try следует список обработчиков – блоков catch, отвечающих за обработку исключительных ситуаций, возникающих в процессе выполнения блока try.

Исключения обрабатываются в блоках catch.
Когда какая-то инструкция внутри блока try возбуждает исключение – просматривается список последующих блоков catch в поисках того, который сможет обработать возникшее исключение.

Для «отлова» любых исключений, возникающих в блоке try, вне зависимости от их типа, следует использовать конструкцию следующего вида:

```
try {
  function();
} catch (MyException) {
  // Обработка исключительной ситуации типа MyException
} catch (...) {
  // Обработка всех остальных типов исключительных ситуаций
}
```

59. Что такое раскрутка стека?

Если среди блоков catch не обнаружилось способного обработать возбужденное исключение – производится поиск подходящих обработчиков на вышестоящих уровнях, т.е. в блоках кода, непосредственно вызвавших текущий.

Таким образом происходит «раскрутка» стека до точки входа в приложение. Если на всем протяжении не было встречено подходящего обработчика – исключение считается не обработанным.

В таком случае вызывается функция terminate() – текущий обработчик завершения. По умолчанию он реализован как abort() – не вызывает деструкторы, завершает программу с ошибкой.

Изменить такое поведение можно задав функцию-обработчик завершения через set_terminate(terminate_handler).

60. Где и для чего используется спецификатор throw?

Для декларации списка исключений, которые может породить функция в процессе своего выполнения использовался спецификатор throw с параметрами, указываемый после списка аргументов функции

```
int firstFunction() throw() {
  // Тело функции
}
int secondFunction() throw(MyFirstException, MySecondException) {
  // Тело функции
}
int thirdFunction() {
  // Тело функции
}
```

61. Где и для чего используется спецификатор noexcept?

Для снижения накладных расходов, возникающих при использовании механизма исключений используется спецификатор noexcept (добавлен в стандарте C++11).

Он гарантирует, что функция, отмеченная данным спецификатором не сгенерирует никакого исключения в процессе выполнения.

Если же это произойдет – будет вызвана функция terminate() без раскрутки стека.

```
int firstFunction() noexcept {
  // Тело метода
}
int secondFunction() {
  // Тело метода
}
```

62. К чему приводит вызов throw без аргументов?

В конце оператора catch может стоять оператор throw без параметров. В этом случае работа catch-блока считается незавершенной а исключение – не обработанным до конца, и происходит поиск соответствующего обработчика на более высоких уровнях.

Пример:

```
try
{
   //....
   try
   {
      // Call something
   }
   catch(const std::exception& )
   {
      // Make/Check something..
      throw; // Пересылаем исключение следующему обработчику
   }
   //...
}
catch(const std::exception& e)
{
   std::cout << e.what() << std::endl;
}
```

63. Что такое exception-safe операция?

Для того чтобы операцию можно было считать exception-safe она должна оставлять программу в консистентном состоянии, вне зависимости от того, была она завершена успешно или возникла исключительная ситуация.

Применительно к объектам консистентное состояние значит что конструктор объекта успешно выполнился, деструктор еще не был вызван, а сам объект соответствует инварианту класса.

64. Что такое делегирующие конструкторы?

Если не читать вот это всё….то грубо говоря, делегирующие конструкторы - это те, которые вызываются внутри другого конструктора...

Допустим, у нас есть класс с двумя конструкторами: один от параметра типа int, а другой от double. Конструктор для int делает то же самое, что и конструктор для double, только сначала он переводит параметр от типа int к типу double. Т.е. конструктор для int делегирует создание объекта конструктору для double.

После того, как конструктор для double закончит выполнение, конструктор для int может продолжить выполняться и “доконструировать” объект. Сама по себе это очень полезная фича, без которой в коде выше нам наверняка пришлось бы ввести дополнительную функцию init(double param) для инкапсуляции общего кода по созданию объект от типа double.

Но в дополнение у этой фичи есть один очень интересный побочный эффект. Дело в том, что как только один из конструкторов объекта закончит выполнение, объект считается созданным. И значит, если другой конструктор, из которого произошел делегирующий вызов первого конструктора, завершится с выбросом исключения, для этого объекта все равно будет вызван деструктор. Заметьте критический момент: для объекта теперь может выполниться больше одного конструктора. Но объект считается созданным после выполнения самого первого конструктора.

Продемонстрируем это поведение на следующем примере:

```
class MyClass
{
public:
    MyClass(double)
    {
        cout << "ctor(double)\n";
    }

    MyClass(int val)
        : MyClass(double(val))
    {
        cout << "ctor(int)\n";
        throw "oops!";
    }

    ~MyClass()
    {
        cout << "dtor\n";
    }
};

int main()
try
{
    MyClass obj(10);
    cout << "obj created";
}
catch (...)
{
    cout << "exception\n";
}
```

Конструктор MyClass(int) вызывает другой конструктор MyClass(double), после чего сам выбрасывает исключение. Это исключение ловится в catch(...), и при раскрутке стека вызывается деструктор ~MyClass. На консоль при выполнении данного кода выведется следующее:

```
ctor(double)
ctor(int)
dtor
exception
```

Делегирующие конструкторы и RAII

Нетрудно заметить, что такое интересное поведение конструкторов при делегировании можно очень эффективно использовать в нашем примере реализации RAII для FILE. Теперь нам не нужно вводить никакой дополнительный класс FileHandle для освобождения ресурса FILE, а тем более не нужен и try/catch. Нужно ввести всего лишь один дополнительный конструктор, которому будет произведена делегация из основного конструктора. То есть:

```
class File
{
    File(FILE * file)
        : file_(file)
    {}

public:
    File(char const * filename, char const * mode)
        : File(fopen(filename, mode))
    {
        put_time_stamp();
    }

    ~File()
    {
        fclose(file_);
    }

    void put_time_stamp() { ... }

private:
    FILE * file_;
};
```

65. Какого уровня гарантии предоставляются библиотечными функциями?

•Basic guarantee (для всех операций) – соблюдается базовый инвариант для всех объектов, нет утечки ресурсов (памяти, дескрипторов и т.д.).

•Strong guarantee (для ключевых операций) – в дополнение к базовым гарантиям, операции либо выполняются полностью, либо не имеют эффекта совсем.

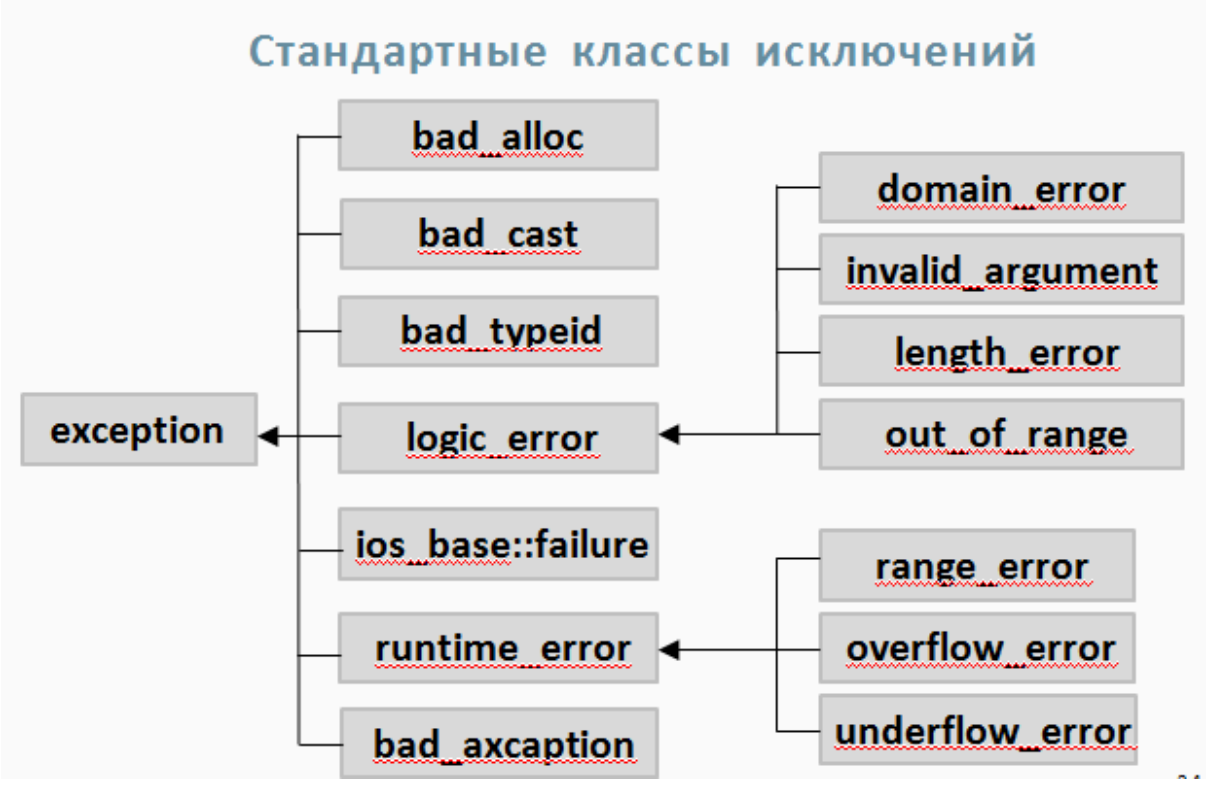
•Nothrow guarantee (для некоторых операций) – гарантируют отсутствие возможности возбуждения исключения при выполнении операции.

66. Что вы можете сказать о генерации исключений в конструторе/деструкторе?

Исключения в конструкторе:
•Допустимы;
•Объект не считается полностью сконструированным;
•Не вызывается деструктор.

Исключения в деструкторе:
•Недопустимы;
•После броска объект находится в неопределенном состоянии.

67. Какие стандартные классы исключений вам известны, для чего они предназначены?



Классы исключений языковой поддержки

bad_alloc – генерируется при неудачном выполнении оператора new.
bad_cast – генерируется оператором dynamic_cast, если преобразование типа по ссылке во время выполнения завершается неудачей.
bad_typeid – генерируется оператором typeid, предназначенным для идентификации типов во время выполнения, в случае если в качестве аргумента передан nullptr.
bad_exception – предназначено для обработки непредвиденных исключений. В его обработке задействована функция unexpected

Классы исключений стандартной библиотеки

invalid_argument – передано недопустимое значение аргумента.
length_error – осуществлена попытка выполнения операции, нарушающей ограничения на максимальный размер.
out_of_range – аргумент не входит в интервал допустимых значений.
domain_error – ошибка выхода за пределы области допустимых значений.
ios_base::failure – генерируется при изменении состояния потока вследствие ошибки или достижения конца файла.

Классы исключений для внешних ошибок

Исключения, производные от runtime_error, сообщают о событиях, не контролируемых программой. В стандартную библиотеку C++ включены следующие классы ошибок времени выполнения:
range_error – выход за пределы допустимого интервала во внутренних вычислениях.
overflow_error – математическое переполнение.
underflow_error – математическая потеря точности.
<exception> – базовые классы exception и bad_exception.
<new> – класс bad_alloc.

<typeinfo> – классы bad_cast и bad_typeid.
<ios> – класс ios_base::failure.
<stdexcept> - все остальные классы стандартных исключений

68. Что такое ассоциация?

•Структурное отношение;
•Означает отношение между классами объектов;
•Позволяет одному экземпляру объекта вызывать другого, чтобы выполнить действие от его имени;
•В общем случае показывает некую связь объектов разных классов между собой.

69. Что такое композиция и агрегация, чем они отличаются?

агрегация
•Структурное отношение;
•Определяет отношение HAS A, т.е. отношение владения;
•Описывает целое и составные части, в которые в него входят;
Целое НЕ является владельцем части и НЕ управляет временем её жизни.
композиция
•Структурное отношение;
•Определяет отношение HAS A, т.е. отношение владения;
•Описывает целое и составные части, в которые в него входят;
•Конкретный экземпляр части может принадлежать только одному владельцу;
•Целое управляет временем жизни входящих в него частей.

70. Время жизни агрегируемого объекта меньше времени жизни агрегата?

Нет, так как агрегируемый объект не зависит от жизни агрегата

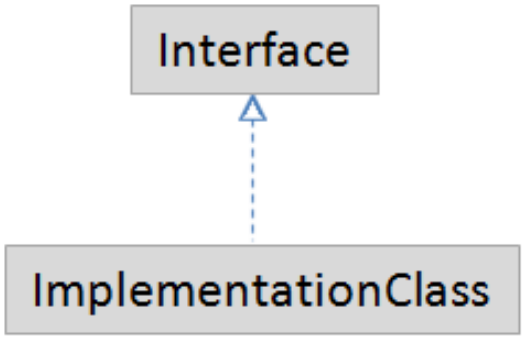
71. Какие классы называются дружественными, для каких целей используется это отношение?

дружественность
•Структурное отношение;
•Однонаправленная связь классов;
•Не может быть предписана извне класса;
•Предоставляет внешнему коду доступ к внутреннему состоянию и методам класса;
•Следует использовать с осторожностью.

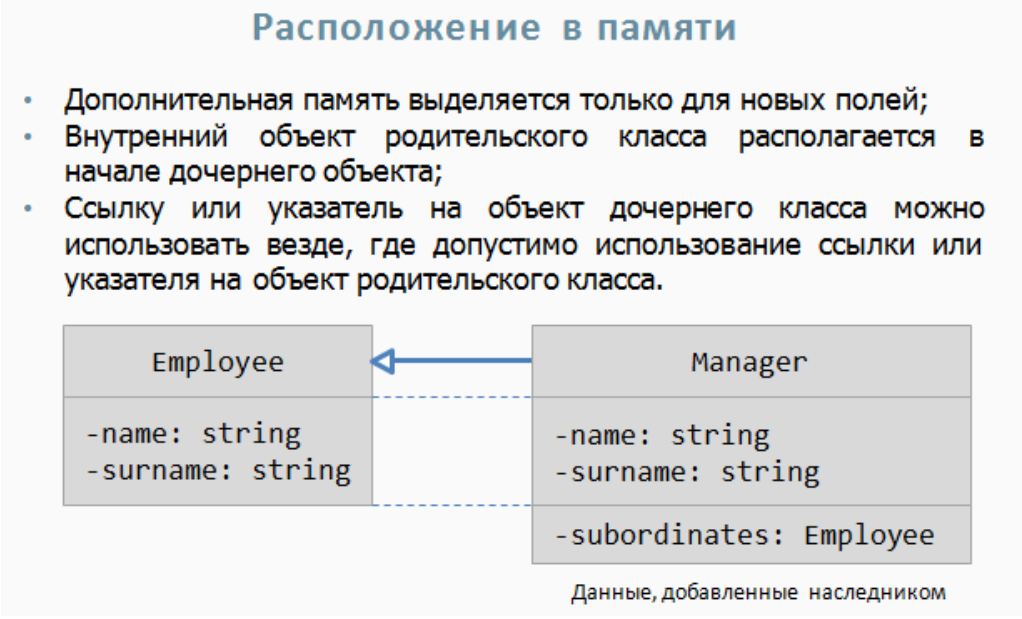
```
class Tv {
  int currentChanel;
  friend class RemoteControlTv;
};
class RemoteControlTv {
public:
  void changeChannel(Tv& tv, int channel) {
    tv.currentChannel = channel;
  }
};
```

72. В каком случае можно говорить об отношение «реализация»?

