Практические вопросы

1. Структура программы на языках C и C++. Функции C и C++. Перегрузка функций в C++. Параметры функций по умолчанию.

Программа состоит из набора файлов, который включает в себя объявление определений.

Существуют абстракции по:

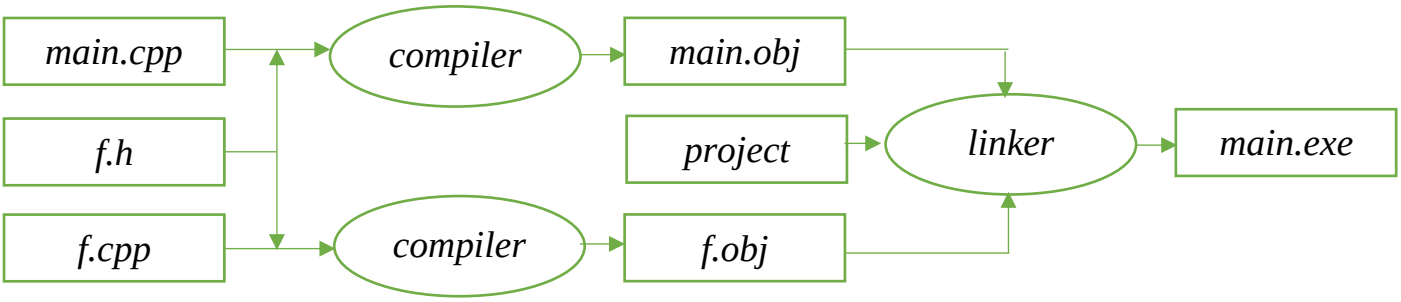
- Действию:
  - Функции - можем объявлять и определять
- Данным:
  - Типы данных - можем объявлять и определять
  - Переменные - можем объявлять и определять
  - Константы

Выполнение программы как правило начинается с ф-ции *"main"*, она обязательно должна присутствовать. Существуют приоритеты выполнения ф-ций.

Функции могут быть вложенными. Любая ф-ция может вызвать любую другую ф-цию, в том числе и саму себя (кроме main).

Для использования файлов реализации, все что мы хотим использовать необходимо сначала определить, либо объявить в том месте, где хотим использовать. Для удобства создаются заголовочные файлы, в кот. выносятся определения.

Процесс сборки:



В заголовочном файле мы можем определять константы времени компиляции, типы данных. Не можем переменные.

Значение из ф-ции может быть возвращено непосредственно, либо через аргумент.

Перегрузка функций – возможность создавать функции с одними и теми же названиями. Можно определить функцию, которую необходимо вызвать по кол-ву параметров, или по типу данных аргументов.

```cpp
void func1(int& x) { cout << "func1(int&)" << endl; }
void func1(const int& x) { cout << "func1(const int&)" << endl; }

void func2(int x) { cout << "func2(int)" << endl; }
void func2(int& x) { cout << "func2(int&)" << endl; }

void func3(const int& x) { cout << "func3(const int&)" << endl; }
void func3(int&& x) { cout << "func3(int&&)" << endl; }

void func4(int& x) { cout << "func4(int&)" << endl; }
void func4(int&& x) { cout << "func4(int&&)" << endl; }

void func5(int x) { cout << "func5(int)" << endl; }
void func5(int&& x) { cout << "func5(int&&)" << endl; }

void func6(int x) {}
void func6(const int& x) {}


    int i = 0;          const int ci = 0;       int& lv = i;
    const int& clv = ci;        int&& rv = i + 1;

    func1(i);           // int&
    func1(ci);          // const int&
    func1(lv);          // int&
    func1(clv);         // const int&
    func1(rv);          // int&
    func1(i + 1);       // const int&
    cout << endl;

    // func2(i);        // Error!
    func2(ci);          // int
    // func2(lv);       // Error!
    func2(clv);         // int
    // func2(rv);       // Error!
    func2(i + 1);       // int
    cout << endl;

    func3(i);           // const int&
    func3(ci);          // const int&
    func3(lv);          // const int&
    func3(clv);         // const int&
    func3(rv);          // const int&
    func3(i + 1);       // int&&
    cout << endl;

    func4(i);           // int&
    // func4(ci);       // Error!
    func4(lv);          // int&
    // func4(clv);      // Error!
    func4(rv);          // int&
    func4(i + 1);       // int&&
    cout << endl;

    func5(i);           // int
    func5(ci);          // int
    func5(lv);          // int
    func5(clv);         // int
    func5(rv);          // int
    // func5(i + 1);    // Error!

    // func6(i);        // Error!
    // func6(ci);       // Error!
    // func6(lv);       // Error!
    // func6(clv);      // Error!
    // func6(rv);       // Error!
    // func6(i + 1);    // Error!
```

В C++ один или несколько аргументов функции могут задаваться по умолчанию. Для каждого параметра значение по умолчанию можно указать не более одного раза, но каждое последующее объявление функции, а также определение функции может назначать параметрам значения по умолчанию. Параметры по умолчанию должны быть последними в списке параметров.

```
// Пример: функция сортировки по возрастанию массива вещественных чисел
void sort(double *arr, int count, int key = 0);

// key показывает направление сортировки (0 - по возрастанию, 1 - по убыванию).
// По умолчанию сортировка будет по возрастанию
// (key = 0, если в функцию передаются только два параметра - указатель на массив
// и количество элементов в массиве).
```

2. Ссылки. lvalue и rvalue ссылки. Передача параметров в функции по ссылке. Автоматическое выведение типа.

Ссылка – тот же указатель, но всегда инициализированный (не может указывать на null). Но указатель – тип данных, а ссылка нет.

```
int i;
int &ai = i;
ai = 2; // i = 2;
// int &  - левая ссылка
// int && - правая ссылка

int i = 0;
const int c = 0;

int &lv1 = i;
const int &clv1 = c;
const int& clv2 = i + 1; // создается новая ячейка памяти

int &&rv1 = i;           // Error
int &&rv2 = i + 1;
++rv2;                   // можем изменять содержимое данной области

int &&rv3 = rv2;         // Error
int &lv2 = rv2;
int &&rv4 = i + 0;       // ссылка на копию i
int &&rv5 = (int) i;     // ссылка на копию i

// получить правую ссылку из любого выражения:

int &&rv6 = std::move(i);
```

Ссылки используются для передачи параметра в функцию. На низком уровне ссылки хоть ничем не отличается от указателя, но идет контроль.

```
void swap(double &d1, double &d2)
{
    double temp = d1;
   d1 = d2;
   d2 = temp;
}

swap(arr[i], arr[j]);
```

Начиная с C++11 ключевое слово *auto* при инициализации переменной может использоваться вместо типа переменной, чтобы сообщить компилятору, что он должен присвоить тип переменной исходя из инициализируемого значения. Это называется выводом типа (или «автоматическим определением типа данных компилятором»). Например:

```
auto x = 4.0; // 4.0 - это литерал типа double, поэтому и x должен быть типа double
auto y = 3 + 4; // выражение 3 + 4 обрабатывается как целочисленное, поэтому и
переменная y должна быть типа int
```

Это работает даже с возвращаемыми значениями функций:

```
int subtract(int a, int b) {
    return a - b;
}

int main()
{
    auto result = subtract(4, 3); // функция subtract() возвращает значение типа int и,
следовательно, переменная result также должна быть типа int
    return 0;
}
```

Переменные, объявленные без инициализации, не могут использовать эту особенность (поскольку нет инициализируемого значения, и компилятор не может знать, какой тип данных присвоить переменной).

В C++14 функционал ключевого слова *auto* был расширен до автоматического определения типа возвращаемого значения функции. Например:

```
auto subtract(int a, int b)
{
    return a - b;
}
```

Так как выражение $a - b$ является типа *int*, то компилятор делает вывод, что и функция должна быть типа *int*.

3. Классы и объекты в C++. Определение класса с помощью class, struct, union. Ограничение доступа к членам класса в C++. Члены класса и объекта. Методы класса и объекта. Константные члены класса. Схемы наследования.

Уровни доступа к членам класса:

- private - нет доступа извне
- protected - к ним имеют доступ потомки класса
- public - методы / функции

Эти методы контролируют целостность объекта.

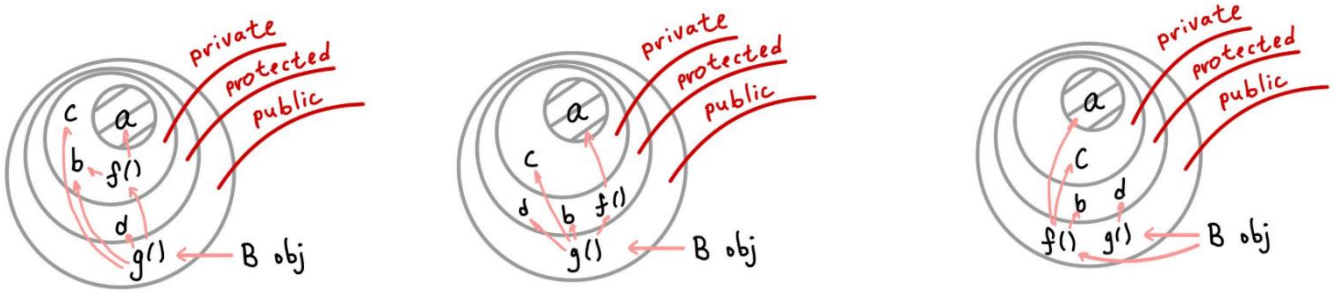
У *struct* по умолчанию уровень доступа для членов *public*, а у *class* – *private*. У *union* – *public*. В *union* нет уровня protected, так как *union* не может быть ни базовым классом, ни производным.

```
class <имя> [: <список баз>]
{
private:
    <члены>
protected:
    <члены>
public:
    <члены>
};
```

Схемы наследования.

```
class A
{
private:
   int a;
protected:
    int b;
public:
    int f();
};

class B: private A      // public / protected
{
private:
    int c;
protected:
    int d;
public:
    int g();            // все члены базового класса получат уровень доступа private
};
```

private *protected* *public*



Для того, чтобы восстановить уровень доступа можно написать:

```
public:
    using A::f;
```

Члены класса не относятся к конкретному объекту, они общие для всех объектов данного класса. Перед ними ставится ключевое слово *static*. Память под них не относится к объекту (при вызове *sizeof* они не будут учитываться).

Метод можно определить вне класса.

```
class A {
private:
   int a;
   const int cb;
   static int sc;
   static const int scd = 1;

public:
   int f();
};

int A::f() { return this->a; }
```

Методы могут быть константными, они не могут менять члены класса. Чтобы они могли менять член объекта, перед его определением необходимо написать ключевое слово *mutable*.

С членами класса можно работать, не создав ни одного объекта данного класса. Для этого необходимо создать методы класса.

```
public:
    static int g();
```

Статические методы не принимают указатель *this*. Для доступа к методам могут использоваться операторы: «.», «->», «::», «.*», «->*».

4. Создание и уничтожение объектов в C++. Конструкторы и деструкторы. Раздел инициализации конструкторов. Способы создания объектов. Явный и неявный вызов конструкторов. Приведение типа.

После создания объекта сразу должна пройти его инициализация, для этого мы используем специальные методы – конструкторы. Его вызов может быть как явным, так и неявным. По умолчанию создается конструктор без параметров, конструктор копирования, и конструктор переноса. Конструктор – метод, который не имеет типа возврата, его имя совпадает с именем класса:

```
<имя класса>::<имя конструктора>([параметры])[:<раздел инициализации>]
{
}
```

Пример:

```
class A
{
private:
   int a;
   const int cb = 2;
   static int sc;
   static const int scd = 1;

public:
    A(int i);
};
```

```
A::A(int i): a(i) //, cb(i), sc(i), scd(i)
{
    a = i;
    // cb = i;      // Error
    // sc = i;      // Не относится к классу в общем
    // scd = i;     // Error
}
```

Порядок, который мы написали в разделе инициализации не влияет на порядок создания объектов.

Для уничтожения объектов используются деструкторы. Он порождается по умолчанию. Как и оператор присваивания. Для создания/удаления объектов используются операторы языка *new*, *delete*. Они возвращают типизированный указатель (в отличие от *void* в C). Рассмотрим следующий класс:

```
class Array {
private:
    double* arr;
    int count;

public:
    Array() = default;
    Array(int cnt) : count(cnt) { arr = new double[count] {}; }
    Array(const Array& ar);
    Array(Array&& ar) noexcept;
    ~Array();

    bool equals(Array ar);

    static Array minus(const Array& ar);
};

Array::Array(const Array& ar) : count(ar.count) {
    arr = new double[count];

    for (int i = 0; i < count; ++i)
        arr[i] = ar.arr[i];
}

Array::Array(Array&& ar) noexcept : count(ar.count) {
    arr = ar.arr;
    ar.arr = nullptr;
}

Array::~Array() { delete[] arr; }

bool Array::equals(Array ar) {
    if (count != ar.count)
        return false;

    int i;
    for (i = 0; i < count && arr[i] == ar.arr[i]; ++i);

    return i == count;
}

Array Array::minus(const Array& ar)
{
    Array temp(ar);

    for (int i = 0; i < temp.count; ++i)
        temp.arr[i] *= -1;

    return temp;
}
```

Если перед именем конструктора поставить слово *explicit*, это будет гарантировать, что конструктор неявно не вызовется. Все конструкторы с 1 параметром должны быть с этим модификатором, это позволяет избежать случайного приведения типов.

Явно можно вызвать конструктор следующими способами:

```
<тип> <идентификатор> [(<параметры>)];            // A obj();
<тип> <идентификатор> {[параметры]};
<тип> *<идентификатор> = new <тип>{(параметры)};
<тип> <идентификатор> = <тип>{(параметры)};
```

Неявно:

```
<тип> <идентификатор> = <значение>;
<тип> <идентификатор> = {(параметры)};
```

Любой конструктор - оператор приведения типа.

5. Конструкторы копирования и переноса. Модификатор explicit. Удаление конструктора и default конструктор. Делегирующие и унаследованные конструкторы.

Обычный конструктор используется для той или иной инициализации объекта, он не должен вызываться для копирования уже существующего объекта, так как такой вызов изменит содержимое объекта (передаст объект в начальном состоянии) а мы хотим передать функции текущее состояние объекта, то есть нужен конструктор копирования.

Когда мы передаем в функцию какой-то объект по ссылке, конструктор не вызывается. Когда мы возвращаем объект из какой-либо функции (не по ссылке), если этот объект создается внутри этой функции, тоже вызывается конструктор копирования.

Конструктор копирования должен принимать в качестве параметра объект того же класса. Причем параметр лучше принимать по ссылке, потому что при передаче по значению компилятор будет создавать копию объекта. А для создания копия объекта будет вызываться конструктор копирования, что приведет бесконечной рекурсии.

```
class Person
{
    std::string name;
    unsigned age;
public:
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
    }
    Person(const Person &p)
    {
        name = p.name;
        age = p.age + 1;     // для примера
    }
};
int main()
{
    Person tom{"Tom", 38};
    Person tomas{tom};     // создаем объект tomas на основе объекта tom
}
```

Конструктор переноса и оператор переноса был добавлен в C++ 11. Основная идея применения этих двух конструкций состоит в том, чтобы ускорить выполнение программы путем избегания копирования данных при начальной инициализации и присвоении так называемых *rvalue*-ссылок.

Конструктор переноса и оператор переноса целесообразно объявлять в классах, содержащих большие массивы данных.

Если в классе не реализован конструктор переноса, то его вызов заменяется конструктором копирования.

Общая форма объявления конструктора переноса в классе:

```
ClassName(ClassName&& rObj) noexcept
{
...
}
```

rObj – ссылка на ссылку на временный экземпляр класса, значение которого будет скопировано в текущий экземпляр.

В приведенной выше общей форме используется ключевое слово *noexcept*. Этот спецификатор указывает, что наш конструктор переноса не генерирует исключение или аварийно завершает свою работу. Компилятор рекомендует использовать слово noexcept для конструктора переноса. В конструкторе переноса не происходит никаких операций с памятью, а происходит простое присвоение указателя.

Если перед именем конструктора поставить слово *explicit*, это будет гарантировать, что конструктор неявно не вызовется. Все конструкторы с 1 параметром должны быть с этим модификатором, это позволяет избежать случайного приведения типов.

Ключевое слово *default* введено в C++ 11. Его использование указывает компилятору самостоятельно генерировать (использовать) соответствующую функцию класса, если таковая не объявлена в классе.

Общая форма использования ключевого слова default:

```
class ClassName
{
    // ...
    ClassName(parameters) = default;
    // ...
};
```

Ключевое слово *delete* используется в случаях, когда нужно запретить использование какого-либо конструктора, тогда его вызов приведет к ошибке.

Общая форма объявления конструктора класса с ключевым словом *delete*:

```
class ClassName
{
    // ...
    ClassName(parameters) = delete;
    // ...
};
```

6. Наследование в C++. Построение иерархии классов. Выделение общей части группы классов. Расщепление классов.

Наследование представляет один из ключевых аспектов объектно-ориентированного программирования, который позволяет наследовать функциональность одного класса (базового класса) в другом - производном классе.