•Структурное отношение;
•Необходимо определение интерфейса (класса, содержащего только чистые виртуальные методы);
•Наследование от подобного интерфейса.



73. Что из себя представляет объект дочернего класса в памяти?



74. Какие существуют типы наследования, чем они различаются?

Базовый класс может быть объявлен с одним из следующих модификаторов доступа:

•public – публичные члены базового класса доступны как публичные члены производного класса, защищенные члены базового класса как защищенные члены производного;

•protected – публичные и защищенные члены базового класса доступны как защищенные члены производного класса;

•private – публичные и защищенные члены базового класса доступны как приватные члены производного класса.

75. Наследуются ли конструкторы и деструкторы?

Конструкторы и деструкторы не наследуются.

76. Наследуются ли приватные поля базового класса?

Приватные члены базового класса недоступны в дочернем ни при каком типе наследования.

77. Что такое виртуальная функция?

Это функция-член класса, которую предполагается переопределить в производных классах.

При ссылке на объект производного класса с помощью указателя или ссылки на базовый класс можно вызвать виртуальную функцию для этого объекта или выполнить версию функции производного класса.

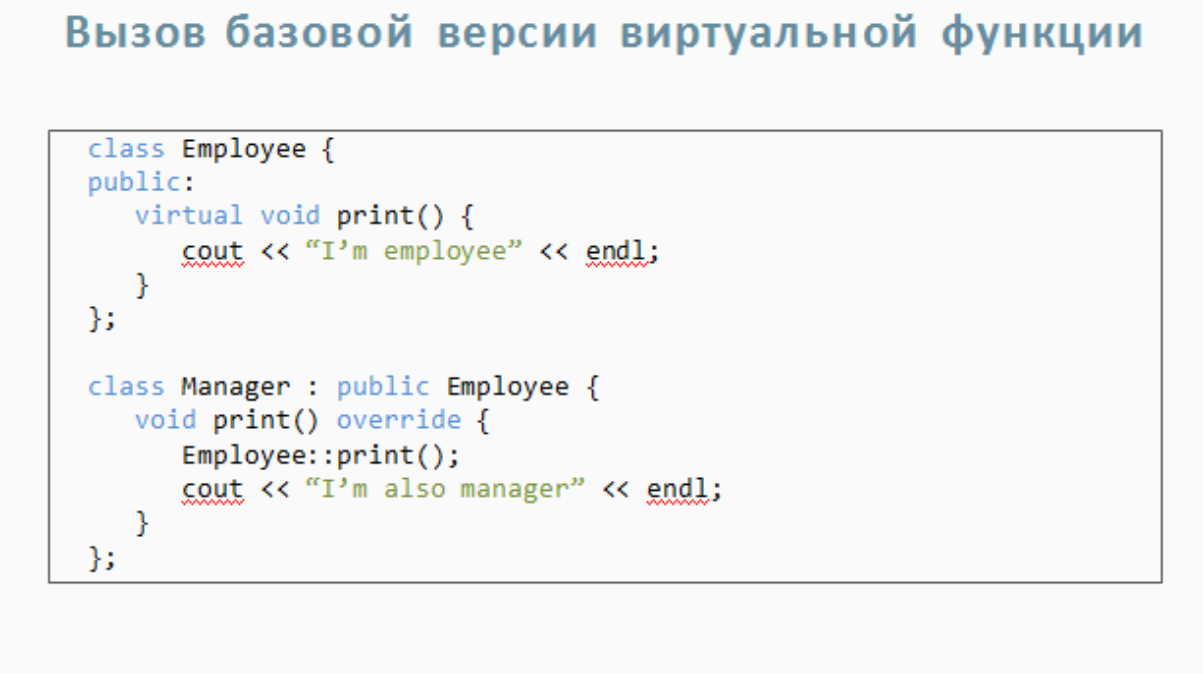
Для объявления функции виртуальной – следует использовать ключевое слово virtual, расположив его перед типом возвращаемого значения.

Должна быть определена в месте первого объявления (кроме случая, когда это чисто виртуальная функция).

Может быть переопределена в дочерних классах. При этом следует использовать спецификатор override.Контекстно-зависимое ключевое слово override указывает, что элемент типа переопределяет член базового класса или базового интерфейса.

78. Как осуществить вызов базовой реализации функции при её переопределении в дочернем классе?

Внутри класса использовать указание области видимости, т.е. base::f(), где f() - необходимая виртуальная функция.



79. Как связаны виртуальные функции и полиморфизм?

Любой класс, содержащий, по крайней мере, одну виртуальную функцию является полиморфным.

При объявлении функции как виртуальной она в разных классах может иметь разную реализацию, при сохранении одного и того же интерфейса, что является проявлением полиморфизма.

При использовании механизма виртуальных определение конкретного типа объекта, хранящегося в указателе (или ссылке) на базовый класс, не требуется.

При вызове будет использован полиморфизм и реализация метода будет выбрана в зависимости от типа хранимого объекта.

Каждый объект такого класса содержит таблицу виртуальных функций (vtable).

При использовании ссылки или указателя на такой класс разрешение методов происходит динамически в момент выполнения.

80. Что такое переопределение функций?

Переопределение (англ. overriding) означает, что вы создали иерархию классов, у которой в базовом классе есть виртуальная функция и вы можете переопределить (override) её в производном классе.

81. Работает ли переопределение для приватных функций?

Только для виртуальных функций, для остальных нет

82. Что такое таблица виртуальных функций?

Она же координирующая таблица или vtable – механизм, используемый для осуществления позднего связывания.

Указатель на vtable хранится в каждом объекте (при условии, что класс содержит хотя бы одну виртуальную функцию).

Содержит адреса динамически связанных методов объекта.

Определение необходимого для вызова метода и его вызов осуществляется в процессе выполнения программы путем выбора адреса требуемого метода из таблицы.

83. Как себя ведут виртуальные функции в конструкторе и деструкторе?

Виртуальные функции в конструкторе и деструкторе

При вызове виртуальных функций в конструкторе или деструкторе будет вызвана версия, специфичная для класса, в конструкторе (деструкторе) которого она вызвана.

```
struct Person {
   Person(string const &name) : name(name);
   virtual string name() const { return name; }
   virtual ~Person();
};

struct Teacher : Person {
   Teacher(string const &name) : Person(name) { cout << name(); }
};

struct Professor : Teacher {
   Professor(string const &name) : Teacher(name) {}
   string name() const { return "Prof. " + name; }
};
```

Виртуальная функция не является виртуальной, если вызывается из конструктора или деструктора.

Правило надо заучивать, что неудобно. Проще понять принцип. А принцип тут в краеугольном камне реализации наследования в C++: при создании объекта конструкторы в иерархии вызываются от базового класса к самому последнему унаследованному. Для деструкторов все наоборот.

Что получается: конструктор класса всегда работает в предположении, что его дочерние классы еще не созданы, поэтому он не имеет права вызывать функции, определенные в них. И для виртуальной функций ему ничего не остается, как только вызвать то, что определено в нем самом. Получается, что механизм виртуальных функций тут как-бы не работает. А он тут действительно не работает, так как таблица виртуальных функций дочернего класса еще не перекрыла текущую таблицу.

В деструкторе все наоборот. Деструктор знает, что во время его вызова все дочерние классы уже разрушены и вызывать у них ничего уже нельзя, поэтому он замещает адрес таблицы виртуальных функций на адрес своей собственной таблицы и благополучно вызывает версию виртуальной функции, определенной в нем самом.

Итак, виртуальная функция не является виртуальной, если вызывается из конструктора или деструктора.

84. В каких случаях допустимо приведение указателей/ссылок на дочерний класс к базовому?

Приведение производного класса к базовому:

Использование ссылки или указателя на объект производного класса допустимо везде, где предполагается использование ссылки или указателя на объект базового класса.

```
Manager manager("Name", "Surname", "Sales");
Employee &ref = manager;              // Manager& -> Employee&
Employee *ref = &manager;             // Manager* -> Employee*
```

Допустимо присвоение переменной базового класса объекта производного. При этом используется конструктор копирования родительского класса (срезка)

```
Manager manager("Name", "Surname", "Sales");
Employee employee = manager;    // Employee("Name", "Surname");
```

Приведение классов с модификаторами доступа

При использовании публичного наследования описанный ранее механизм использования указателя (ссылки) на производный класс допустим везде.

```
class Class {};
class PublicInheritor : public Class{};
class ProtectedInheritor : protected Class{};
class PrivateInheritor : private Class{};
```

При наследовании с модификатором protected, о том что Class является базовым для ProtectedInheritor «знают» только он сам и его наследники.

При использовании же модификатора private приведение указателя (ссылки) к базовому классу допустимо только внутри класса PrivateInheritor.

85. Что такое чистая виртуальная функция?

Функция, которая объявляется в базовом классе, но не имеет в нем определения.

Всякий производный класс обязан иметь свою собственную версию определения.

Для объявления чистой виртуальной функции следует использовать ключевое слово virtual, расположив его перед типом возвращаемого значения и указать = 0; после списка аргументов, исключив при этом тело функции (оставить её без реализации).

86. Какой класс называется абстрактным?

Любой класс, содержащий по крайней мере одну чистую виртуальную функцию является абстрактным.

Предназначен для хранения общей реализации и поведения некоторого множества дочерних классов.

Объекты абстрактного класса создать нельзя.

Может содержать чисто виртуальный деструктор.

87. Как в C++ реализуются интерфейсы?

В C++ отсутствует специальная синтаксическая конструкция для определения интерфейса.

Интерфейсом является класс, содержащий только public секцию и толь ко чистые виртуальные методы.

Интерфейс не должен содержать поля.

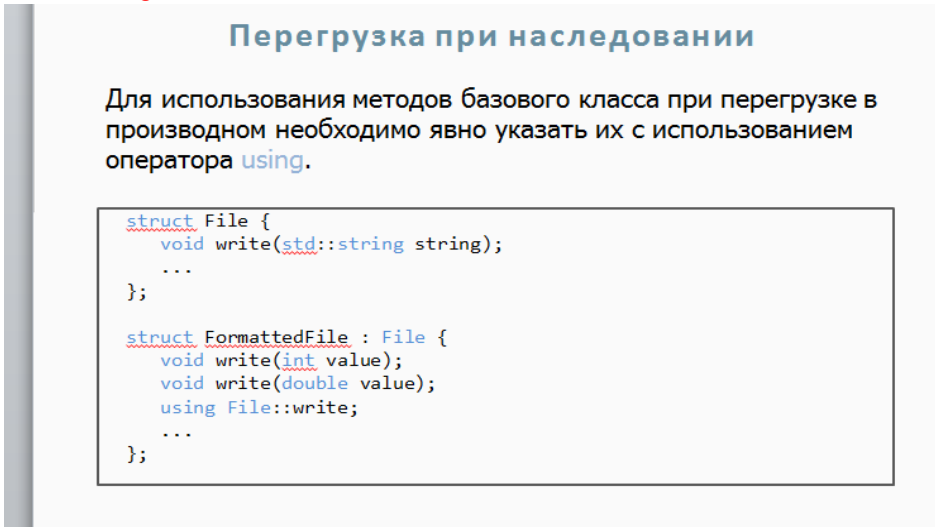
Каждый интерфейс является абстрактным классом, но не каждый абстрактный класс – интерфейс.

При использовании интерфейс реализуют, абстрактный класс – наследуют (в C++ синтаксически оба подхода выражаются через механизм наследования).

88. Что такое перегрузка функций?

Определение нескольких функций с одинаковым именем, но различными параметрами.

89. Как ведет себя перегрузка при наследовании?
пишем using *ClassParent::functionParent*



90. Опишите процесс выбора функции среди перегруженных.
разрешение перегрузки
1.При наличии точного совпадения сигнатуры – используется найденная функция.
2.Если не найдена функции, которая могла бы подойти с учетом типом преобразований – выдается ошибка.
3.Если есть функции, подходящие с учетом преобразований:
3.1 Расширение типов:

1. char, short -> int
2. unsigned char, unsigned short -> int / unsigned int
3. float -> double
4. int-> unsigned int->long-> unsigned long;
5. bool-> int.

3.2 Стандартные преобразования (числа, указатели).

1. преобразования целых типов: приведение от целого типа или перечисления к любому другому целому типу (исключая трансформации, которые выше были отнесены к категории расширения типов);
2. преобразования типов с плавающей точкой: приведение от любого типа с плавающей точкой к любому другому типу с плавающей точкой (исключая трансформации, которые выше были отнесены к категории расширения типов);
3. преобразования между целым типом и типом с плавающей точкой: приведение от любого типа с плавающей точкой к любому целому типу или наоборот;
4. преобразования указателей: приведение целого значения 0 к типу указателя или трансформация указателя любого типа в тип void*;
5. преобразования в тип bool: приведение от любого целого типа, типа с плавающей точкой, перечислимого типа или указательного типа к типу bool.

3.3 Пользовательские преобразования.

Подходящая функция должна быть единственной и строго лучше остальных по каждому из параметров.

Для выбора необходимого перегруженного метода используется ранее связывание. Для определения требуемой реализации переопределенного метода – позднее связывание

91. Чем отличаются механизмы раннего и позднего связывания?

Перегрузка функций:

Выбор функции происходит в момент компиляции на основе типов аргументов функции – статический полиморфизм (раннее связывание).