В C ++ есть несколько типов наследования:

- публичный (public) — публичные (public) и защищенные (protected) данные наследуются без изменения уровня доступа к ним;
- защищенный (protected) — все унаследованные данные становятся защищенными;
- приватный (private) — все унаследованные данные становятся приватными.

В C ++ конструкторы и деструкторы не наследуются. Однако они вызываются, когда дочерний класс инициализирует свой объект. Конструкторы вызываются один за другим иерархически, начиная с базового класса и заканчивая последним производным классом. Деструкторы вызываются в обратном порядке.

Иерархия классов определяет взаимоотношения между ними. Т.е. при ее построении мы описываем само наследование, указывая есть ли расширение функционала, взаимосвязь не прямо наследуемых классов и т.п. В конечном итоге у нас получается общая структура всех классов.

Базовый класс выделяют в следующих случаях:

- Общая схема исполнения разных объектов.
- В объектах один и тот же набор методов.
- Имеются два разных класса с разными методами. Если у методов похожая реализация, то выделяем базовый класс.
- У нас имеется два разных класса. Если в дальнейшем они будут участвовать вместе, лучше сделать для них базовый класс уже на этом этапе разработки. Это нужно сделать, чтобы в дальнейшем нам было легко модифицировать программу.

Разбиваем класс в следующих случаях:

- Если один объект исполняет разные роли.
- Два множества методов используются в разной манере.
- Методы между собой никак не связаны.
- Одна сущность, но используется в разных частях программы.
- На классах возможно множественное наследование.

7. Множественное наследование. Прямая и косвенная базы. Виртуальное наследование. Понятие доминирования. Порядок создания и уничтожения объектов. Проблемы множественного наследования. Неоднозначности при множественном наследовании.

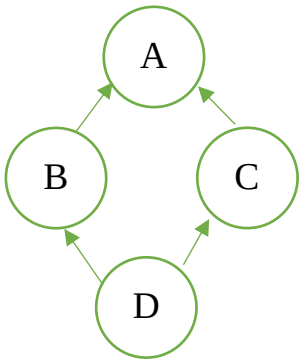
В C++ реализовано множественное наследование. Оно хорошо тем, что позволяет нам четко разделять наследуемые объекты, но на деле от него пытаются избавляться, в большинстве языков его уже нет.

Прямая база – база, непосредственно от которой мы породили нашу сущность.

Косвенная база – база, которая входит в нашу сущность, но от нее мы не порождались. Или же прямая база от прямой базы.

Виртуальное наследование.

```
class A{};
class B: virtual public A{};
class C: virtual public A{};
class D: public B, public C{};
```



Если убрать первый *virtual*, то класс A будет рассматриваться как подобъект класса D. И подобъект будет создаваться для класса D, потом будет создан подобъект класса B, а для подобъекта класса C создастся подобъект класса A, и потом создастся объект класса C и D.

Если мы не напишем второй *virtual*, то подобъект класса A не будет создаваться для класса D.

Если мы напишем два *virtual*, то подобъект класса A создастся только для класса D.

Конструкторы виртуальных баз вызываются в первую очередь.

Методы, определяемые в производных классах, доминируют над методами базовых классов. То есть они их подменяют. Для решения этой проблемы можно использовать *using*.

Проблемы множественного наследования:

- При ромбовидной иерархии как выше, что-то может отрабатывать два раза.

```
class A
{
public:
    void f() { cout << "Executing f from A;" << endl; }
};

class B : virtual public A
{
public:
    void f()
```

```
    {
        A::f();
        cout << "Executing f from B;" << endl;
    }
};

class C : virtual public A
{
public:
    void f()
    {
        A::f();
        cout << "Executing f from C;" << endl;
    }
};

class D : virtual public C, virtual public B
{
public:
    void f()
    {
        C::f();
        B::f();
        cout << "Executing f from D;" << endl;
    }
};

int main()
{
    D obj;

    obj.f();
}
```

Ее решение:

```
class A
{
protected:
    void _f() { cout << "Executing f from A;" << endl; }
public:
    void f() { this->_f(); }
};

class B : virtual public A
{
protected:
    void _f() { cout << "Executing f from B;" << endl; }
public:
    void f()
    {
        A::_f();
        this->_f();
    }
};

class C : virtual public A
{
protected:
    void _f() { cout << "Executing f from C;" << endl; }
public:
    void f()
    {
        A::_f();
        this->_f();
    }
```

```
};

class D : virtual public C, virtual public B
{
protected:
    void _f() { cout << "Executing f from D;" << endl; }
public:
    void f()
    {
        A::_f(); C::_f(); B::_f();
        this->_f();
    }
};

int main()
{
    D obj;

    obj.f();
}
```

- Неоднозначность

```
class A
{
public:
    int a;
    int (*b)();
    int f();
    int f(int);
    int g();
};

class B
{
    int a;
    int b;
public:
    int f();
    int g;
    int h();
    int h(int);
};

class C: public A, public B {};

class D
{
public:
    static void fun(C& obj)
    {
        // obj.a = 1;       // Error!!!
        // obj.b();            // Error!!!
        // obj.f();            // Error!!!
        // obj.f(1);        // Error!!!
        // obj.g = 1;       // Error!!!
        obj.h(); obj.h(1);  // Ok!
    }
};

int main()
{
    C obj;
    D::fun(obj);
}
```

- Подмена методов

```
class A
{
public:
    void f1() { cout << "Executing f1 from A;" << endl; }
    void f2() { cout << "Executing f2 from A;" << endl; }
};

class B
{
public:
    void f1() { cout << "Executing f1 from B;" << endl; }
    void f3() { cout << "Executing f3 from B;" << endl; }
};

class C : private A, public B {};

class D
{
public:
    void g1(A& obj)
    {
        obj.f1(); obj.f2();
    }
    void g2(B& obj)
    {
        obj.f1(); obj.f3();
    }
};


int main()
{
    C obj;
    D d;

    // obj.f1();  Error! Множественное определение
    // d.g1(obj); Error! Нет приведения к баз. классу при наследовании по схеме private
    d.g2(obj);
}
```

Исправить можно так:

```
class A
{
public:
    void f1() { cout << "Executing f1 from A;" << endl; }
    void f2() { cout << "Executing f2 from A;" << endl; }
};

class B
{
public:
    void f1() { cout << "Executing f1 from B;" << endl; }
    void f3() { cout << "Executing f3 from B;" << endl; }
};

class C : public A, public B {};

class D
{
public:
    void g1(A& obj)
    {
        obj.f1(); obj.f2();
```

```
    }
    void g2(B& obj)
    {
        obj.f1(); obj.f3();
    }
};


int main()
{
    C obj;
    D d;

    d.g1(obj);
    d.g2(obj);
}
```

8. Полиморфизм в C++. Виртуальные методы. Чисто виртуальные методы. Виртуальные и чисто виртуальные деструкторы. Понятие абстрактного класса. Ошибки, возникающие при работе с указателем или ссылкой на базовый класс. Дружественные связи.

Полиморфизм – возможность подменять одно другим, не изменяя написанный код. Возможность обработки разных типов данных, с помощью "одной и той же" функции, или метода.

Если в классе определен хотя бы один метод с модификатором *virtual*, то данный класс считается полиморфом и при создании объектов этих классов в них добавляется указатель на виртуальную таблицу, то есть на те методы, которые использует класс.

Ключевое слово *override* дает гарантию того, что метод является полиморфом и подменяет метод базового класса.

```
class A
{
public:
    virtual void f() { cout << "Executing f from A;" << endl; }
};

class B : public A
{
public:
    void f() override { cout << "Executing f from B;" << endl; }
};

class C
{
public:
    static void g(A& obj) { obj.f(); }
};

int main()
{
    B obj;
    C::g(obj);
}
```

Базовый класс всегда должен быть абстрактным. Его объекты создавать нельзя. Для того, чтобы создать абстрактный класс, в нем должен быть хотя бы один чисто виртуальный метод. Он задается следующим образом:

```
public:
    virtual void f() = 0;
```

Таким образом, производные классы должны подменить этот метод. Если они не реализуют этот метод, они тоже будут абстрактными.

Базовый класс всегда должен содержать виртуальный деструктор. Чисто виртуальный деструктор:

```
class A
{
public:
    virtual ~A() = 0;
};

A::~A() = default;

class B : public A
{
public:
    ~B() override { cout << "Class B destructor called;" << endl; }
};

int main()
{
    A* pobj = new B;
    delete pobj;
}
```

Ошибки, возникающие при работе с указателями на базовый класс:

- Подобъект может находится по другому адресу самого объекта внутри объекта. Если это произошло, то следующий код может выдавать непредсказуемый ответ:

```
A *pa = 0;
B *pb = 0;
if (pa == pb)
```

- Может случайно вызваться конструктор копирования:

```
void f(A obj);

B obj;
A &alias = obj;
f(alias);
```

Поэтому для полиморфного класса необходимо запретить/определить копирование с explicit.

- Нельзя работать с массивами объектов, вместо этого необходимо создавать структуры с указателями на объекты.

```
A &index(A *arr, int i)
{
    return arr[i];
}

B *ar - new B[10];
A obj= index(B, 2);
```

Дружественные связи. Мы можем дать объектам одного классам доступ ко всем членам объектов другого класса. Сделать это позволяет модификатор friend, при этом доступ является односторонним. В современных языках от этого избавились, т.к. это нарушает целостность объекта. Пример с наследованием:

```
class C; // forward объявление

class A
{
private:
    void f1() { cout << "Executing f1;" << endl; }

    friend C;
};

class B : public A
{
private:
    void f2() { cout << "Executing f2;" << endl; }
};

class C
{
public:
    static void g1(A& obj) { obj.f1(); }
    static void g2(B& obj)
    {
        obj.f1();
//    obj.f2(); // Error!!! Имеет доступ только к членам A
    }
};

class D : public C
{
public:
// static void g2(A& obj) ( obj.f1(); } // Error!!! Дружба не наследуется
};



int main()
{
    A aobj;

    C::g1(aobj);

    B bobj;

    C::g1(bobj);
    C::g2(bobj);
}
```

Дружба и виртуальные методы:

```
class C; // forward объявление

class A
{
protected:
    virtual ~A() = default;
    virtual void f() { cout << "Executing f from A;" << endl; }

    friend C;
};
```

```
class B : public A
{
protected:
    void f() override { cout << "Executing f from B;" << endl; }
};

class C
{
public:
    static void g(A& obj) { obj.f(); }
};

int main()
{
    B bobj;

    C::g(bobj);      // Executing f from B;
}
```

9. Обработка исключительных ситуаций в C++. Решение проблем структурного программирования. Блоки try и catch. Блоки try и catch методов и конструкторов. Безопасный код относительно исключений. Обертывание исключения в exception_ptr. Задачи, которые может решать исключение. Проблемы с динамической памятью при обработке исключительных ситуаций.

Обработка исключительных ситуаций выглядит следующим образом:

```
try
{
    ->throw <объект>;
}
catch(<тип> &<объект>) {}
```

- Инструкция try - заворачиваем в неё блок кода, в котором может произойти ошибка, и берём его под контроль.
- Если в блоке try возникает исключительная ситуация, мы можем перейти на обработчик catch. Обработчики идут непосредственно после блока try.
- Генерируем исключительную ситуацию, используя инструкцию throw.

Здесь нет приведения типов, т.е. жесткая типизация, за исключением перевода ссылки с базового класса на производный, и указателя с базового на производный. Обработчиков может быть несколько. Если на этом уровне ни один из *catch* не перехватил этот объект, то это передается на более высокий уровень. Ошибка может никем не перехватиться, в этом случае программа "падает".

Мы можем перехватить любые исключительные ситуации, используя catch с 3 точками. Этот обработчик должен быть в конце списка, иначе он перехватит любую ситуацию, и другие обработчики ниже не будут работать.

При пробросе все статические переменные уничтожаются, однако динамические не будут освобождены.

Недостатки обработки ошибок в структурном программировании:

- Если где-то возникает ошибка в коде, мы вынуждены "протащить" ее через все уровни абстракции/иерархии до того места, пока мы не сможем обработать эту ошибку.
- Весь код насыщен непрерывными проверками. Обработка ошибки совмещена вместе с кодом.

Теперь, при ОО подходе мы получили из недостатков структурного следующие плюсы:

- Мы не "протаскиваем ошибку"
- Вся обработка сводится в одно место

Мы можем обрабатывать исключительные ситуации в методах и конструкторах класса. Исключения могут быть пойманы и вне.

Безопасный относительно исключений код должен представлять одну из трех гарантий:

1. Функции, предоставляющие базовую гарантию, обещают, что если исключение будет возбуждено, то все в программе остается в корректном состоянии. Никакие объекты или структуры данных не повреждены, и все объекты находятся в непротиворечивом состоянии. Однако точное состояние программы может быть непредсказуемо.
2. Функции, предоставляющие строгую гарантию, обещают, что если исключение будет возбуждено, то состояние программы не изменится. Вызов такой функции является атомарным; если он завершился успешно, то все запланированные действия выполнены до конца, если же нет, то программа останется в таком состоянии, как будто функция никогда не вызывалась.
   Работать с функциями, представляющими такую гарантию, проще, чем с функциями, которые дают только базовую гарантию, потому что после их вызова может быть только два состояния программы: то, которое ожидается в результате ее успешного завершения, и то, которое было до ее вызова. Напротив, если исключение возникает в функции, представляющей только базовую гарантию, то программа может оказаться в любом корректном состоянии.
3. Функции, предоставляющие гарантию отсутствия исключений, обещают никогда не возбуждать исключений, потому что всегда делают то, что должны делать. Все операции над встроенными типами (например, целыми, указателями и т. п.) обеспечивают такую гарантию. Это основной строительный блок безопасного относительно исключений кода. Разумно предположить, что функции с пустой спецификацией исключений не возбуждают их, но это не всегда так.

std::exception_ptr:

Этот тип позволяет в себе хранить исключение абсолютно любого типа. Его поведение сходно с *std::shared_ptr*: его можно копировать, передавать в качестве параметра, при этом само исключение не копируется. Основное предназначение *exception_ptr* — это передача исключений в качестве параметров функции, возможна передача исключений между потоками. Таким образом, объекты данного типа позволяют сделать обработку ошибок более гибкой.

Проблемы с динамической памятью.

В C++, перейдя на обработчик, мы не можем вернуться в место возникновения ошибки (все временные объекты будут уничтожены). Предположим, у класса *A* есть метод *f()*. Если мы динамически выделили память:

```
try {
A* obj = new A; // Динамически выделили память под объект
obj->f();       // Если внутри функции f() произошла ошибка и мы вышли на обработчик,
delete obj;     // происходит утечка памяти
}
```

Если при вызове метода *f()* возникает исключительная ситуация и мы выходим на какой-то из обработчиков, объект *obj* не удаляется. Происходит утечка памяти.

Два варианта решения проблемы:

```
void A::f() noexcept // Первый способ
void A::f() throw()  // Второй способ
```

Модификатор *noexcept* «дает обещание» компилятору не обрабатывать исключение, тогда вызывается специальный метод *terminate,* задача которого очистить стек. Метод *terminate()* приводит к тому, что будут вызываться все деструкторы только временных объектов в порядке, обратном их созданию.

Со *throw* результат непредсказуем, это старый синтаксис, его лучше не использовать.

- Если пишем *noexcept* без параметров аналогичен *noexcept(True)* - это говорит о том, что данный метод не должен обрабатывать исключительную ситуацию.
- Если пишем *noexcept(False)* или *throw(...)*, то этот метод может обрабатывать все исключительные ситуации, как и в случае если ничего не пишем.

Любой деструктор по умолчанию не бросает исключения, т.к. может выйти бесконечный вызов деструктора. В блоке *catch* можно писать *throw*, тогда исключения необходимо ловить в «старшем» блоке.

std::exception содержит виртуальный метод *what*, и на основе него мы можем создавать свои классы. От него есть производный класс ошибок *std::bad_alloc*. При создании своих производный классов мы придерживаемся такого подхода, что отдельный класс обрабатывает только одну ситуацию. Даже стандартные ошибки мы будет перекладывать на себя, создавая свои.

Исключения решают следующие задачи:

- Исключают необходимость прокидывания ошибок через много уровней, все делает обработчик
- Разделяют логику программы от обработки ошибок, выносы обработчики отдельно
- Позволяют легко модифицировать и развивать ПО
- Буквально контролируют работу программы: создание объектов и выполнение действий над ними

Пример обработки исключительных ситуаций:

```
class ExceptionArray : public std::exception
{
protected:
    static const size_t sizebuff = 128;
    char errormsg[sizebuff]{};

public:
    ExceptionArray() noexcept = default;
    ExceptionArray(const char* msg) noexcept
    {
        strcpy_s(errormsg, sizebuff, msg);
    }
    ~ExceptionArray() override {}

    const char* what() const noexcept override { return errormsg; }
};

class ErrorIndex : public ExceptionArray
{
private:
    const char* errIndexMsg = "Error Index";
    int ind;

public:
    ErrorIndex(const char* msg, int index) noexcept : ind(index)
    {
        sprintf_s(errormsg, sizebuff, "%s %s: %4d!", msg, errIndexMsg, ind);
    }
    ~ErrorIndex() override {}

    const char* what() const noexcept override { return errormsg; }
};

int main()
{
    try
    {
        throw(ErrorIndex("Index!!", -1));
    }
    catch (const ExceptionArray& error)
    {
        cout << error.what() << endl;
    }
    catch (std::exception& error)
    {
        cout << error.what() << endl;
    }
    catch (...)
    {
        cout << "All errors!" << endl;
    }
}
```

10. Перегрузка операторов в C++. Операторы .*, ->*. Правила перегрузки операторов. Перегрузка операторов =, () и [ ]. Перегрузка операторов ->, * и –>*.