Виртуальные методы:

Выбор метода происходит в момент выполнения на основе типа объекта, у которого вызывается виртуальный метод – динамический полиморфизм (позднее связывание).

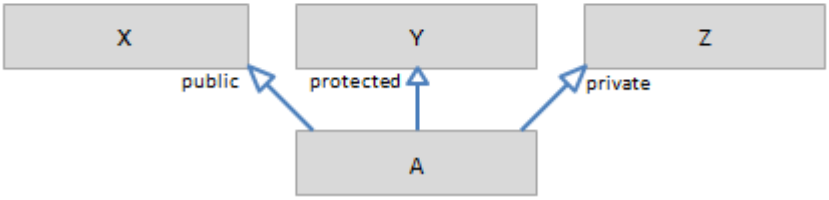
92. Что такое множественное наследование?

Если порожденный класс наследует элементы одного базового класса, то такое наследование называется одиночным.

Множественное наследование позволяет порожденному классу наследовать элементы более чем от одного базового класса.

Синтаксис описания заголовка класса расширяется так, чтобы разрешить создание списка базовых классов и обозначение их уровней доступа.

```
class X {};
class Y {};
class Z {};
class A : public X, protected Y, private Z {};
```



93. Что такое ромбовидное наследование?

94. Какой существует механизм разрешения проблемы ромбовидного наследования в C++?

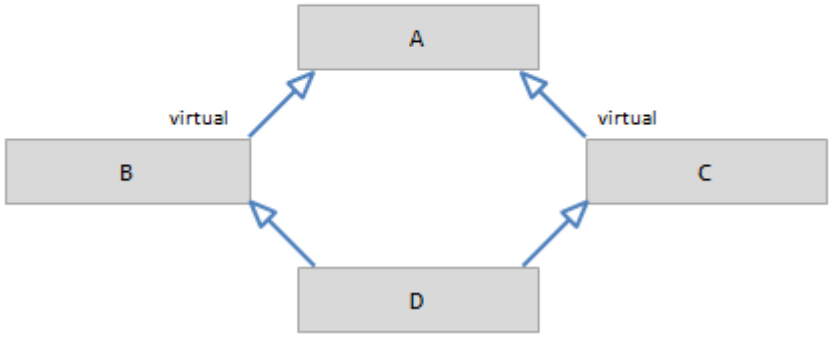
C++ по умолчанию не создает ромбовидного наследования: компилятор обрабатывает каждый путь наследования отдельно.

В результате объект D будет содержать два разных подобъекта A, и при использовании членов A потребуется указать путь наследования (B::A или C::A).

Для создания полноценного ромбовидного наследования следует использовать механизм виртуального наследования. Один из вариантов наследования, который нужен для решения некоторых проблем, порождаемых наличием возможности множественного наследования. Позволяет разрешить неоднозначность выбора методов и полей суперклассов при множественном наследовании.

Базовый класс, наследуемый множественно, определяется виртуальным с помощью ключевого слово virtual.

Важный аспект виртуального программирования – оно должно появиться в иерархии раньше, чем в нем возникнет реальная необходимость.



95. Для чего используется оператор разрешения контекста?

Множественное наследование. Доступ к членам:

Для доступа к членам порожденного класса, унаследованного от нескольких базовых, используются те же правила, что и при порождении из одного базового класса.

Проблемы могут возникнуть в следующих случаях:

•Если в порожденном классе используется член с таким же именем, как в одном из базовых классов;

•В нескольких базовых классах определены члены с одинаковыми именами.

В этих случаях необходимо использовать оператор разрешения контекста для уточнения элементам, к которому осуществляется доступ. это оператор “::”

96. Как реализовано приведение типов в C?

97. Что такое статическое приведение типов?

98. Что такое динамическое приведение типов?

99. Что такое константное приведение типов?

100. Что такое интерпретирующее преобразование типов?

101. Как работает преобразование в С-стиле на языке С++?

102. Для чего предназначен механизм RTTI, как его использовать?

Динамическая идентификация типа данных (англ. *run-time type information, run-time type identification, RTTI*) — механизм в некоторых языках программирования, который позволяет определить тип данных переменной или объекта во время выполнения программы.

сложна

В C++ для динамической идентификации типов[1] применяются операторы dynamic_cast и typeid (определён в файле typeinfo.h), для использования которых информацию о типах во время выполнения обычно необходимо добавить через опции компилятора при компиляции модуля.

Оператор dynamic_cast пытается выполнить приведение к указанному типу с проверкой. Целевой тип операции должен быть типом указателя, ссылки или void*.

- Если целевой тип — тип указателя, то аргументом должен быть указатель на объект класса.
- Если целевой тип — ссылка, то аргумент должен также быть соответствующей ссылкой.
- Если целевым типом является void*, то аргумент также должен быть указателем, а результатом операции будет указатель, с помощью которого можно обратиться к любому элементу «самого производного» класса иерархии, который сам не может быть базовым ни для какого другого класса.

Оператор typeid[2] возвращает ссылку на структуру type_info, которая содержит поля, позволяющие получить информацию о типе.

Использование RTTI оказывается необходимым при реализации таких приложений, как отладчики или объектные базы данных, когда тип объектов, которыми манипулирует программа, становится известен только во время выполнения путем исследования RTTI-информации, хранящейся вместе с типами объектов. Однако лучше пользоваться статической системой типов C++, поскольку она безопаснее и эффективнее.

103. Что такое умные указатели?

Объекты, с которыми можно работать как с обычным указателем, но при этом, в отличии от последнего, он представляет некоторый дополнительный функционал (например, автоматическое освобождение закрепленной за указателем области памяти).
Используются для борьбы с утечками памяти.
Особенно удобны в местах возникновения исключений.

104. Опишите принцип работы boost::scoped_ptr.

Запрет на конструктор копирования
Функционал и реализация схожа с описанным выше smart_pointer.
Однако, указатель не может быть скопирован.
Память, на которую указывает scoped_ptr будет гарантированно освобождена при уничтожении объекта указателя или при явном вызове метода reset.
Применяется как указатель-обертка для каких-либо объектов, которые выделяются динамически в начале функции и удаляются в конце, чтобы избавиться от необходимости «ручной» очистки ресурсов.

105. Опишите принцип работы std::auto_ptr.

106. Опишите принцип работы std::shared_ptr.

Самый популярный и широко используемый указатель.

Входит в стандартную библиотеку (добавлен стандартом C++11). Изначально входил в библиотеку boost::shared_ptr.

Умный указатель с подсчетом ссылок на ресурс. Счетчик инкрементируется при каждом вызове либо оператора копирования либо оператора присваивания. Освобождение ресурса происходит когда счетчик ссылок на него будет равен 0.

Также как unique_ptr и auto_ptr предоставляет методы get() и reset().

107. Опишите принцип работы std::weak_ptr.

Входит в стандартную библиотеку (добавлен стандартом C++11). Изначально входил в библиотеку boost::weak_ptr.

Позволяет разрушить циклическую зависимость, которая может возникнуть при использовании std::shared_ptr.

Не позволяет работать с ресурсом напрямую, но обладает методом lock(), который генерирует std::shared_ptr.

108. В чем особенности работы умных указателей с массивами?

109. Какие группы операторов в C++ вам известны

110. Что такое перегрузка операторов, для чего она используется?

Всего лишь более удобный способ вызова функций. Позволяет определять поведение встроенных операторов для объектов пользовательских классов.
Не стоит «увлекаться» перегрузкой операторов при разработке новых классов. Использовать её следует только тогда, когда это упростит написание и чтение кода.
Доступна для пользовательских типов, нельзя перегружать операторы встроенных типов.

111. Для каких типов допустима перегрузка операторов?

Перегрузка операторов в C++
Доступна для пользовательских типов

В C++ можно выделить четыре типа перегрузок операторов:

1.Перегрузка обычных операторов + - * / % ^ & | ~ ! = < > += -= *= /=%= ^= &= |= << >> >>= <<= == != <= >= && || ++ -- , ->* -> ( ) [ ]

2.Перегрузка операторов преобразования типа

3.Перегрузка операторов аллокации и деаллокации new и delete

4.Перегрузка литералов operator""

112. Где может быть объявлена перегрузка оператора?

Перегруженные операторы должны быть нестатической функцией-членом класса или глобальной функцией. Глобальная функция, которой требуется доступ к частным или защищенным членам класса, должна быть объявлена в качестве дружественной функции этого класса. Глобальная функция должна принимать хотя бы один аргумент, имеющий тип класса или перечисляемый тип либо являющийся ссылкой на тип класса или перечисляемый тип.

113. Какие особенности у перегрузки операторов инкремента и декремента?

Операторы инкремента и декремента относятся к особой категории, поскольку имеется два варианта каждого из них:

- преинкрементный и постинкрементный операторы;
- предекрементный и постдекрементный операторы.

При написании функций перегруженных операторов полезно реализовать отдельные версии для префиксной и постфиксной форм этих операторов. Для различения двух вариантов используется следующее правило: префиксная форма оператора объявляется точно так же, как и любой другой унарный оператор; в постфиксной форме принимается дополнительный аргумент типа int.

Примечание
При задании перегруженного оператора для постфиксной формы оператора инкремента или декремента дополнительный аргумент должен быть типа int; при указании любого другого типа выдается ошибка.

114. Как ведут себя операторы с особым порядком вычисления при перегрузке?

Если речь о || и &&, то никак, вычисления производятся в явном виде.

Логические операторы И ИЛИ и оператор ЗАПЯТАЯ имеют особый порядок вычислений.

```
int main() {
  int a = 0;
  int b = 5;
  bool resultAnd = (a != 0) && (b / a);
  bool resultOr = (a == 0) || (b / a);

  foo() && bar();
  foo() || bar();
  foo() ,  bar();
}
MyClass operator&&(MyClass const& first, MyClass const& second) {
  ...
}
```

это чет не то

Особые операторы

В C++ есть операторы, обладающие специфическим синтаксисом и способом перегрузки. Например оператор индексирования []. Он всегда определяется как член класса и, так как подразумевается поведение индексируемого объекта как массива, то ему следует возвращать ссылку.

Оператор запятая

В число «особых» операторов входит также оператор запятая. Он вызывается для объектов, рядом с которыми поставлена запятая (но он не вызывается в списках аргументов функций). Придумать осмысленный пример использования этого оператора не так-то просто.

Оператор разыменования указателя

Перегрузка этих операторов может быть оправдана для классов умных указателей. Этот оператор обязательно определяется как функция класса, причём на него накладываются некоторые ограничения: он должен возвращать либо объект (или ссылку), либо указатель, позволяющий обратиться к объекту.

Оператор присваивания

Оператор присваивания обязательно определяется в виде функции класса, потому что он неразрывно связан с объектом, находящимся слева от "=". Определение оператора присваивания в глобальном виде сделало бы возможным переопределение стандартного поведения оператора "=".

115. Наследует ли производный класс перегруженные операторы?

Все перегруженные операторы, кроме оператора присваивания (operator=), наследуются производными классами.

116. Как защитить объект от копирования?

Сделать private конструктор копирования и оператор =

Это можно просто почитать, тут типа поподробне написано..а ответ одна строчка выше

Внимательный читатель наверняка заметил в посте про реализацию параллельных потоков на C++ следующий фрагмент кода:

```
class Thread {
public:
 ...
private:
 ...
 // Защита от случайного копирования объекта в C++
 Thread(const Thread&);
 void operator=(const Thread&);
};
```

Это определения конструктора копирования и перегруженого оператора присваивания. Причем непосредственно реализаций этих функций нигде нет, только определения. Вопрос: для чего все это?

Давайте разберемся с назнакласса создаете объекты динамически в куче и храните в классе только указатели на них. Указатеть — это базовый тип, и компилятор его нормально скопирует. А вот данные, на которые этот указатель указывает он копировать не будет. В итоге два объекта (старый-оригинал и новый-копия) будут ссылаться на один кусок памяти в куче. Теперь понятно, что при попытке освобождения этой памяти в деструкторе (если вы не забыли этого сделать ;-) кто-то из этих двух объектов попытается освободить уже освобожденную память. Вероятность аварийного завершения программы в этом случае крайне высока, а поиск подобных ошибок может быть крайне долгим и мучительным.

Отсюда мораль: если для вашего класса не заданы конструктор копирования и оператор присваивания (они вам не нужны по смыслу), сделайте им пустые объявления в разделе закрытом разделе (private). Тогда попытка скопировать этот объект сразу приведет к ошибке при компиляции. Во-первых, объявления являются закрытыми (private), и сторонний пользователь вашего класса получит ошибку доступа к закрытым данным класса. Во-вторых, у этих функций нет тел, а значит вы сами не выстрелите себе в ногу, попытавшись случайно скопировать объект данного класса в нем же самом (тут вам private уже не помеха), если вы на это не рассчитывали при проектирование класса.

Лично я делаю так. У меня есть следующий файл:чением этих функций. Их прямая задача уметь копировать объект данного класса. А что произойдет, если вы по какой-то причине не определили конструктор копирования или оператор присваивания (может просто забыли), а пользователь вашего класса решил скопировать объект, возможно даже неосознанно? В этом случае компилятор сам определит конструктор копирования *по умолчанию*, который будет тупо копировать объект байт за байтом без учета смысла копируемых данных. И вам крупно повезет, если все члены-данные вашего класса

являются либо базовыми типами (int, long, char и т.д.), либо имеют корректные конструкторы копирования и операторы присваивания. В этом случае все будет хорошо — базовые типы компилятор умеет копировать правильно, а сложные типы скопируют себя сами через их конструкторы копирования. А представьте, что вы внутри своего

ctorguard.h:

```
#ifndef _EXT_CTOR_GUARD_H
#define _EXT_CTOR_GUARD_H
#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
 TypeName(const TypeName&);            \
 void operator=(const TypeName&)
#endif
```

Теперь определение класса будет выглядеть так:

```
#include "ctorguard.h"
class Thread {
public:
 ...
private:
 ...
 // Защита от случайного копирования объекта в C++
 DISALLOW_COPY_AND_ASSIGN(Thread);
};
```

Теперь вы надежно предохранены.

Кстати, вдогонку. При реализации оператора присваивания надо обязательно проверять — не пытаетесь ли вы копировать объект сам в себя, то есть, не является ли источник копирования самими объектом куда идет копирование. Если это произойдет, вы легко можете получить переполнения стека как самый вероятный исход из-за бесконечного вызова оператора присваивания себя самим.