Перегрузка операторов – задание новой операции на основе существующего оператора. Это позволяет не заботится об используемом литерале, приоритете, арности и т.д. Есть операторы, которые нельзя перегружать, к ним относятся «.», «.*», «::», «?:» (тернарный), «sizeof», «typeid».

Посмотрим на очень интересный пример с функциями:

```
void f();      // Определили функцию f()
void (*pf)(); // Определили указатель на функцию
pf = f;          // Этот указатель инициализируем адресом функции(так как имя любой
функции - это ее адрес в памяти)
pf();          // Через указатель на функцию вызываем функцию
```

Оператор () - оператор разыменования - вызов функции по адресу. Так же вызвать функцию f() можно и таким образом: (*pf)().

Что касается методов класса:

```
void A::f();      // Метод класса A
void (A::*pf)(); // Указатель на метод класса A

// Хотелось бы проинициализировать этот указатель.
// pf = A::f; - Если мы таким образом напишем, мы получим не адрес этого метода
// Метод не находится в классе. Он вызывается по указателю, и чтобы получить этот
адрес,
// было принято решение добавить вот такой синтаксис
pf = &A::f;  // Вычисление адреса метода.

A obj;
(obj.*pf)(); // Чтобы вызвать метод через указатель, используется оператор .*
// Этот указатель имеет более низкий приоритет, чем (),
// поэтому, чтобы использовать (),
// надо повысить его приоритет, взяв obj.*pf в круглые скобки.

A* p = &obj;
(p->*pf)();  // Оператор ->* используется для указателя на объект
// В метод, на который указывает этот указатель, будет передаваться
// указатель на объект.
```

Таким образом, мы разделяем вызов функции и вызов метода. Если мы вызываем метод класса через указатель для объекта, используется оператор .*, а если работаем с указателем на объект, используется оператор ->.

Правила перегрузки операторов:

1. Операторы, которые можно перегрузить только как члены классов:
   - Оператор = - оператор присваивания (бинарный)
   - Оператор () - функтуатор (бинарный)
   - Оператор [] - индексация (бинарный)
   - Оператор -> - унарный
   - Оператор ->* - бинарный, так как принимает указатель на метод и объект, метод которого вызываем
2. Бинарные операторы можно перегружать как члены класса или как внешние функции-операторы. Это зависит от ситуации. Конечно, надо отдавать предпочтение члену класса. Если мы перегружаем бинарный оператор, как член класса, он принимает 1 параметр (второй параметр он принимает неявно - *this).
3. Унарные операторы перегружаем как члены класса.
4. Нельзя перегружать операторы «.», «.*», «::», «?:» (тернарный), «sizeof», «typeid»

```
class Array
{
    ...
public:
    bool operator==(const Array& arr) const;
    ...
};

// Первый вариант - перегрузка оператора, как члена класса
bool Array::operator==(const Array& arr) const
{...}

// Второй вариант - перегрузка оператора, как внешней функции
bool Array::operator!=(const Array& arr1, const Array& arr2) const
{...}
```

Перегрузка оператора ():

Этот оператор можно реализовать только как член класса. Он может иметь любое число параметров любого типа, тип возвращаемого значения также произвольный. Классы, с перегруженным оператором (), называются функциональными, их экземпляры называются функциональными объектами или функторами. Именно с помощью таких классов и объектов в C++ реализуется парадигма функционального программирования. Функциональные классы и объекты, используемые в стандартной библиотеке, в зависимости от назначения имеют свои названия: предикаты, компараторы, хеш-функции, аккумуляторы, удалители. В зависимости от контекста использования, стандартная библиотека предъявляет определенные требования к функциональным классам. Экземпляры этих классов должны быть копируемыми по значению, не модифицировать аргументы, не иметь побочных эффектов и изменяемое состояние (чистые функции), соответственно реализация перегрузки оператора () обычно является константным членом класса. Есть исключение — алгоритм std::for_each(), для него функциональный объект может модифицировать аргумент и иметь изменяемое состояние.

Перегрузка оператора []:

Этот бинарный оператор, который обычно называют индексатором, может быть реализован только, как функция-член, которая должна иметь ровно один параметр. Тип этого параметра произвольный, соответственно, перегрузок может быть несколько, для разных типов параметра. Индексатор обычно перегружается для «массивоподобных» типов, а также для других контейнеров, например ассоциативных массивов. Возвращаемое значение обычно является ссылкой на элемент контейнера. Также может быть возврат по значению, но следует иметь в виду, что при этом для получения адреса элемента нельзя будет использовать выражения &x[i], допустимые для встроенного индексатора. Такое выражение не будет компилироваться, если возвращаемый тип встроенный, и будет давать адрес временного объекта для пользовательского возвращаемого типа.

Индексатор часто перегружают в двух вариантах — константном и неконстантном.

```
T& operator[](int ind);
const T& operator[](int ind) const;
```

Первая версия позволяет модифицировать элемент, вторая только прочитать и она будет выбрана для константных экземпляров и в константных функциях-членах.

Перегрузка оператора =:

Если мы динамически выделяем память под члены класса, мы обязаны явно определить оператор присваивания или запретить. Дело в том, что для любого типа неявно определяется оператор присваивания, который побайтно копирует данные. Может выйти так, что два объекта указывают на одну область памяти. В большинстве случаев это не нужно.

Разница оператора присваивания с копированием - создает копию объекта, а оператор присваивания с переносом - захватывает временный объект.

Копирование - выделяем новую память и копируем из одной области в другую, а при переносе захватываем область того объекта, который получаем. Параметр обнуляем, чтобы при деструкторе не произошло удаления области памяти, которой мы захватили.

Перегрузка операторов -> и *:

Оператор -> перегружается как член класса, он унарный, принимающий один параметр и должен возвращать либо указатель, либо ссылку на объект. Совместно с указателем -> еще перегружается оператор *. Он помогает делать примерно то же самое - возвращает ссылку на объект, и по ссылке мы уже вызываем метод. Мы можем перегружать эти операторы как для константных, так и для не константных объектов.

```
class A
{
public:
    void f() const { cout << "Executing f from A;" << endl; }
};

class B
{
private:
    A* pobj;

public:
    B(A* p) : pobj(p) {}

    A* operator->() noexcept { return pobj; }
    const A* operator->() const noexcept { return pobj; }
    A& operator*() noexcept { return *pobj; }
    const A& operator*() const noexcept { return *pobj; }
};

void main()
{
    A a;

    B b1(&a);
    b1->f();

    const B b2(&a);
    (*b2).f();
}
```

Перегрузка оператора ->*:

Оператор ->* перегружается как член класса и является бинарным (*this и указатель на метод)

Класс Pointer по существу скрывает связь объекта который выбирает связку и указателя на метод. Создаем объект и вызывая перегруженный оператор, он возвращает нам объект, который отвечает за связь и этот объект имеет перегруженный оператор круглые скобочки. Он уже вызывает указатель.

```
class Callee
{
private:
    int index;

public:
    Callee(int i = 0) : index(i) {}
    int inc(int d) { return index += d; }
};

class Caller
{
public:
    using FnPtr = int (Callee::*)(int);

private:
    Callee* pobj;
    FnPtr ptr;

public:
    Caller(Callee* p, FnPtr pf) : pobj(p), ptr(pf) {}
    int operator ()(int d) { return (pobj->*ptr)(d); }
};

class Pointer
{
private:
    Callee* pce;

public:
    Pointer(int i) { pce = new Callee(i); }
    ~Pointer() { delete pce; }

    Caller operator->*(Caller::FnPtr pf) { return Caller(pce, pf); }
};

int main()
{
    Caller::FnPtr pn = &Callee::inc;

    Pointer pt(1);

    cout << "Result: " << (pt->*pn)(2) << endl;
}
```

11. Перегрузка унарных и бинарных операторов. Проблемы с перегрузкой операторов &&, ||, „ &. Перегрузка операторов ++, --. Перегрузка операторов приведения типов. Тривалентный оператор spaceship.

Перегрузка операторов – задание новой операции на основе существующего оператора. Это позволяет не заботится об используемом литерале, приоритете, арности и т.д. Есть операторы, которые нельзя перегружать, к ним относятся «.», «.*», «::», «?:» (тернарный), «sizeof», «typeid».

Унарные операторы перегружаем как члены класса.

Пример перегрузки бинарных операторов:

```
class Complex
{
private:
   double re, im;

public:
   Complex(double r = 0., double i = 0.) : re(r), im(i) {}

   Complex operator-() const { return Complex(-re, -im); }
   Complex operator-(const Complex& c) const { return Complex(re + c.re, im + c.im); }
   friend Complex operator+(const Complex& c1, const Complex& c2);

   friend ostream& operator<<(ostream& os, const Complex& c);
};

Complex operator+(const Complex& c1, const Complex& c2) {
   return Complex(c1.re + c2.re, c1.im + c2.im);
}

ostream& operator<<(ostream& os, const Complex& c) {
   return os << c.re << " + " << c.im << "i";
}

int main() {
   Complex c1(1., 1.), c2(1., 2.), c3(2., 1.);

   Complex c4 = c1 + c2;
   cout << c4 << endl;

   Complex c5 = 5 + c3;
   cout << c5 << endl;

   // Complex c6 = 6 - c3; Error!!!

   Complex c7 = -c1;
   cout << c7 << endl;
}
```

Не рекомендуется перегружать следующие три бинарных оператора: , (запятая), &&, ||. Дело в том, что для них стандарт предусматривает порядок вычисления операндов (слева направо), а для последних двух еще и так называемую семантику быстрых вычислений (short-circuit evaluation), но для перегруженных операторов это уже не гарантируется или просто бессмысленно, что может оказаться весьма неприятной неожиданностью для программиста. (Семантика быстрых вычислений заключается в том, для оператора && второй операнд не вычисляется, если первый равен false, а для оператора || второй операнд не вычисляется, если первый равен true.)

Также не рекомендуется перегружать унарный оператор & (взятие адреса). Тип с перегруженным оператором & опасно использовать с шаблонами, так как они могут использовать стандартную семантику этого оператора. Правда в C++11 появилась стандартная функция (точнее шаблон функции) std::addressof(), которая умеет получать адрес без оператора & и правильно написанные шаблоны должны использовать именно эту функцию вместо встроенного оператора.

Перегрузка унарных операторов, в том числе ++ и --:

```
class Integer
{
private:
    int value;
public:
    Integer(int i): value(i) {}
    //унарный +
    friend const Integer& operator+(const Integer& i);
    //унарный -
    friend const Integer operator-(const Integer& i);
    //префиксный инкремент
    friend const Integer& operator++(Integer& i);
    //постфиксный инкремент
    friend const Integer operator++(Integer& i, int);
    //префиксный декремент
    friend const Integer& operator--(Integer& i);
    //постфиксный декремент
    friend const Integer operator--(Integer& i, int);
};

//унарный плюс ничего не делает.
const Integer& operator+(const Integer& i) {
    return i.value;
}

const Integer operator-(const Integer& i) {
    return Integer(-i.value);
}

//префиксная версия возвращает значение после инкремента
const Integer& operator++(Integer& i) {
    i.value++;
    return i;
}

//постфиксная версия возвращает значение до инкремента
const Integer operator++(Integer& i, int) {
    Integer oldValue(i.value);
    i.value++;
    return oldValue;
}

//префиксная версия возвращает значение после декремента
const Integer& operator--(Integer& i) {
    i.value--;
    return i;
}

//постфиксная версия возвращает значение до декремента
const Integer operator--(Integer& i, int) {
    Integer oldValue(i.value);
    i.value--;
    return oldValue;
}
```

Можно реализовать оператор приведения типа с автоматическим выведением типа:

```
class A
{
private:
    int val;

public:
    A(int i) : val(i) {}

    operator auto() const& { return val; }
    operator auto()&& { return val; }
    operator auto* () const { return &val; }
};

int main()
{
    A obj{ 10 };

    int v1 = obj;               // operator auto() const&
    double v2 = obj;            // operator auto() const&
    const double& al = obj;     // operator auto() const&
    int v3 = std::move(obj);    // operator auto()&&
    const int* p = obj;         // operator auto*() const
}
```

Оператор space ship <=>:

Используется для сравнения двух объектов:

```
(a <=> b) < 0  //true if a < b
(a <=> b) > 0  //true if a > b
(a <=> b) == 0 //true if a is equal/equivalent to b
```

Данный оператор был добавлен в C++20 и может быть определен компилятором с помощью *default*, что сильно упрощает задачу во многих случаях, когда для класса необходимо написать операторы сравнения. Если не использовать space ship, придется отдельно перегружать <, >, <=, >=, ==. Рассмотрим такой пример:

```
# define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <compare>
#include <string.h>

using namespace std;

class MyInt {
public:
    constexpr MyInt(int val) : value{ val } { }
    auto operator<=>(const MyInt&) const = default;

private:
    int value;
};

class MyDouble {
public:
    constexpr MyDouble(double val) : value{ val } { }
    auto operator<=>(const MyDouble&) const = default;

private:
    double value;
};
```

```
class MyString {
public:
    constexpr MyString(const char* val) : value{ val } { }
    auto operator<=>(const MyString&) const = default;
private:
    const char* value;
};

int main() {
    MyInt i1{ 1 }, i2{ 2 };
    cout << (i1 < i2) << endl;

    MyDouble d1{ -0. }, d2{ 0. };
    cout << (d1 != d2) << (1. < d2) << (d1 < 2.) << endl;

    char st[5];
    strcpy(st, "Ok!!");
    MyString s1{ "Ok!" }, s2{ st };
    cout <<  (s1 < s2) << ("Ok!!" == s2) << endl; // сравнение адресов
}
```

Еще варианты перегрузки:

```
class MyInt {
private:
    int value;

public:
    MyInt(int val = 0) : value(val) {}

    //strong_ordering operator <=>(const MyInt& rhs) const
    //{
    //    return value <=> rhs.value;
    //}

    //strong_ordering operator <=>(const MyInt& rhs) const
    //{
    //    return value == rhs.value ? strong_ordering::equal :
    //           value < rhs.value ? strong_ordering::less :
    //                               strong_ordering::greater;
    //}

    //weak_ordering operator <=>(const MyInt& rhs) const
    //{
    //    return value == rhs.value ? weak_ordering::equivalent :
    //           value < rhs.value ? weak_ordering::less :
    //                               weak_ordering::greater;
    //}

    partial_ordering operator <=>(const MyInt& rhs) const {
        return value == rhs.value ? partial_ordering::equivalent :
               value < rhs.value ? partial_ordering::less :
               value > rhs.value ? partial_ordering::greater :
               partial_ordering::unordered;
    }

    bool operator ==(const MyInt&) const = default;
};

int main() {
    MyInt a{ 1 }, b{ 2 }, c{ 3 }, d{ 1 };
    cout << "a < b: " << (a < b) << ", c > b: " << (c >= b) << endl;
    cout << "a < b: " << (a < b) << ", c > b: " << (c > b) << ", a != b: " << (a != b)
<< endl;
    cout << "a < 5: " << (a < 5) << ", 1 < c: " << (1 < c) << endl;
}
```

12. Шаблоны функций, методов классов и классов в C++. Недостатки шаблонов. Параметры шаблонов. Параметры типы и параметры значения. Шаблоны функций и методов классов. Подстановка параметров в шаблон. Выведение типов параметров шаблона. Явное указание значений типов параметров шаблона при вызове функции. Срезание ссылок и модификатора const.

В языке Си приходилось создавать подобные функции, работающие с разными типами данных. По существу, был copy-past кода под разные типы данных. Можно решать эту проблему с помощью макросов с параметрами, но это крайне опасная вещь. На этапе препроцесирования происходит подстановка макроса в код, на этапе компиляции идет проверка подставленного. Если у нас ошибка в макросе, то определить её крайне сложно.

```
#define print(a) printf("%d \n", a)
#define min(a,b) (a < b ? a : b)
```

В языке C++ эта проблема подобного копирования кода решается с помощью шаблонов. Мы можем определить шаблон. Это может быть функция, класс, метод класса и также мы можем определить шаблон имени типа. Во время компиляции будет подстановка значения параметров шаблона с проверкой шаблона.

Шаблон не является ни классом, ни функцией. Функция или класс генерируется на основе шаблона во время использования. Компилятор встречает вызов функции, смотрит, может ли использоваться шаблон, и, если может, создаёт по шаблону функцию или класс.

Параметрами шаблона могут быть:

- Типы. Параметрами типа могут быть простые типы языка си, производные типы языка си и классы.
- Параметры значений. Параметры значения - только константные параметры целого типа или указатели с внешним связыванием.

Шаблоны можно определять:

- функций
- типов
- классов
- методов класса

Синтаксис шаблона в общем виде:

```
template<[параметры шаблона]>
функция | класс | тип
```

Программа, использующая шаблоны, содержит код для каждого порожденного типа, что может увеличить размер исполняемого файла. Кроме того, с одними типами данных шаблоны могут работать не так эффективно, как с другими. В этом случае имеет смысл использовать специализацию шаблона.