117. Что такое шаблоны классов?

Шаблоны класса можно использовать для создания семейства классов, воздействующих на тип. Шаблоны класса являются параметризованными типами. Они подразумевают, что для каждого мыслимого значения передаваемых параметров (известных как аргументы шаблона) можно создать отдельный класс.

Аргументы шаблона могут быть типами или константными значениями указанного типа. Например:

```
// class_templates.cpp
template <class T, int i> class TempClass
{
public:
    TempClass( void );
    ~TempClass( void );
    int MemberSet( T a, int b );
private:
    T Tarray[i];
    int arraysize;
};
```

```
int main()
{
}
```

В этом примере, класс-шаблон использует два параметра, тип T и int i. Параметру T можно передать любой тип, включая структуры и классы. Параметру i необходимо передать целочисленную константу. Поскольку i — это константа, определенная во время компиляции. Можно определить членский массив размера i с помощью стандартного объявления массива

118. Что такое шаблоны функций?

Шаблоны функций похожи на шаблоны классов, но определяют семейство функций. С помощью шаблонов функций можно задавать наборы функций, основанных на одном коде, но действующих в разных типах или классах. Следующий шаблон функции меняет местами два элемента.

```
// function_templates1.cpp
template< class T > void MySwap( T& a, T& b ) {
   T c(a);
   a = b;
   b = c;
}
int main() {
}
```

Этот код определяет семейство функций, которые меняют местами значения аргументов. Из этого шаблона можно создавать функции, меняющие местами типы int и long, а также пользовательские типы. Функция MySwap даже меняет местами классы, если правильно определены конструктор копии и оператор присваивания.
Кроме того, шаблон функции не позволяет разработчику менять местами объекты разных типов, поскольку компилятор во время компиляции знает типы параметров a и b.
Хотя эта функция может выполняться нешаблонной функцией с использованием указателей void, версия шаблона типобезопасна. Рассмотрим следующие вызовы:

```
int j = 10;
int k = 18;
CString Hello = "Hello, Windows!";
MySwap( j, k );          //OK
MySwap( j, Hello );      //error
```

Второй вызов MySwap приводит к ошибке времени компиляции, поскольку компилятору не удается создать функцию MySwap с параметрами разных типов. Если бы использовались указатели void, оба вызова функции скомпилировались бы правильно, но функция не работала бы должным образом во время выполнения.
Для шаблона функции можно явно задавать аргументы. Например:

```
// function_templates2.cpp
template<class T> void f(T) {}
int main(int j) {
   f<char>(j);   // Generate the specialization f(char).
   // If not explicitly specified, f(int) would be deduced.
}
```

Если аргумент шаблона задан явно, выполняются обычные неявные преобразования для преобразования аргумента функции в тип соответствующих параметров шаблона функции. В приведенном выше примере компилятор преобразует (char j) в тип int.

119. Как осуществляется вывод аргументов шаблона?

120. Что такое специализация шаблонов?

Шаблоны класса можно настроить для определенных типов или значений аргументов шаблона.Специализация позволяет выполнять настройку кода шаблона для конкретного типа аргумента или значения.Без специализации один и тот же код создается для всех типов, используемых в создании экземпляра шаблона.В специализации, когда используются определенные типы, вместо исходного определения шаблона используется определение для специализации.Специализация имеет то же имя, что и шаблон, который она специализирует.Однако специализация шаблона может по многим параметрам отличаться от исходного шаблона.Например, она может иметь разные члены данных и функции-члены.

121. Назовите хотя бы один пример возникновения undefined behavior.

Неопределённое поведение (англ. undefined behaviour) — свойство некоторых языков программирования (наиболее заметно в Си и Си++) в определённых ситуациях выдавать результат, зависящий от реализации компилятора или заданных ключей оптимизации.

Примеры ситуаций, приводящих к неопределённому поведению:

- Использование переменной до её инициализации. Неопределённое поведение возникает при попытке обращения к переменной.
- Выделение памяти с использованием оператора new [] и последующее её освобождение с использованием оператора delete. Пример: T *p = new T[10]; delete p;. Правильный вариант: T *p = new T[10]; delete [] p;.
- Переменная несколько раз изменяется в рамках одной точки следования. Часто в качестве канонического примера приводят выражение i=i++, в котором происходит присваивание переменной i и её же инкремент. Более подробно с данной разновидностью ошибок можно познакомиться в разделе

122. Что такое шаблон проектирования?

Шаблоны (или паттерны) проектирования описывают типичные способы решения часто встречающихся проблем при проектировании программ.

Паттерны отличаются по уровню сложности, детализации и охвата проектируемой системы. Это как в кулинарии, у вас может быть простой рецепт борща, либо большая карта блюд для званого ужина.

Самые низкоуровневые и простые паттерны — *идиомы*. Они не очень универсальные, так как применимы только в рамках одного языка программирования.

Самые универсальные — *архитектурные паттерны*, которые можно реализовать практически на любом языке. Они нужны для проектирования всей программы, а не отдельных её элементов.
Самое главная причина — паттерны упрощают проектирование и поддержку программ.

- **Проверенные решения.**

Вы тратите меньше времени, используя готовые решение, вместо повторного изобретения велосипеда.

- **Стандартизация кода.**

Вы делаете меньше ошибок, так как используете типовые унифицированные решения, в которых давно найдены все скрытые проблемы.

- **Общий программистский словарь.**

Вы произносите название паттерна, вместо того, чтобы час объяснять другим программистам какой крутой дизайн вы придумали и какие классы для этого нужны.

Недостатки

Костыль для слабого языка программирования

Нужда в паттернах появляется тогда, когда люди выбирают для своего проекта язык или технологию с недостаточным уровнем абстракции. В этом случае, паттерны — это костыль, который придаёт языку суперспособности.

Например, паттерн Стратегия в современных языках можно реализовать простой анонимной (лямбда) функцией.

Впервые эту точку зрения выразил Пол Грэм в эссе «Месть Ботанов».

Приводят к неэффективным решениям

Паттерны пытаются стандартизировать подходы, которые и так уже широко используются.

Эта стандартизация кажется некоторым догмой и они реализуют паттерны «как в книжке», не приспосабливая паттерны к реалиям проекта.

Если у тебя в руках молоток, то все предметы вокруг начинают напоминать гвозди

Похожая проблема возникает у новичков, только-только познакомившихся с паттернами.

Вникнув в паттерны, человек пытается применить свои знания везде. Даже там, где можно было бы обойтись кодом попроще.

123. Какие группы паттернов описаны в книге «Design patterns» GoF?
+ Вопросы по каждому из 23 шаблонов проектирования: название, область применения, решаемая задача, uml-диаграмма (можно не точную), достоинства и недостатки.

Виды
Порождающие

Отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.
Структурные

Отвечают за построение удобных в поддержке иерархий классов.
Поведенческие

Решают задачи эффективного и безопасного взаимодействия между объектами программы.

Порождающие:

1)Фабричный метод

- **Суть паттерна:** Определяет интерфейс для создания объектов, но позволяет подклассам изменять тип создаваемых объектов.

- **Проблема**

Представьте, что вы создаёте программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях, поэтому весь ваш код работает с объектами класса Грузовик
Но ваша программа становится настолько известной, что морские перевозчики выстраиваются в очередь и просят добавить поддержку морской логистики в программу.
Вы радуетесь, потому что, на первый взгляд, всё просто — нужно создать несколько подклассов тут и там. Но вот беда, весь ваш код рассчитан только на грузовики, и для внедрения классов Кораблей придётся перелопатить весь код.

- **Решение**

Паттерн *Фабричный метод* предлагает заменить непосредственное создание объекта-продукта (через оператор new) вызовом особого «фабричного» метода, который и будет создавать продукт.

```
// Было:
Truck t = new Truck();

// Стало:
Transport t = createVehicle(); // return new Truck()
```

На первый взгляд, такое перемещение может показаться бессмысленным. Однако теперь вы сможете переопределить фабричный метод в подклассах, чтобы изменить тип создаваемого продукта.

```
class Logistics
    abstract method createVehicle(): Transport

class RoadLogistics
    method createVehicle(): Transport is
        return new Truck()

class SeaLogistics
    method createVehicle(): Transport is
        return new Ship()"
```

Чтобы эта система работала, все создаваемые продукты должны иметь общий интерфейс (например, Транспорт). Конкретные возвращаемые продукты могут быть разными, покуда они реализуют общий интерфейс (например, Грузовик и Корабль оба реализуют интерфейс Транспорт).

```
interface Transport
    method deliver(something, destination)

class Truck implements Transport
    method deliver(something, destination) is
        // Доставить по земле.

class Ship implements Transport
    method deliver(something, destination) is
        // Доставить по морю.
```

Пользователи фабричного метода всё равно какой конкретно продукт вернул метод, так как они начнут работать со всеми продуктами через общий интерфейс.

```
class Logistics
    abstract method createVehicle(): Transport

class SomeClientClass
    someClientMethod(Logistics logistics, package, destination) is
        // создаст какой-то транспорт
        Transport t = logistics.createVehicle();
        // не важно какой, т.к. все транспорты имеют метод deliver
        t.deliver(package, destination);
```

- **Структура**

- **Применимость**

❖ ***Вам заранее неизвестны типы и зависимости объектов, с которыми будет работать ваш код.*** Например, классы чтения данных из файлов, базы данных или сети будут иметь разные зависимости. Причём клиентскому коду необязательно знать о них. Все подробности инициализации этих компонентов могут быть спрятаны внутри фабричных методов, определённых в разных подклассах

❖ ***Вы делаете свой фреймворк и хотите дать возможность пользователям расширять его части***. Пользователи могут расширять классы вашего фреймворка через наследование, но как сделать так, чтобы фреймворк создавал объекты из этих новых классов вместо стандартных? Ответ: дать пользователям возможность расширять не только желаемые компоненты, но и классы, которые создают эти компоненты. А для этого создающие классы должны иметь конкретные создающие методы, которые можно определить. Например, вы используете готовый UI фреймворк для своего приложения. Но вот беда, требуется иметь круглые кнопки, вместо стандартных прямоугольных. Вы создаёте класс КруглаяКнопка. Но как сказать фреймворку, чтобы он теперь создавал круглые кнопки, вместо стандартных. Для этого вы создаёте подкласс ФреймворкСКруглымиКнопками из базового класса фреймворка, переопределяете в нём метод создатьКнопку и вписываете туда создание своего класса кнопок. Затем, используете ФреймворкСКруглымиКнопками вместо обычного.

❖ ***Вы планируете повторно использовать готовые объекты, вместо создания новых.*** Например, если вы имеете дело с тяжёлыми ресурсоёмкими объектами такими как подключение к базе данных, файловой системе и прочее. Представьте, сколько действий вам нужно совершить, чтобы получить такую функциональность. Сперва, нужно определить, есть ли уже готовый объект и свободен ли он, выбрать его из списка, вернуть пользователю. Если свободных объектов нет — создать новый. Но вы не можете делать эти проверки в конструкторе, так как он по определению должен вернуть новый объект. Кроме того, клиентскому коду, использующему ваш класс не нужно знать обо всех этих подробностях, а значит, должен быть один метод, из которого они смогут получать как существующие, так и новые объекты. Это и будет фабричный метод.

- **Преимущества и недостатки**

“+”	“-”
• Реализует *принцип открытости/закрытости*. • Позволяет не привязывать код приложения к конкретным классам продуктов. • Упрощает программу за счёт выноса кода создания продуктов в одно место. • Упрощает внесение в программу новых продуктов.	• Для указания типа создаваемых продуктов требуются подклассы.

2)Абстрактная фабрика

- **Суть паттерна:** Позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам продуктов.
- **Проблема**

Представьте, что вы пишете симулятор мебельного магазина. Ваш код содержит:

1. Семейство зависимых продуктов. Скажем, Кресло + Диван + Столик.
2. Несколько вариаций таких семейств. Например, продукты Кресло, Диван и Столикпредставлены в вариациях Набор-из-IKEA, Викторианская-Мебель, Старый-советский-гарнитур.

Вам нужен такой способ создавать объекты продуктов, чтобы:

1. Они сочетались с другими продуктами того же семейства. *Покупатели расстраиваются, если получают несочитающуюся мебель.*
2. Было бы возможно добавить новый продукт или семейство продуктов, не залезая в существующий код. *Вы часто меняете поставщиков, а они предлагают совершенно разные модели мебели.*

Решение

Для начала, паттерн *Абстрактная фабрика* предлагает собрать семейства продуктов в отдельные иерархии классов с общими интерфейсами. Так, стулья будут иметь общий интерфейс Стул, столы реализуют интерфейс Стол и так далее.

Далее, вы создаёте общий интерфейс для фабрик — «абстрактную фабрику». Она знает из каких классов состоит семейство продуктов и описывает операции создатьКресло, создатьДиван и создатьСтолик. Эти операции должны возвращать абстрактные продукты — Кресла, Диваны и Столики.

Как насчёт вариаций продуктов? Каждой вариации соответствует собственный класс-фабрика, реализующий интерфейс абстрактной фабрики. Например, `ФабрикаIKEA`, `ФабрикаАнтиквариата`,`ФабрикаСоветскойМебели`. Эти классы знают какие конкретно продукты следует создавать.

Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Это позволит подавать в ваши классы любой тип фабрики и производить любые продукты, ничего не ломая.

Например, клиентский код просит фабрику сделать стул. Он не знает какого типа фабрика это была. Он не знает, получит ли викторианский стул или стул из IKEA. Для него важно, чтобы на этом стуле можно сидеть, и чтобы этот стул отлично смотрелся с диваном той же фабрики.

Осталось прояснить последний момент — кто создаёт объекты конкретных фабрик, если клиентский код работает только с интерфейсами фабрик? Обычно программа создаёт конкретный объект фабрики при запуске, причём тип фабрики выбирается исходя из параметров окружения или конфигурации.

- **Структура**

- **Применимость:**

❖ **Бизнес-логика программы должна работать с разными видами связанных друг с другом продуктов, не завися от конкретных классов продуктов.** Абстрактная фабрика скрывает от клиентского кода подробности того, как и какие конкретно объекты будут созданы. Но при этом клиентский код может работать со всеми типами создаваемых продуктов, так как их общий интерфейс был заранее определён.

- ❖ **Если в программе уже используется Фабричный метод, но очередные изменения предполагают введение новых типов продуктов.** В хорошей программе, каждый *класс отвечает только за одну вещь*. Множество фабричный методов в клиентском классе способны затуманить его основную функцию. Поэтому им нет смысл вынести всю логику создания продуктов в отдельную иерархию классов — в *Абстрактную фабрику*.
- **Преимущества и недостатки**

“+”	“-”
• Реализует *принцип открытости/закрытости*. • Позволяет конструировать семейства продуктов, гарантируя их сочетаемость. • Избавляет от жёсткой зависимости между компонентами программы. • Разделяет ответственность между классами	• Усложняет код программы за счёт множества дополнительных классов.

3) Строитель

- **Суть паттерна:** Позволяет конструировать объекты пошагово и производить различные продукты, используя один и тот же порядок строительства.
- **Проблема**

У вас есть объект, который требует кропотливой пошаговой инициализации многих полей и вложенных объектов. Этот код либо упрятан в конструктор с множеством параметров, либо распылён по всему клиентскому коду.

Например, один из возможных продуктов — Дом. Чтобы его построить нужно поставить 4 стены, установить двери и два окна, постелить крышу. Но в некоторых случаях вам нужен дом побольше, посветлее, с чёрным ходом и прочим добром.

Проблему можно решить в лоб, создав подклассы домов для всех этих вариаций. Но в реальности их будет слишком много, да и в придачу вы получите все проблемы, связанные с наследованием.

С другой стороны, вы можете создать гигантский конструктор Дома, принимающий уйму параметров для контроля над создаваемым продуктом. Но большая часть этих параметров будет простаивать, а вызовы конструктора будут выглядеть монструозно из-за длинного списка параметров.

- **Решение**

Паттерн Строитель предлагает определить и формализовать все шаги конструирования продукта в общем интерфейсе — *Строителе*.

Чтобы создать продукт, потребуется поочерёдно вызывать методы класса, реализующего интерфейс Строителя.

В программе может быть несколько видов Строителей, при условии, что все они реализуют общий интерфейс. Они могут по-разному реализовывать шаги строительства (например, делать стены из камня, бронированные двери и прочее). За счёт общего интерфейса, их можно взаимозаменять, чтобы получать продукты с разными свойствами.

Чтобы не загромождать клиентский код вызовами методов Строителя, можно ввести промежуточный класс Директор. В этом случае Директор будет отвечать за порядок вызова шагов, а Строитель — за реализацию этих шагов.

- **Структура**

- **Применимость:**

❖ **Проблема телескопического конструктора** Это когда у вас есть один конструктор с десятью опциональными параметрами. Такой конструктор неудобно вызывать, поэтому вас есть ещё десять вспомогательных альтернативных конструкторов с меньшим количеством параметров. Всё что они делают — переадресуют вызов к главному конструктору, подавая какие-то значения по умолчанию в опциональные параметры главного конструктора.

❖ **Паттерн Строитель позволяет создавать объекты, вызывая минимум шагов, требуемых для этого.** Именно поэтому Строители часто используют для создания неизменяемых объектов.

❖ **Ваш код создаёт разные представления какой-то одной сущности (например, деревянные и железо-бетонные дома). Создание этой сущности содержит одинаковые этапы, но отличается в деталях. Конечные продукты могут не быть потомками одного класса или интерфейса.** В этом случае, паттерн Строитель идеально подходит, чтобы собрать в одном месте код конструирования ваших сущностей.

- **Для каждой сущности нужно создать собственного Строителя, а этапы строительства переедут в общий класс-Директор.**Требуется конвертировать одну сущность в множество других. Например, HTML документ требуется превращать в PDF, DOC и TXT. Это классический пример применения Строителей. Абстрактный строитель будет иметь методы конвертации для каждого HTML-тега. Другие форматы поддерживают не все HTML теги, но они смогут определить только те шаги строительства, которые поддерживают.
- **Требуется создание дерева Компоновщика или другого составного объекта.** Паттерн Строитель конструирует объекты пошагово, а не за один проход. Более того, шаги строительства можно выполнять рекурсивно. А без этого безопасно не построить древовидную структуру вроде Компоновщика.
- **Преимущества и недостатки**

“+”	“-”
• Позволяет создавать продукты пошагово. • Позволяет использовать один и тот же код для создания различных продуктов. • Изолирует сложный код сборки продукта от его основной бизнес-логики.	• Усложняет код программы за счёт дополнительных классов.

4) Прототип

- **Суть паттерна:** Позволяет копировать существующие объекты, не вдаваясь в подробности их реализации.
- **Проблема**

Как скопировать объект? Нужно создать пустой объект такого же класса, а затем поочерёдно копировать значения всех полей.

Но не каждый объект удастся скопировать таким образом, ведь часть его состояния может быть приватной. Не говоря уже о том, что вам придётся жёстко привязать код к конкретным классам каждого копируемого объекта.

- **Решение**

Паттерн Прототип поручает создание копий самим копируемым объектам.

Он вводит общий интерфейс для всех объектов, поддерживающих клонирование. Это позволяет копировать объекты, не привязываясь к их конкретным классам. Обычно такой интерфейс имеет все один метод clone.

Реализация этого метода в разных классах очень схожа. Метод создаёт новый объект текущего класса и копирует в него значения всех полей объекта. Так получится скопировать даже приватные поля, так как большинство языков программирования разрешает доступ к приватным полям отдельного объекта текущего класса.

Объект, который копируют, называется *прототипом* (откуда и название паттерна). Когда объекты программы содержат сотни полей и тысячи возможных конфигураций, прототипы могут служить своеобразной альтернативой созданию подклассов. В этом случае, все возможные прототипы заготавливаются и настраиваются на этапе инициализации программы. Потом, когда программе нужен новый объект, она создаёт копию из приготовленного прототипа.

- **Структура**

- **Применимость:**

❖ **Ваш код не должен зависеть от создаваемых продуктов. Например, если тип создаваемого продукта нельзя определить наперёд или он может изменятся во время выполнения.** Паттерн прототип предоставляет клиенту общий интерфейс для работы со всеми прототипами. Клиенту не нужно зависеть от всех классов копируемых объектов, а только от интерфейса клонирования.

❖ **Вам нужно избежать разрастания иерархии классов продуктов, причём продукты отличаются только внутренним состоянием.** Паттерн прототип предлагает создать набор объектов-прототипов, вместо создания иерархии подклассов продуктов. Таким образом, вместо прямого создания объектов из подклассов, вы будете копировать существующие объекты-прототипы, в которых уже настроено внутреннее состояние. Это позволит избежать взрывного роста количества классов в программе и уменьшить её сложность.

- **Преимущества и недостатки**

“+”	“-”
● Позволяет клонировать объекты, без привязки к их конкретным классам. ● Меньше повторяющегося кода инициализации объектов.	● Сложно клонировать составные объекты, имеющие ссылки на другие объекты.

● Ускоряет создание объектов. ● Альтернатива созданию подклассов для конструирования сложных объектов.	

5) Одиночка

- **Суть паттерна:** Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.
- **Проблема**

Одиночка решает сразу две проблемы (нарушая *принцип единственной ответственности*):

1. *Гарантирует единственный экземпляр класса.* Чаще всего это полезно для доступа к какому-то общему ресурсу, например базе данных.

Вы создали объект, а через некоторое время, пробуете создать ещё один. В этом случае, хотелось бы получить старый объект, вместо создания нового. Это невозможно реализовать с помощью обычного конструктора, так как конструктор класса всегда возвращает новый объект.

2. *Предоставляет глобальную точку доступа*. Это не просто глобальная переменная, в которую вы засунули нужный объект. Глобальная переменна не защищена от записи и любой код может подменить её значение.

Но есть и другая грань этой проблемы. Хотелось бы хранить в одном месте и код, который гарантирует производство лишь одного экземпляра класса.

- **Решение**

Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

Если у вас есть доступ к классу-одиночке, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его не вызвали, он всегда будет отдавать один и тот же объект.

- **Структура**

- **Применимость:**
- ❖ **В программе должен быть единственный экземпляр какого-то класса, доступный всем клиентам. Например, общий доступ к базе данных из разных частей программы.** Одиночка скрывает от клиентов все способы создания нового объекта, кроме специального метода. Этот метод либо создаёт объект, либо отдаёт существующий объект, если он уже был создан.
- ❖ **Вам хочется иметь больше контроля над глобальными переменными.** В отличие от глобальных переменных, с Одиночкой вы абсолютно уверены в том, что в любой момент существует только экземпляр класса. Тем не менее в любой момент вы можете расширить это ограничение и позволить любое количество объектов-Одиночек, поменяв код в одном месте (метод getInstance()).
- **Преимущества и недостатки**

“+”	“-”
• Позволяет отложенную инициализацию	• Нарушает *принцип единственной обязанности класса*. • Маскирует плохой дизайн. • Проблемы мультипоточности. • Проблемы юнит-тестирования

Структурные:

1)Адаптер

- **Суть паттерна:** Обеспечивает совместную работу классов с несовместимыми интерфейсами.

- **Проблема**

К вашему приложению, работающему с данными в XML, нужно прикрутить стороннюю библиотеку, работающую в JSON.

Например, ваше приложение скачивает биржевые котировки из нескольких источников в XML и рисует красивые графики. В какой-то момент вы решаете улучшить приложение, применив

стороннюю библиотеку аналитики. Но вот беда, библиотека поддерживает только JSON формат данных.

Вы смогли бы просто переписать её для поддержки XML. Но во-первых, это может нарушить работу существующего кода, который уже зависит от библиотеки. А во-вторых, у вас может просто не быть доступа к её исходникам.

- **Решение**

Вы можете создать Адаптер. Это объект, который оборачивает другой объект с неудобным интерфейсом, но сам реализует интерфейс, понятный клиенту.

Адаптер получает вызов от клиента и переводит его в формат понятный обёрнутому объекту. Адаптер может конвертировать не только данные из одного формата в другой, но и вызывать разные методы оборачиваемого объекта, если их интерфейсы кардинально отличаются.

Иногда возможно создать даже *двухсторонний адаптер*, который работал бы в обе стороны.

Таким образом, в приложении биржевых котировок, вы могли бы создать класс XML_To_JSON_Adapter, который бы оборачивал класс библиотеки аналитики. Ваш код посылал бы запросы этому объекту в XML, а адаптер бы сначала транслировал входящие данные в JSON, а затем передавал бы их определённым методам библиотеки

- **Структура**

- **Применимость**

- ❖ **Вы хотите использовать существующий класс, но его интерфейс не соответствует остальному коду приложения.** Адаптер позволяет создать для существующего класса новый интерфейс-прокладку, который будет превращать вызовы в новом формате к старому.
- ❖ **Вам нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности. Причём расширять суперкласс вы не можете.** Вы могли бы создать ещё один уровень подклассов, и добавить в них недостающую функциональность. Но при этом придётся дублировать один и тот же код в обоих ветках подклассов. Более элегантное решение — поместить недостающую функциональность в адаптер и приспособить его для работы с суперклассом. Такой адаптер сможет работать со всеми подклассами иерархии. Это решение будет сильно напоминать паттерн Посетитель.
- **Преимущества и недостатки**

“+”	“-”
• Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.	• Усложняет код программы за счёт дополнительных классов.

2)Мост

- **Суть паттерна:** Разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

- **Проблема**

Абстракция? Реализация?! Звучит пугающе! Чтобы понять о чём идёт речь, давайте разберём очень простой пример.

У вас есть класс геометрических Фигур, который имеет подклассы Квадрат и Треугольник. Вы хотите расширить иерархию фигур по цвету, то есть иметь Красные и Синие фигуры.

Чтобы всё это объединить, вам придётся создать 4 комбинации подклассов вроде СинииеКвадраты и КрасныеТреугольники.

Но при добавлении новых видов фигур и цветов количество комбинаций будет расти в геометрической прогрессии. Чтобы добавить подкласс Круга, придётся создать сразу два новых класса под каждый цвет. После этого новый цвет потребует создания уже трёх классов, для всех видов фигур. Чем дальше, тем хуже.

- **Решение**

Корень проблемы заключается в том, что мы пытаемся расширить классы сразу в двух независимых плоскостях — по виду и по цвету. Именно это приводит к разрастанию дерева классов.

Паттерн Мост предлагает заменить наследование делегированием. Для этого нужно выделить одну из таких «плоскостей» в отдельную иерархию и ссылаться на объект этой иерархии, вместо хранения его состояния и поведения внутри одного класса.

Таким образом, мы можем сделать Цвет отдельным классом с подклассами Красный и Синий. Класс Фигур получит ссылку на объект Цвета и сможет делегировать ему работу если потребуется. Такая связь и станет мостом между Фигурами и Цветом. При добавлении новых классов цветов, не потребуется трогать классы фигур и наоборот.

Абстракция и Реализация

Эти термины были введены в книге GoF при описании Моста. На мой взгляд, они выглядят слишком академичными, делая описание паттерна сложнее, чем он есть на самом деле. Помня о примере с фигурами и цветами, давайте все же разберемся что имели в виду авторы GoF.

Итак, «Абстракция» (или «интерфейс») — это образный слой управления чем-либо. Он не делает работу самостоятельно, а делегирует её **слою**«реализации» (иногда называемому «платформой»). Только не путайте эти термины с *интерфейсами* или *абстрактными классами* из вашего языка программирования, это не одно и то же.

Если говорить о реальных программах, то абстракцией может выступать графический интерфейс программы, а реализацией — API к которому интерфейс обращается по реакции на действия пользователя.

Вы можете развивать программу в двух разных направлениях:

- иметь несколько различных интерфейсов (например, для простых пользователей и администраторов).

поддерживать много видов API (например, работать под Windows, Linux и MacOS).

Такая программа может выглядеть как один большой клубок кода, в котором намешаны условные операторы слоёв интерфейса и API операционных систем.

Клубок можно попытаться распутать, выделив подклассы для каждой из вариаций интерфейса-платформы. Но с этим возникнет другая проблема — взрывной рост классов-комбинаций, которую мы уже рассмотрели выше.

Паттерн Мост предлагает распутать клубок этого кода и выделить из него две части:

- Абстракцию: слой GUI приложения.
- Реализацию: слой взаимодействия с операционной системой.

Абстракция будет делегировать работу одному из объектов-Реализаций. Реализации можно будет взаимозаменять, если все они будут иметь общий интерфейс.