Возможно два варианта создания функции по шаблону:

- Функция принимает параметры шаблона, например, void freeArray(Type* arr);
- Явное указание параметров функции, например, Type* initArray(int count);

Пример шаблона функции и вызова функции:

```
template<typename Type>
void swap(Type&, Type&);
...
swap<double>(ar[i], ar[j]);
```

В обычном классе можно определить шаблонный метод.

Правило вызова. Компилятор рассматривает, может ли он вызвать перегруженную функцию. Если он находит перегруженную функцию, он ее вызывает. Если он не нашел ни одной перегруженной функции, он рассматривает специализации и подбирает подходящую. Если не подошла ни одна специализация, компилятор использует шаблон. То есть, шаблонная функция используется только в том случае, если для вызова функции компилятор не смог подобрать подходящую перегруженную функцию или функцию со специализацией.

Выведение типов параметров шаблона — это процесс, при котором компилятор определяет типы аргументов, передаваемых в параметры шаблона, на основе типов аргументов, передаваемых при вызове шаблона.

Когда используется шаблон функции или шаблон класса и передаются аргументы, компилятор анализирует их и определяет типы параметров шаблона, если они не были явно указаны. Это позволяет использовать шаблоны с разными типами данных без явного указания типов.

Можно явно указывать значения типов параметров шаблона:

```
template <typename T>
void print(T value) {
    std::cout << value << std::endl;
}

print<int>(10);     // Явное указание типа параметра шаблона как int
print<double>(3.14);  // Явное указание типа параметра шаблона как double
print<const char*>("Hello");  // Явное указание типа параметра шаблона как const char*
```

Срезание ссылок — это механизм, который определяет, как типы ссылок комбинируются при объявлении или использовании шаблонов с параметрами ссылочного типа.

- T& & становится T&
- T& && становится T&
- T&& & становится T&
- T&& && становится T&&

Выведение типов, срезание ссылок и const:

```
# define V_1

# ifdef V_1
template <typename T>
T f(T v) { return v; }

# elif defined(V_2)
template <typename T>
T f(T& v) { return v; }

# elif defined(V_3)
template <typename T>
T f(const T& v) { return v; }

# elif defined(V_4)
template <typename T>
T f(T&& v) { return v; }

# elif defined(V_5)
template <typename T>
T& f(T&& v) { return v; }

# elif defined(V_6)
template <typename T>
T&& f(T&& v) { return std::forward<T>(v); }

# elif defined(V_7)
auto f(auto v) { return v; }

# elif defined(V_8)
auto f(auto& v) { return v; }

# elif defined(V_9)
auto f(const auto& v) { return v; }

# elif defined(V_10)
auto&& f(auto&& v) { return v; }

# elif defined(V_11)
auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }

# elif defined(V_12)
decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }

# elif defined(V_13)
template <typename T>
auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }

# endif


int main()
{
    int i;
    int& a = i;
    const int& b = 0;

    decltype(auto) r1 = f(i);
    // 1. T f(T v) --->   int f<int>(int)
    // 2. T f(T& v)    --->   int f<int>(int&)
    // 3. T f(const T& v)  --->   int f<int>(const int&)
    // 4. T f(T&& v)   --->   int& f<int&>(int&)
    // 5. T& f(T&& v)  --->   int& f<int&>(int&)
    // 6. T&& f(T&& v) { return std::forward<T>(v); }  --->   int& f<int&>(int&)
    // 7. auto f(auto v)   --->   int f<int>(int)
```

```
    // 8. auto f(auto& v)  --->   int f<int>(int&)
    // 9. auto f(const auto& v)    --->   int f<int>(const int&)
    // 10. auto&& f(auto&& v)  --->   int& f<int&>(int&)
    // 11. auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   int& f<int&>(int&)
    // 12. decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   int& f<int&>(int&)
    // 13. auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }
    // --->   int& f<int&>(int&)

    decltype(auto) r2 = f(a);
    // 1. T f(T v) --->   int f<int>(int)
    // 2. T f(T& v)    --->   int f<int>(int&)
    // 3. T f(const T& v)  --->   int f<int>(const int&)
    // 4. T f(T&& v)   --->   int& f<int&>(int&)
    // 5. T& f(T&& v)  --->   int& f<int&>(int&)
    // 6. T&& f(T&& v) { return std::forward<T>(v); }  --->   int& f<int&>(int&)
    // 7. auto f(auto v)   --->   int f<int>(int)
    // 8. auto f(auto& v)  --->   int f<int>(int&)
    // 9. auto f(const auto& v)    --->   int f<int>(const int&)
    // 10. auto&& f(auto&& v)  --->   int& f<int&>(int&)
    // 11. auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   int& f<int&>(int&)
    // 12. decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   int& f<int&>(int&)
    // 13. auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }
    // --->   int& f<int&>(int&)

    decltype(auto) r3 = f(b);
    // 1. T f(T v) --->   int f<int>(int)
    // 2. T f(T& v)    --->   const int f<const int>(const int&)
    // 3. T f(const T& v)  --->   int f<int>(const int&)
    // 4. T f(T&& v)   --->   const int& f<const int&>(const int&)
    // 5. T& f(T&& v)  --->   const int& f<const int&>(const int&)
    // 6. T&& f(T&& v) { return std::forward<T>(v); }
    //     --->   const int& f<const int&>(const int&)
    // 7. auto f(auto v)   --->   int f<int>(int)
    // 8. auto f(auto& v)  --->   int f<const int>(const int&)
    // 9. auto f(const auto& v)    --->   int f<int>(const int&)
    // 10. auto&& f(auto&& v)  --->   const int& f<const int&>(const int&)
    // 11. auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   const int& f<const int&>(const int&)
    // 12. decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   const int& f<const int&>(const int&)
    // 13. auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }
    // --->   const int& f<const int&>(const int&)

    decltype(auto) r4 = f(std::move(a));
    // 1. T f(T v) --->   int f<int>(int)
    // 2. T f(T& v)    --->   Error!
    // 3. T f(const T& v)  --->   int f<int>(const int&)
    // 4. T f(T&& v)   --->   int f<int>(int)
    // 5. T& f(T&& v)  --->   int& f<int>(int&&)
    // 6. T&& f(T&& v) { return std::forward<T>(v); }  --->   int&& f<int>(int&&)
    // 7. auto f(auto v)   --->   int f<int>(int)
    // 8. auto f(auto& v)  --->   Error!
    // 9. auto f(const auto& v)    --->   int f<int>(const int&)
    // 10. auto&& f(auto&& v)  --->   int f<int>(int&&)
    // 11. auto&& f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   int&& f<int>(int&&)
    // 12. decltype(auto) f(auto&& v) { return std::forward<decltype(v)>(v); }
    //     --->   int&& f<int>(int&&)
    // 13. auto f(T&& v) -> decltype(v) { return std::forward<T>(v); }
    // --->   int&& f<int>(int&&)
}
```

13. Неявные шаблоны. Протаскивание типа передаваемого параметра через шаблон (шаблон std::forward). Определение типа с помощью decltype. decltype(auto).

В языке Си приходилось создавать подобные функции, работающие с разными типами данных. По существу, был copy-past кода под разные типы данных. Можно решать эту проблему с помощью макросов с параметрами, но это крайне опасная вещь. На этапе препроцесирования происходит подстановка макроса в код, на этапе компиляции идет проверка подставленного. Если у нас ошибка в макросе, то определить её крайне сложно.

```
#define print(a) printf("%d \n", a)
#define min(a,b) (a < b ? a : b)
```

В языке C++ эта проблема подобного копирования кода решается с помощью шаблонов. Мы можем определить шаблон. Это может быть функция, класс, метод класса и также мы можем определить шаблон имени типа. Во время компиляции будет подстановка значения параметров шаблона с проверкой шаблона.

Шаблон не является ни классом, ни функцией. Функция или класс генерируется на основе шаблона во время использования. Компилятор встречает вызов функции, смотрит, может ли использоваться шаблон, и, если может, создаёт по шаблону функцию или класс.

Неявные шаблоны – функции, в которых вместо типов используется *auto*. Происходит то же самое, что и при обычных шаблонах, но встречаются некоторые нюансы.

```
auto f(auto v)
```

Но если мы хотим вернуть тип v, может произойти срезание ссылок (если бы мы передавали, например, *&&v*). Так же невозможно определить функцию, например, в заголовочном файле, т.к. ее тип не известен.

Срезание ссылок — это механизм, который определяет, как типы ссылок комбинируются при объявлении или использовании шаблонов с параметрами ссылочного типа.