Вы сможете изменять интерфейс приложения, не трогая код работы с ОС. И наоборот, сможете добавить поддержку новой операционной системы, создав подкласс реализации, не трогая классы абстракции.

- **Структура**

- **Применимость**

❖ **У вас есть один монолитный класс, который содержит несколько различных реализаций какой-то функциональности (например, может работать с разными системами баз данных). В коде класса тяжело разобраться, и это затягивает разработку. Кроме того, изменения, вносимые в одну из реализаций, приводят к редактированию всего класса, что может привести к ошибкам.** Мост разделяет монолитный класс на несколько отдельных иерархий. После этого вы можете менять их код независимо друг от друга. Это упрощает работу над кодом и уменьшает вероятность внесения ошибок.

❖ **Класс нужно расширять в двух независимых плоскостях.** Мост предлагает выделить одну из таких плоскостей в отдельную иерархию классов, храня ссылку на один из её объектов в первоначальном классе.

❖ **Вы хотите, чтобы реализацию можно было бы изменять во время выполнения программы.** Мост позволяет заменять реализацию даже во время выполнения программы, так как конкретная реализация не «вшита» в класс абстракции.

Кстати, из-за этого пункта Мост часто путают со Стратегией. Обратите внимания, что у Моста этот пункт стоит на последнем месте по значимости, так как его главная задача — структурная.

- **Преимущества и недостатки**

“+”	“-”
• Платформо-независимость.	• Усложняет код программы за счёт

• Реализует *принцип открытости/закрытости*. • Скрывает лишние или опасные детали реализации от клиента.	дополнительных классов.

3) Компоновщик

Суть паттерна: Группирует объекты в древовидные структуры и позволяет работать с ними так, если бы это был единичный объект.

- **Проблема**

Паттерн Компоновщик имеет смысл только тогда, когда основная модуль вашей программы может быть структурирована в виде дерева.

Например, есть два объекта — Продукт и Коробка. Коробка может содержать несколько Продуктов и других Коробок поменьше. Те, в свою очередь, тоже содержат либо Продукты, либо Коробки и так далее.

Теперь, предположим, ваши Продукты и Коробки могут быть частью заказов.

Ваша задача в том, чтобы узнать цену всего заказа. Причём в заказе может быть как просто Продукт без упаковки, так и пустая или составная Коробка.

Если решать задачу в лоб, то вам потребуется открыть все коробки заказа, перебрать все продукты и посчитать их суммарную цену.

Но это слишком хлопотно.

- **Решение**

Компоновщик предлагает рассматривать Продукт и Коробку через единый интерфейс с общим методом получитьЦену().

Продукт просто вернёт свою цену. Коробка спросит цену каждого предмета внутри себя и вернёт сумму результатов. Если одним из внутренних предметов окажется коробка поменьше, она тоже будет перебирать своё содержимое, и так далее, пока не посчитаются все составные части.

Для вас, клиента, главное, что теперь не нужно ничего о структуре заказов. Вы вызываете метод получитьЦену(), он возвращает цифру, а вы не тонете в горах картона и скотча.

- **Структура**

- **Применимость**
- ❖ **Нужно представить древовидную структуру объектов — контейнеры и содержимое.**

Паттерн Компоновщик предлагает хранить в составных объектах ссылки на другие простые или составные объекты. Те, в свою очередь, тоже могут хранить свои вложенные объекты и так далее. В итоге вы можете строить сложную древовидную структуру данных, используя всего два вида объектов.

- ❖ **Клиенты должны единообразно трактовать простые и составные объекты.**

Благодаря тому, что простые и составные объекты реализуют общий интерфейс, клиенту безразлично с каким именно объектом ему предстоит работать.

- **Преимущества и недостатки**

“+”	“-”
• Упрощает архитектуру клиента при работе со сложным деревом компонентов. • Облегчает добавление новых видов компонентов.	• Создаёт слишком общий дизайн классов.

4) Декоратор

Суть паттерна: Динамически добавляет объектам новую функциональность, оборачивая их в полезные «обёртки».

- **Проблема**

Вам нужно динамически добавлять и снимать с объекта новые обязанности таким образом, чтобы он оставался совместим с остальным кодом программы.

Наследование — первое что приходит в голову, если вам нужно добавить новое поведение объекту. Но механизм наследования статичен — в программу нельзя на лету добавлять подклассы.

- **Решение**

Декоратор имеет ещё одно название — *Обёртка*. Оно удачнее описывает суть паттерна. Итак, с Декоратором вы помещаете целевой объект в другой объект-обёртку, который расширяет базовое поведение объекта.

Оба объекта имеют общий интерфейс, поэтому пользователю всё равно с чем работать — с чистым объектом или обёрнутым.

Вы можете использовать несколько разных обёрток одновременно — результат будет иметь функции всех обёрток сразу.

- **Структура**

- **Применимость**

❖ **Вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует.**

Объекты помещают в обёртки, имеющие дополнительные поведения. Обёртки и сами объекты имеют одинаковый интерфейс, а значит для клиентов нет разницы с чем работать — обычным или обёрнутым объектом.

❖ **Если нельзя расширить обязанности объекта с помощью наследования.**

Во многих языках программирования есть ключевое слово final, которое может заблокировать наследование класса. Расширить такие классы можно только с помощью Декоратора.

- **Преимущества и недостатки**

“+”	“-”
• Большая гибкость, чем у наследования. • Позволяет добавлять обязанности на лету. • Можно добавлять несколько новых обязанностей сразу. • Позволяет иметь несколько мелких объектов вместо одного объкта на все случаи жизни.	• Трудно конфигурировать многократно обёрнутые объекты. • Обилие крошечных классов.

5) Фасад

Суть паттерна: Предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

- **Проблема**

Вашему коду приходится работать с большим количеством объектов некой сложной библиотеки или фреймворка. Вы должны самостоятельно инициализировать эти объекты, следить за правильным порядком зависимостей и так далее.

В результате, бизнес-логика ваших классов тесно переплетается с деталями реализации сторонних классов. Такой код довольно сложно понимать и поддерживать.

- **Решение**

Фасад — это простой интерфейс работы со сложной подсистемой, содержащей множество классов. Фасад может иметь урезанный интерфейс, не имеющий 100% функциональности, которую можно достичь, используя сложную подсистему напрямую. Но он предоставляет именно те фичи, которые нужны клиенту, и скрывает все остальное.

Фасад полезен, если вы используете какую-то сложную библиотеку со множеством подвижных частей, но вам нужна только часть её возможностей.

К примеру, программа, заливающая видео котиков в социальные сети, может использовать профессиональную библиотеку сжатия видео. Но все что нужно клиентскому коду этой программы — простой метод encode(filename, format). Создав класс с таким методом, вы реализуете свой первый фасад.

Аналогия из жизни

Заказ товаров по телефону

Когда вы звоните в магазин и делаете заказ по телефону, сотрудник службы поддержки является вашим фасадом ко всем службам и отделам магазина.

Он предоставляет вам простой интерфейс к системе создания заказа, платёжной системе и отделу доставки.

- **Структура**

- **Применимость**

❖ **Если нужно представить простой или урезанный интерфейс к сложной подсистеме.** Часто подсистемы усложняются по мере развития. Применение большинства паттернов приводит к появлению меньших классов, но в бóльшем количестве. Такую подсистему проще повторно использовать и настраивать под конкретные нужды, но вместе с тем применять подсистему без настройки становится труднее. Фасад предлагает некоторый вид системы по умолчанию, устраивающий большинство клиентов.

❖ **Если вы хотите разложить подсистему на отдельные слои.** Используйте фасады для определения точек входа на каждый уровень подсистемы. Если подсистемы зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

- **Преимущества и недостатки**

“+”	“-”
● Изолирует клиентов от компонентов системы. ● Уменьшает зависимость между подсистемой и клиентами.	● Фасад рискует стать «божественным объектом», привязанным ко всем классам программы.

6) Легкоовес

Суть паттерна: Позволяет иметь громадное количество объектов, экономя оперативную память за счёт разделения общего состояния объектов между собой.

- **Проблема**

На досуге вы решили написать небольшую игру-стрелялку, в которой игроки перемещаются по карте и стреляют друг в друга. Фишкой игры должна была стать реалистичная система частиц. Пули, снаряды, осколки от взрывов — всё это должно красиво летать и радовать взгляд.

Игра отлично работала на вашем мощном компьютере. Однако ваш друг сообщил, что игра начинает тормозить и вылетает через несколько минут после запуска.

Покопавшись в логах, вы обнаружили, что игра вылетает из-за недостатка оперативной памяти. И действительно, каждая частица представлена собственным объектом, имеющим множество данных.

В определённый момент, когда побоище на экране достигает кульминации, новые объекты частиц уже не помещаются в оперативную память компьютера и программа вылетает.

- **Решение**

Если внимательно посмотреть на класс частиц, то можно заметить, что цвет и спрайтзанимают больше всего памяти. Более того, они хранятся в каждом объекте, хотя фактически их значения

одинаковые для большинства частиц.

Остальное состояние объектов — координаты, вектор движения и скорость отличаются для всех частиц. Таким образом, эти поля можно рассматривать как контекст, в котором частица используется. А цвет и спрайт — это данные, не изменяющиеся во времени.

Неизменяемые данные объекта принято называть «внутренним состоянием». Все остальные данные — это «внешнее состояние».

Паттерн Легковес предлагает не хранить в классе внешнее состояние, а передавать его в те или иные методы через параметры. Таким образом, одни и те же объекты можно будет повторно использовать в различных контекстах. Но главное, понадобится гораздо меньше объектов, ведь они теперь будут отличаться только внутренним состоянием, а оно имеет не так много вариаций.

В нашем примере с частицами, достаточно будет оставить всего три объекта с отличающимися спрайтами и цветом — для пуль, снарядов и осколков. Несложно догадаться, что такие облегчённые объекты называют «легковесами» (название пришло из бокса и означает весовую категорию до 50 кг).

Хранилище внешнего состояния

Но куда переедет внешнее состояние? Ведь кто-то должен его хранить. Чаще всего, их перемещают в контейнер, который управлял объектами до применения паттерна. В нашем случае, это был главный объект игры. Вы могли бы создать в нём дополнительные поля-массивы для хранения координат, векторов и скоростей. Плюс, понадобится ещё один массив для хранения ссылок на объекты-легковесы, соответствующие той или иной частице. Но более элегантным решением было бы создать дополнительный класс-контекст, который связывал внешнее состояние с тем или иным легковесом. Это позволит обойтись только одним полем массивом в классе контейнера. «Но погодите-ка, нам потребуется столько же этих объектов, сколько было в самом начале!» — скажете вы и будете правы! Но дело в том, что объекты-контексты занимают намного меньше места, чем первоначальные. Ведь самые тяжёлые поля остались в легковесах (простите за каламбур), и сейчас мы будем ссылаться на эти

объекты из контекстов вместо того, чтобы хранить дублирующее состояние.

Неизменяемость Легковесов

Так как объекты легковесов будут использованы в разных контекстах, вы должны быть уверены в том, что их состояние невозможно изменять после создания. Всё внутреннее состояние легковес должен получать через параметры конструктора. Он не должен иметь сеттеров и публичных полей.

Фабрика Легковесов

Для удобства работы с легковесами и контекстами можно создать фабричный метод, принимающий в параметрах всё внутреннее (а иногда и внешнее) состояние желаемого объекта. Главная польза от этого метода в том, чтобы искать уже созданные легковесы с таким же внутренним состоянием, что и требуемое. Если легковес находится, его можно повторно использовать. Если нет — просто создаём новый. Обычно этот метод добавляют в контейнер легковесов, либо создают отдельный класс-фабрику. Его даже можно сделать статическим и поместить в класс легковесов.

Структура

- **Применимость**
- ❖ **Экономия системных ресурсов.**

Эффективность паттерна Легковес во многом зависит от того, как и где он используется. Применяйте этот паттерн, когда выполнены все перечисленные условия:

- в приложении используется большое число объектов;
- из-за этого высоки расходы оперативной памяти;
- большую часть состояния объектов можно вынести за пределы их классов;
- многие группы объектов можно заменить относительно небольшим количеством разделяемых объектов, поскольку внешнее состояние вынесено.
- **Преимущества и недостатки**

“+”	“-”
● Экономит оперативную память.	● Расходует процессорное время на поиск/вычисление контекста. ● Усложняет код программы за счёт множества дополнительных классов.

7) Заместитель

Суть паттерна: Оборачивает полезный объект или сервис специальным объектом-заменителем, который «притворяется» оригиналом и перехватывает все вызовы к нему, а затем после некоторой обработки, направляет их обёрнутому объекту.

- **Проблема**

Для чего вообще контролировать доступ к объектам? Рассмотрим такой пример: у вас есть внешний ресурсоёмкий объект, который нужен не все время, а изредка.

Мы могли бы не создавать этот объект в самом начале программы, а только когда он кому-то реально понадобится. Каждый клиент объекта получил бы некий код отложенной инициализации. Но, вероятно, это привело бы к множественному дублированию кода.

В идеале, этот код хотелось бы поместить прямо в служебный класс, но это не всегда возможно. Например, код класса может находится в закрытой сторонней библиотеке.

- **Решение**

Паттерн заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента, объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.

Но в чём же здесь польза? Вы могли бы поместить в класс заместителя какую-то промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте. А благодаря одинаковому интерфейсу, объект заместитель можно передатьдать в любой код, ожидающий сервисный объект.

Аналогия из жизни

Банковский чек

Банковский чек — это заместитель пачки наличности. И чек, и наличность имеют общий интерфейс — ними можно оплачивать товары.

Для покупателя польза в том, что не надо таскать с собой тонны наличности. А владелец магазина может превратить чек в зелёные бумажки, обратившись в банк.

- **Структура**

- **Применимость**

❖ **Ленивая инициализация (виртуальный прокси). Когда у вас есть тяжёлый объект, грузящий данные из файловой системы или базы данных.** Вместо того чтобы грузить данные сразу после старта программы, можно сэкономить ресурсы и создать объект тогда, когда он действительно понадобится.

❖ **Защита доступа (защищающий прокси). Когда в программе есть разные типы пользователей и вам хочется защищать объект от неавторизованного доступа. Например, если ваши объекты — это важная часть операционной системы, а пользователи — сторонние программы (хорошие или вредоносные).** Прокси может проверять доступ при каждом вызове и передавать выполнение служебному объекту, если доступ разрешён.

❖ **Локальный запуск сервиса (удалённые прокси). Когда настоящий сервисный объект находится на удалённом сервере.** В этом случае заместитель транслирует запросы клиента в вызовы по сети, в протоколе понятном удалённому сервису.

❖ **Кеширование объектов («умная» ссылка). Когда нужно кешировать результаты запросов клиентов и управлять их жизненным циклом.** Заместитель может подсчитывать количество ссылок на сервисный объект, которые были отданы клиенту и остаются активными. Когда все ссылки освобождаются — можно будет освободить и сам сервисный объект (например, закрыть подключение к базе данных). Кроме того, Заместитель может отслеживать, не менял ли клиент сервисный объект. Это позволит использовать объекты повторно и здорово экономить ресурсы, особенно если речь идёт о больших ресурсоёмких сервисах.

❖ **Логирование запросов (логирующий прокси). Когда требуется хранить историю обращений к сервисному объекту.** Заместитель может сохранять историю обращения клиента к сервисному объекту.

- **Преимущества и недостатки**

“+”	“-”
• Позволяет контролировать над сервисный объект, незаметно для клиента. • Может работать, даже если сервисный объект ещё не создан. • Может контролировать жизненный цикл служебного объекта.	• Увеличивает время отклика от сервиса

Поведенческие:

1)Цепочка обязанностей

- **Суть паттерна:** Связывает объекты-получатели в цепочку и передаёт запрос вдоль этой цепочки, пока его не обработают.

Избавляет от жёсткой привязки отправителя запроса к его получателю, позволяя выстраивать цепь из различных обработчиков динамически.

- **Проблема**

Для начала определимся, что значит посылать запрос? Не пугайтесь, это синоним обычного вызова метода. Мы говорим «посылать запрос», т.к. клиент не уверен какой именно объект его обработает и обработает ли вообще.

Проблема в том, что объект, инициирующий запрос (например, кнопка), может не располагать информацией о том, какой объект этот запрос обработает. Кнопка не знает, находится ли она на панели, или в окне редакторе или где-то ещё; а значит, она не знает кому именно «сообщить» о своём нажали.

Кроме того, лучше не привязывать классы кнопок к классам бизнес-логики программы, если вы хотите повторно использовать эти кнопки где-нибудь ещё.

- **Решение**

Идея цепочки обязанностей заключается в том, чтобы разорвать связь между отправителями и получателями запросов, дав возможность обработать запрос нескольким объектам. Запрос перемещается по цепочке объектов, пока один из них не обработает его.

Как связаны объекты в цепочке? Каждый из объектов имеет ссылку на следующий объект цепочки (например, кнопка имеет ссылку на панель, панель на диалог, диалог на окно). Все объекты цепочки должны реализовать общий интерфейс, чтобы связка была гибкая и в цепочку можно было подставлять разнообразные классы.

Что происходит, когда клиент обращается к одному из объектов цепочки? Этот объект проверяет, может ли он выполнить операцию. Если да, он её тут же выполняет и возвращает результат клиенту. Если нет, он переадресует операцию следующему объекту в цепочке и так далее либо пока запрос не будет выполнен, либо пока мы не достигнем конца цепочки.

Аналогия из жизни

Связь с отделом поддержки

Представьте, что вы купили новую видеокарту к своему рабочему компьютеру. Она автоматически определилась и заработала под Windows, но в вашей любимой Ubuntu «завести» её не удалось. В надежде, вы звоните в службу поддержки.

Первым вы слышите стандартный автоответчик, предлагающий выбор из десятка стандартных проблем и решений. Ни один из вариантов не подходит и робот соединяет вас с живым оператором.

К сожалению, рядовой оператор поддержки умеет общаться только заученными фразами и давать шаблонные ответы. После очередного предложения «выключить и включить компьютер», вы просите связать вас с его супервайзером.

Оператор перебрасывает звонок супервайзеру, который знает чуточку больше о своей работе. Он понимает в чём заключается ваша проблема, но не имеет быстрого решения.

Супервайзер переадресует ваш звонок дежурному инженеру, изнывающему от скуки в своей коморке. Уж он-то знает как вам помочь и рассказывает где и как можно скачать подходящие драйвера, и как настроить их под Ubuntu.

Запрос удовлетворён. Вы кладёте трубку.

- **Структура**

- **Применимость**
- ❖ **Если программа содержит более одного объекта, способного обработать запрос, но не знает заранее кто из них его действительно обработает.** Вы связываете потенциальных обработчиков в одну цепь и поочерёдно спрашиваете, хочет ли данный объект обработать запрос. Если нет, двигаетесь дальше по цепочке.
- ❖ **Если набор объектов, способных обработать запрос, должен задаваться динамически.** В любой момент вы можете вмешаться в существующую цепочку и переназначить связи так, чтобы убрать или добавить новое звено.
- **Преимущества и недостатки**

“+”	“-”
● Уменьшает зависимость между клиентом и обработчиками.	● Запрос может остаться никем не обработанным.

• Соблюдает *принцип единственной обязанности класса*. • Соблюдает *принцип открытости/закрытости*	

2)Команда

- **Суть паттерна:** Превращает операцию в объект. Это позволяет передавать операции как аргументы при вызове методов, ставить операции в очередь, логировать их, а также поддерживать отмену операций.

- **Проблема**

Если вам нужно выполнить какую-то операцию, вы берёте определённый объект и вызываете нужный метод. Но что, если вам нужно выполнить операцию не сейчас, а, скажем, через 10 минут?

Вам придётся либо ставить на паузу всю программу, либо запустить таймер, запомнив, что через 10 минут, вы должны выполнить такое-то действие, и в таком контексте. Первый вариант не понравится вашим пользователям, а второй не очень удобен в реализации, если отложенных операций будет много.

- **Решение**

Вот тут и приходит на помощь паттерн Команда. Он предлагает превратить операции в объекты. Параметры операции станут полями этого объекта.

Например, мы вынесем операцию «вырезать» из класса Документ в класс КомандаВырезать. Чтобы объект команды понимал что же ему вырезать, мы будем подавать ему в конструктор ссылку на текущий документ.

Теперь, благодаря тому, что операции являются объектами, вы можете делать с ними всякие интересные вещи. Например, в три этапа вы могли бы реализовать систему отмены для вашей программы:

1. Добавить в классы команд метод отменить(), который бы делал то же, что и основной метод команды, но наоборот.
2. Сохранять все выполненные объекты команд в один список.
3. При нажатии кнопки отмены, брать последнюю команду из списка и выполнять её метод отменить().

Важно, чтобы все классы команд имели общий интерфейс. Тогда клиенты и обработчики команд будут работать с любыми командами, без привязки к конкретным классам.

Аналогия из жизни

Заказ в ресторане

Вы заходите в ресторан и садитесь у окна. К вам подходит вежливый официант и принимает заказ, записывая все пожелания в блокнот. Откланявшись, он уходит на кухню, где вырывает

лист из блокнота и клеит на стену. Сорвав лист со стены, шеф читает содержимое заказа и готовит блюдо, которое вы заказали. В этом примере, вы являетесь *Отправителем*, официант с блокнотом — *Командой*, а шеф — *Получателем*. Как и в паттерне, вы не соприкасаетесь напрямую с шефом. Вместо этого, вы отправляете заказ с официантом, который самостоятельно «настраивает» шефа на работу.

- **Структура**

- **Применимость**
- ❖ **Вы хотите параметризовать объекты выполняемым действием.**

К примеру, вы разрабатываете библиотеку графического меню. Вы хотите, чтобы пользователи вашей библиотеки могли создавать объекты меню и передавать в них желаемые операции по клику на элементы меню. И да, это могут быть не только элементы меню, а и кнопки, поля для ввода, горячие клавиши и прочее.

- ❖ **Вы хотите ставить операции в очередь или выполнять их по расписанию.**

Команда позволяет превратить операцию в объект. А значит, объекты можно хранить в полях других объектов. Кроме того, объекты можно сериализировать, то есть превращать в строку, чтобы потом сохранить в файл или базу данных. А затем достать в любой удобный момент и выполнить.

- ❖ **Если нужна операция отмены.**

Отмену почти всегда реализуют при помощи паттерна Команда. Для отмены вам нужно сохранять список выполненных операций. Как вариант, можно хранить историю в каком-то урезанном суррогате команды (например, в строке, по которой потом находить нужный контекст).

Но в конечном итоге иметь объект для каждой команды окажется удобнее из-за всех фич ООП, который идут в комплекте.

- **Преимущества и недостатки**

“+”	“-”
• Убирает зависимость между разными слоями программы. • Простая отмена и повтор команд. • Отложенный запуск команд. • Комбинирование команд. • Соблюдает *принцип открытости/закрытости*.	• Усложняет код программы за счёт дополнительных классов.

3)Итератор

- **Суть паттерна:** Даёт возможность последовательно обходить все элементы составного объекта, не раскрывая его внутреннего представления.

- **Проблема**

Коллекции — самая частая структура данных, которую вы можете встретить в программировании. Это набор объектов, собранный в одну кучу по каким-то причинам.

Коллекция может быть списком, массивом, деревом и так далее. Но как бы она ни была структурирована, пользователь должен иметь возможность последовательно обходить элементы коллекции чтобы проделывать с ними какие-то действия.

Способ обхода тоже может быть разным. Сегодня вам достаточно простого обхода элементов по порядку. Но завтра может понадобиться обход по отсортированному списку элементов, а послезавтра — фильтрация или случайный обход элементов

- **Решение**

Идея паттерна Итератор в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс.

Объект итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом. А если понадобится добавить новый способ обхода, вы создадите новый класс итератора, не изменяя код коллекции.

Аналогия из жизни

Туристический гид

Вы планируете полететь в Рим и обойти все достопримечательности за пару дней. Приехав, вы можете долго петлять узкими улочками, пытаясь найти Колизей. А можете плюнуть и нанять локального гида, который хоть и обойдётся в копеечку, но знает город как свои пять пальцев и

сможет посвятить вас во все городские легенды. Таким образом, Рим выступает коллекцией достопримечательностей; Гид, навигатор или путеводитель — вашим итератором по коллекции. Вы, как клиентский код, можете выбрать один из них, опираясь на решаемую задачу и доступные ресурсы.

- **Структура**

- **Применимость**
- ❖ **У вас есть сложная структура данных и вы хотите скрыть от пользователям детали её реализации (из-за сложности или вопросов безопасности).** Создайте итератор для простого доступа к своей коллекции. Он скроет детали структуры данных от клиента, а также защитит данные от неосторожных или злоумышленных действий.
- ❖ **Вам нужно иметь несколько вариантов обхода одной и той же структуры данных.** Нетривиальные способы обхода структуры данных могут захламлять бизнес-логику класса. В таком случае код обхода можно вынести в отдельные классы итераторов. Вы будете передавать определённый итератор в параметры операций бизнес-логики.
- ❖ **Вам хочется иметь единый интерфейс обхода различных структур данных.** Итератор позволяет вынести в подклассы реализации различных вариантов обхода. Это позволит легко взаимозаменять объекты итераторов, в зависимости от того, с какой структурой данных приходится работать.
- **Преимущества и недостатки**

“+”	“-”
• Упрощает классы хранения данных. • Позволяет реализовать различные способы обхода структуры данных. • Позволяет одновременно перемещаться по структуре данных в разные стороны. •	• Неоправдан, если можно обойтись простым циклом.

4)Посредник

- **Суть паттерна:** Уменьшает связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

- **Проблема**

Вы хотите повторно использовать некоторые компоненты готовой программы. Однако они имеют слишком тесные связи с другими компонентами, которые вам не нужны. Поэтому вы можете использовать либо все компоненты сразу, либо никаких.

- **Решение**

Посредник упрощает сложные связи и зависимости между классами. А чем меньше связей имеет класс, тем проще его изменять и расширять.

Паттерн заставляет классы общаться не напрямую друг с другом, а через отдельный класс-посредник, который знает кому нужно перенаправить тот или иной запрос. Благодаря этому, компоненты системы будут зависеть только от класса-посредника, а не от десятков других компонентов.

Аналогия из жизни

Диспетчерская башня в аэропорту

Пилоты садящихся или улетающих самолётов не общаются напрямую с другими пилотами. Вместо этого, они связываются с диспетчером, который координирует действия нескольких самолётов одновременно. Без диспетчера, пилотам приходилось бы все время быть начеку и следить за всеми окружающими самолётами самостоятельно. А это приводило бы к частым катастрофам в небе.

- **Структура**

- **Применимость**

❖ **Вам сложно менять некоторые классы/объекты из-за множества хаотичных связей с другими классами/объектами.** Посредник позволяет засунуть все эти связи в один класс. После чего вам будет легче их отрефакторить, сделать более понятными и гибкими.

❖ **Вы не можете повторно использовать класс, поскольку он зависит от уймы других классов.** После применения паттерна, компоненты теряют прежние связи с другими компонентами. А всё их общение происходит косвенно, через посредника.

❖ **Поведение, распределённое между несколькими классами, должно настраиваться без порождения подклассов для каждого компонента.** Раньше изменение отношений в одном компоненте могли повлечь за собой снежный ком изменений в каждом другом компоненте. Теперь же, вам достаточно создать подкласс посредника и изменить в нём связи между компонентами.

- **Преимущества и недостатки**

“+”	“-”
• Устраняет зависимости между компонентами. • Упрощает взаимодействие между компонентами. • Централизует управление в одном месте.	• Посредник может сильно раздуться.

5)Снимок

- **Суть паттерна:** Позволяет делать снимок состояния объекта, не раскрывая подробностей его реализации. Позже можно будет восстановить прошлое состояние объекта, используя этот снимок.

- **Проблема**

Нужно восстанавливать объекты к их прежнему состоянию (т.е. иметь операцию отмены). Причём отмену хочется реализовать так, чтобы не раскрывать внутреннего представления объектов системы.

Другими словами, вам не хочется делать все поля классов публичными ради того, чтобы получить возможность копировать их данные.

- **Решение**

Если снятие снимка «извне» объекта нарушит инкапсуляцию, то почему бы не снять его «изнутри»?

Объект, владеющий состоянием, должен иметь два метода:

- сделатьСнимок(): будет создавать специальный объект-снимок и заполнять его текущими значениями своих публичных и приватных полей.
- восстановитьСостояние(x: Снимок): должен скопировать значения полей из поданного объекта-снимка.

Теперь, перед изменением состояния объекта, Клиент сможет сначала получить его снимок и сохранить его в истории. Затем, если нужно вернуть прежнее состояние, Клиент передаст снимок обратно в объект.

Как видите, чтобы сделать операции отмены и повтора в вашем приложении, достаточно иметь стек из Команд, и стек из Снимков.

- **Структура**

- **Применимость**
- ❖ **Вам нужно сохранять мгновенный снимок состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии.** Паттерн Снимок позволяет делать любое количество снимков объекта и хранить их независимо от объекта, с которого делают снимок. Снимки часто используют не только для реализации операции отмены, но и для транзакций (когда состояние объекта нужно откатить, если операция не удалась).
- ❖ **Прямое получение состояния объекта раскрывает детали его реализации, нарушая инкапсуляцию.** Паттерн предлагает изготовить снимок самому исходному объекту, так как ему доступны все поля, даже приватные.
- **Преимущества и недостатки**

"+"	"-"
● Не нарушает инкапсуляции исходного объекта. ● Упрощает структуру исходного объекта. Ему не нужно хранить историю версий своего состояния.	● Требует много памяти, если клиенты слишком часто создают снимки. ● Может повлечь дополнительные издержки памяти, если объекты хранящие историю не освобождают ресурсы, занятые устаревшими снимками. ● В некоторых языках (например Python, JavaScript) сложно гарантировать, чтобы только исходный объект имел доступ к состоянию снимка.

6)Наблюдатель

- **Суть паттерна:** Создаёт механизм подписки, с помощью которого одни объекты могут подписываться на обновления, происходящие в других объектах.

- **Проблема**

Представим, что у вас есть два объекта — Покупатель и Магазин. В магазин вот-вот должны завезти новый товар, который интересен покупателю.

Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет тратить драгоценное время и злиться.

С другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический и не всем он нужен.

Получается конфликт — либо один объект работает неэффективно, тратя ресурсы на периодические проверки, либо второй объект оповещает слишком широкий круг пользователей, тоже тратя ресурсы впустую.

Есть ещё один минус — в обоих случаях вы жёстко свяжете друг с другом конкретные классы Покупателя и Магазина.

- **Решение**

Паттерн Наблюдатель предлагает чётко разделить роли между объектами и выделить их в интерфейсы Издателя (*предмета* или *отправителя*) и Подписчика (*наблюдателя*, *слушателя* или *получателя*).

Объекты, имеющие интересное для других объектов состояние станут *Издателями*. Они будут хранить списки других объектов — *Наблюдателей* — которым важно ослеживать изменения состояния в Издателе.

Подписчики могут оформлять и закрывать подписку динамически, прямо во время выполнения программы.

При каждом изменении состояния издателя, он будет рассылать уведомления своим подписчикам.

В программе может быть сразу несколько типов издателей и подписчиков. Все они смогут работать друг с другом, используя общие интерфейсы издателей и подписчиков.

Аналогия из жизни

Подписка на газеты

После того как вы оформили подписку на газету или журнал, вам больше не нужно ездить в супермаркет и проверять не вышел ли очередной номер. Вместо этого, издательство будет присылать новые номера сразу после выхода прямо к вам домой.Издательство ведёт список

подписчиков и знает кому какой журнал слать. Вы можете в любой момент отказаться от подписки и журнал перестанет к вам приходить.

- **Структура**

- **Применимость**

❖ **В При изменении состояния одного объекта требуется изменить другие. Но вы не знаете наперёд, сколько и какие объекты нужно изменять.** Например, при разработке GUI фреймворка вам нужно дать возможность сторонним классам реагировать на клики по кнопкам.

❖ **Паттерн Наблюдатель даёт возможность любому объекту с интерфейсом подписчика, подписываться на изменения в объектах-издателях. Одни объекты должны наблюдать за другими, но только в определённых случаях.** Объекты издатели ведут динамические списки. Все наблюдатели могут подписываться или отписываться на обновления прямо во время выполнения программы.

- **Преимущества и недостатки**

“+”	“-”
• Издатель не зависит от конкретных классов подписчиков. • Вы можете подписывать и отписывать получателей на лету. • Реализует *принцип открытости/закрытости*.	• Наблюдатели оповещаются в случайном порядке. • Код подписки издателя можно только унаследовать, поэтому его сложно интегрировать в существующее дерево классов.

7)Состояние

- **Суть паттерна:** Позволяет объекту менять своё поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

- **Проблема**

Давайте разберёмся с терминами.

Что такое *состояние*?

Это значения всех полей объекта.

Например, самолёт находится в условном состоянии полет, если его высота > 0, скорость > 0 и включен_двигатель == true. Тот же самолёт находится в состоянии погрузка, если мы в аэропорту, высота == 0, включен_двигатель == false.

Что значит *менять поведение в зависимости от состояния*?

Например, у самолёта есть функция выгрузить пассажиров. Если её выполнить в состоянии разгрузка, произойдёт одно действие, а если выполнить её во время полёта, то произойдёт... что-то совершенно другое.

Что такое *стейт-машина*?

Стейт-машина — это программа, которая построена вокруг нескольких определённых состояний. Работа этой программы происходит путём постоянных переходов из одного состояния в другое.

Например, наш самолёт можно запрограммировать как стейт-машину с такими состояниями:

При этом из одного состояния может быть несколько переходов в другие состояния, например:

Паттерн Состояние является одним из способов реализации стейт-машины (хотя и не единственным).

Итак, у вас есть объект, который может находиться в некоторых состояниях. Часть его поведения в этих состояниях должна меняться.

- **Решение**

Паттерн предлагает вынести код, зависящих от определённых состояний объекта в отдельные классы-состояния с общим интерфейсом.

В основном классе будет содержаться ссылка на объект, соответствующий текущему состоянию. Вместо того чтобы самому содержать и выполнять контекстно-зависимое поведение, основной класс будет делегировать исполнение объекту-состоянию.

Так как все классы-состояния реализуют общий интерфейс, вы можете подставлять новый объект-состояние в основной класс при смене его состояния. Этим и будет достигаться смена поведения в зависимости от состояния.

Аналогия из жизни

Смартфон

Ваш смартфон ведёт себя по-разному, в зависимости от текущего состояния:

- Когда телефон разблокирован, нажатие кнопок телефона приводит к каким-то действиям.
- Когда телефон заблокирован, нажатие кнопок приводит к экрану разблокировки.
- Когда телефон разряжен, нажатие кнопок приводит к экрану зарядки.

В этом примере телефон является Контекстом, а кнопки — его методами. Нажатие кнопок приводит к вызову методов текущего внутреннего Состояния.

- **Структура**

- **Применимость**

❖ **У вас есть класс, поведение которого кардинально меняется в зависимости от внутреннего состояния. Причём типов состояний много и их код часто изменяется.** Паттерн предлагает создать класс для каждого такого состояния, а затем переместить туда все поля и методы, связанные с состоянием. Контекстный класс будет содержать

экземпляр класса-состояния и делегировать ему работу. При изменении состояния, в контекст будет подставляться другой объект.

- ❖ **В разных методах класса вы видите много больших, похожих друг на друга, условных операторов, которые проверяют состояние объекта и выполняют какие-то действия. Такие условные операторы могут проверять одни и те же данные в каждом месте, но выполнять разную работу.** Паттерн предлагает поместить каждую ветку такого условного оператора в собственный класс. В этот класс можно засунуть и все поля, связанные с данным состоянием.
- **Преимущества и недостатки**

“+”	“-”
● Избавляет от множества больших условных операторов (стейт-машины). ● Концентрирует в одном месте код, связанный с определённым состоянием. ● Упрощает код контекста.	● Может неоправданно усложнить код, если состояний мало и они редко меняются.

8)Стратегия

- **Суть паттерна:** Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс. После чего, алгоритмы можно взаимозаменять прямо во время исполнения программы.

- **Проблема**

Предположим, у вас в каком-нибудь классе (назовём его *контекст*) имеется несколько схожих алгоритмов, то есть способов сделать то или иное действие. Эти алгоритмы приводят к одному и тому же результату, но могут отличаться временем выполнения, потреблением ресурсов, требуемыми условиями и прочим. Например, различные алгоритмы сортировок массивов. Контекст выбирает подходящий алгоритм в зависимости от текущих условий.

Первая проблема заключается в том, что вам приходится держать код алгоритмов в классе контекста, загромождая его основную бизнес-логику.

Кроме того, если эти алгоритмы нужно будет использовать где-то ещё, то придётся дублировать код, так как их нельзя использовать в отрыве от класса-контекста.

- **Решение**

Вот тут и приходит на помощь паттерн Команда. Он предлагает превратить Паттерн Стратегия предлагает вынести все эти алгоритмы из контекста в собственные классы, называемые стратегиями. Объект контекст будет содержать ссылку на один-единственный объект-стратегию, которому и будет делегироваться выполнение.

Так как все объекты-стратегии будут иметь одинаковый интерфейс, вы сможете при желании взаимозаменять их в контексте на лету.

- **Структура**

- **Применимость**

❖ **В Вам нужно использовать разные вариации какого-то алгоритма внутри одного класса (например, отличающиеся балансом скорости и потребления ресурсов).** Стратегия позволяет изменять поведение объекта во время выполнения программы.

❖ **У вас есть множество похожих классов, отличающихся только некоторым поведением.** Стратегия позволяет объединить эти классы в один, сделав поведение настраиваемым**.**

❖ **Алгоритм использует данные, которые нежелательно показывать клиентам этого класса.** Стратегия позволяет скрыть данные и детали реализации каждого алгоритма внутри отдельных классов.

❖ **Различные вариации алгоритмов реализованы в виде развесистого условного оператора. Каждая ветка такого оператора представляет вариацию алгоритма.** Стратегия помещает каждую лапу такого оператора в отдельный класс-стратегию. Затем контекст получает определённый объект-стратегию от клиента и делегирует ему работу. Если вдруг понадобится сменить алгоритм, в контекст можно подать другую стратегию.

- **Преимущества и недостатки**

“+”	“-”
● Горячая замена алгоритмов на лету. ● Уход от наследования к делегированию. ● Реализует *принцип открытости/закрытости.* ● Скрывает опасные/лишние данные алгоритма от Клиента.	● Усложняет программу за счёт дополнительных классов. ● Клиент должен знать в чём разница между стратегиями, чтобы выбрать подходящую.

9)Шаблонный метод

- **Суть паттерна:** Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.
- **Проблема**

Вы хотите, чтобы пользователи вашего класса имели возможность переопределять части основного алгоритма класса, не меняя структуру этого алгоритма.

- **Решение**
  Паттерн предлагает разбить алгоритм на отдельные шаги, вынести их в отдельные методы и вызывать их в одном «шаблонном» методе друг за другом. Это позволит подклассам переопределять шаги алгоритма, оставляя его структуру без изменений.

- **Структура**

- **Применимость**
- ❖ **Если подклассы должны расширять базовый алгоритм, не меняя его структуры.** Шаблонный метод позволяет указать конкретные точки (хуки), в которых базовую функциональность можно расширить через наследование, не меняя при этом структуру алгоритмов.
- ❖ **У вас есть несколько классов, делающих одно и то же с незначительными отличиями. Если вы редактируете один класс, то приходится вносить такие же правки и в остальные классы.** Паттерн шаблонный метод предлагает создать для похожих классов общий суперкласс и оформить в нём главный алгоритма в виде шагов. Отличающиеся шаги можно переопределить в подклассах. Это позволит убрать дублирование кода в нескольких классах с похожим поведением, но отличающихся в деталях.
- **Преимущества и недостатки**

“+”	“-”
● Облегчает повторное использование кода..	● Вы ограничены скелетом существующего алгоритма.

10)Посетитель

- **Суть паттерна:** Позволяет добавить новое поведение ко всем объектам некой связанной структуры, не изменяя самих объектов, к которым это поведение относится.

- **Проблема**

Посетитель имеет смысл только тогда, когда мы работаем с какой-то системой объектов – связанным списком, цепочкой, деревом и так далее. Если у вас нет такой структуры, то применять паттерн незачем.

Зачем кому-то нужно добавлять новое поведение объектам, не меняя их код? Почему нельзя просто взять и добавить какое-то поведение в нужные классы? Вот некоторые причины:

- Когда новое поведение идёт вразрез с тем что класс уже делает. Подумайте о *принципе единственной обязанности*, прежде чем вписать новый код в класс. Нарушив принцип пару раз, вы превратите свои классы в безобразную свалку.
- Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии. Добавив поведение в два класса из десяти, вы усложните код проверками типа. Подумайте нельзя ли вместо этого использовать полиморфизм.
- Классы могут находиться в сторонней библиотеке, к которой у вас нет доступа. Поэтому просто дописать новый метод в класс не получится. Бывает обратная ситуация — когда вы пишете стороннюю библиотеку, и у будущего пользователя не будет возможности дописать что-то в ваш код.
- **Решение**

Посетитель обходит связанную структуру один объект за другим, и, в зависимости от типа посещённого объекта, выполняет над ним какое-то действие.

Аналогия из жизни

Страховой агент

Представьте начинающего страхового агента, жаждущего получить новых клиентов. Он беспорядочно ходит от дома к дому, предлагая свои услуги. Для каждого из «типов» домов, которые он посещает, у него имеется особое предложение.

- Придя в дом к обычной семье, он предлагает оформить медицинскую страховку.
- Придя в банк, он предлагает страховку от грабежа.
- Придя на фабрику, он предлагает страховку предприятия от пожара и наводнения.

- **Структура**

- **Применимость**

❖ **Вам нужно выполнить операцию над всеми элементами сложной структуры данных (например, деревом объектов), причём все элементы разнородны и не имеют общего интерфейса.** Посетитель позволяет добавить одну и ту же операцию в объекты различных типов.

❖ **Над объектами сложной структуры данных, надо выполнять несколько не связанных между собой операций и вы не хотите «засорять» классы такими операциями.** Посетитель позволяет объединить родственные операции, поместив их в один класс. Если структура объектов является общей для нескольких приложений, то паттерн посетитель позволит в каждое приложение включить только относящиеся к нему операции.

- **Преимущества и недостатки**

"+"	"-"
• Упрощает добавление новых операций над всей связанной структурой объектов. • Объединяет родственные операции в одном классе. • Посетитель может накоплять состояние при обходе структуры.	• Паттерн неоправдан, если иерархия компонентов часто меняется. – Нарушает инкапсуляцию компонентов.