- T& & становится T&
- T& && становится T&
- T&& & становится T&
- T&& && становится T&&

Decltype определяет тип выражения. С помощью него можно избежать срезания ссылок.

Decltype(auto) позволяет определить функцию с неизвестным возвращаемым типом, и, главное, избежать срезания ссылок.

```
decltype(auto) f(auto &&v)
{
    return std::forwart<decltype(v)>(v);
}
```

Использование *forward* для идеальной передачи (*lvalue-copy, rvalue-move*):

```
class A
{
public:
    A() = default;
    A(const A&) { cout << "Copy constructor" << endl; }
    A(A&&) noexcept { cout << "Move constructor" << endl; }
};

template <typename Func, typename Arg>
decltype(auto) call(Func&& func, Arg&& arg)
{
//    return func(arg);
    return forward<Func>(func)(forward<Arg>(arg));
}

A f(A a) { cout << "f called" << endl; return a; }

int main()
{
    A obj{};

    auto r1 = call(f, obj);
    cout << endl;
    auto r2 = call(f, move(obj));
}
```

14. Специализация шаблонов функций. Шаблоны типов. Шаблоны классов. Полная или частичная специализация шаблонов классов. Параметры шаблонов, задаваемых по умолчанию. Шаблоны с переменным числом параметров. Пространства имен.

В языке Си приходилось создавать подобные функции, работающие с разными типами данных. По существу, был copy-past кода под разные типы данных. Можно решать эту проблему с помощью макросов с параметрами, но это крайне опасная вещь. На этапе препроцесирования происходит подстановка макроса в код, на этапе компиляции идет проверка подставленного. Если у нас ошибка в макросе, то определить её крайне сложно.

```
#define print(a) printf("%d \n", a)
#define min(a,b) (a < b ? a : b)
```

В языке C++ эта проблема подобного копирования кода решается с помощью шаблонов. Мы можем определить шаблон. Это может быть функция, класс, метод класса и также мы можем определить шаблон имени типа. Во время компиляции будет подстановка значения параметров шаблона с проверкой шаблона.

Шаблон не является ни классом, ни функцией. Функция или класс генерируется на основе шаблона во время использования. Компилятор встречает вызов функции, смотрит, может ли использоваться шаблон, и, если может, создаёт по шаблону функцию или класс.

Шаблонный тип. В языке Си для задания имени типа использовался typedef. Это аналог определения какой-либо переменной, только вместо этого - имя типа.

Например, определим имя для типа, принимающего указатель на функцию, принимающую int и возвращающую void:

```
typedef void (*Tpf)(int);
```

В языке C++ шаблон на основе typedef мы определить не можем. Добавляется еще одна конструкция - using, выполняющая ту же функцию, что и typedef. После using записываем имя типа, а далее записывается так называемый абстрактный описатель (определение переменной без ее имени).

```
using Tpf = void (*)(int);
```

Две вышеприведенные записи с typedef и using эквивалентны, но для typedef мы не можем определить шаблон. Пример шаблонного типа:

```
template<typename T>
using cmp_t = int(*) (const T&, const T&);
//...
cmp_t<double> v;
```

Определение шаблона класса:

```
template <typename T>
class A
{
    T elem;
public:
    void f();
};
```

Методы шаблона класса также являются шаблонами. В принципе, у нас может быть нешаблонный класс, но с шаблонными методами, то есть в обычном классе можно определить шаблонный метод. Шаблонные методы поддерживают шаблон класса. Методы будут создаваться только для того типа, которого класс. Не будет проблемы срезания ссылок.

Так же, как и у любой функции, у шаблона метода может быть специализация.

Для вызовов методов действует такое же правило, как и для функций: сначала для созданного класса, если есть специализация выбирается специализация, если нет специализации, создается метод по шаблону.

Полная специализация.

```
template <typename T>
class A {...};

template <>
class A<float> {...};
```

Специализация является тоже шаблоном. По специализации так же создается класс, если нужно. У нас создаются классы по специализации. Специализация может иметь отличные параметры от шаблонов, вернее тело специализации может быть другим: другие члены, данные, методы, отличные от самого шаблоны.

Пример:

```
template <typename Type>
class A
{
public:
    A() { cout << "constructor of template A;" << endl; }
    void f() { cout << "metod f of template A;" << endl; }
};

template <>
void A<int>::f() { cout << "specialization of metod f of template A;" << endl; }

template <>
class A<float>
{
public:
    A() { cout << "specialization constructor template A;" << endl; }
    void f() { cout << "metod f specialization template A;" << endl; }
    void g() { cout << "metod g specialization template A;" << endl; }
};

int main()
{
    A<double> obj1;
    obj1.f();

    A<float> obj2;
    obj2.f();
    obj2.g();

    A<int> obj3;
    obj3.f();
}
```

Частичная специализация - когда мы указываем не все значения параметра шаблона. При частичной специализации шаблонов возможна неоднозначность при вызове. Здесь аналогичный подход с функциями: сначала идет выбор специализации, если невозможно создать класс по специализации, создается класс по шаблону.

```
template <typename T1, typename T2 = double>
class A
{
public:
    A() { cout << "constructor of template A<T1, T2>;" << endl; }
};

// Specialization #1
template <typename T>
class A<T, T>
{
public:
    A() { cout << "constructor of template A<T, T>;" << endl; }
};

// Specialization #2
template <typename T>
class A<T, int>
{
public:
    A() { cout << "constructor of template A<T, int>;" << endl; }
};
```

```cpp
// Specialization #3
template <typename T1, typename T2>
class A<T1*, T2*> {
public:
    A() { cout << "constructor of template A<T1*, T2*>;" << endl; }
};


int main()
{
    A<int> a0;           // Template
    A<int, float> a1;      // Template
    A<float, float> a2;        // Specialization #1
    A<float, int> a3;      // Specialization #2
    A<int*, float*> a4;        // Specialization #3

    // A<int, int> a5;        // Error!!!
    // A<int*, int*> a6;   // Error!!!
}
```

Так же, как при определении функций, параметры шаблона могут быть по умолчанию. В случае выше сам шаблон имеет один параметр по умолчанию - double. Если мы передаем только один параметр, будет вызываться этот шаблон (а второй параметр по умолчанию типа double):

```cpp
template <typename T1, typename T2 = double>
class A {
public:
    A() { cout << "constructor of template A<T1, T2>;" << endl; }
};
```

Так же как функции, шаблоны мы можем создавать с переменным числом параметров. Это могут быть шаблоны функций, методов и классов.

```cpp
template <typename Type>
Type sum(Type value)
{
    return value;
}

template <typename Type, typename ...Args>
Type sum(Type value, Args... args)
{
    return value + sum(args...);
}

int main()
{
    cout << sum(1, 2, 3, 4, 5) << endl;
    return 0;
}
```

Программа растет, она становится большой. Во время разрастания программы может возникнуть проблема конфликта имён. Мы используем разные библиотеки, в одной библиотеке нужно что-то взять, в другой что-то взять. у некоторых библиотек может дублироваться функционал. Это типичная ситуация. Например, несколько реализаций функции с именем swap в различных библиотеках. Когда подключаем разные библиотеки, может возникнуть конфликт имен. Из какой библиотеки мы вызываем swap? Это так же может касаться имен классов, методов, которые мы используем.

В C++ мы можем задавать пространства имён.

Синтаксис такой:

```
namespace <имя>
{
<блок пространства имён>
}

// Доступ к пространству имён
<имя>::f(); // f() - член пространства имен
```

Или можно сделать так:

```
// или можно включить это пространство и использовать f
using namespace <имя>;
f(); // но таким образом можно вернуться к изначальной проблеме
```

В примере выше тоже можно натолкнуться на проблему конфликта имён, так как подключив несколько namespace, может возникнуть та же самая ситуация. Поэтому, когда у нас есть много пространств имён с пересекающимися параметрами, лучше использовать синтаксис :: - это поможет избежать конфликта.

Пространствами имён злоупотреблять не надо. Вложенных пространств имён надо избегать или сводить к минимуму.

Имя пространства имён может отсутствовать — это анонимные пространства имён. Их особенность в том, что из другого файла нельзя получить доступ к членам анонимного пространства имён. Грубо говоря, если у нас часть, и мы не хотим, чтобы она была видна из других частей, мы можем определить это, как анонимное пространство имен.

15. Ограничения, накладываемые на шаблоны. Требования к шаблонам (requires). Концепты. Типы ограничений. Варианты определения шаблонов функций и классов с концептами.

Ограничения шаблонов (как функций, так и классов) позволяют ограничить набор возможных типов, которые будут применяться параметрами шаблонов. Добавляя ограничения к параметрам шаблона, решаются следующие задачи:

- Из заголовка шаблона сразу видно, какие аргументы шаблона разрешены, а какие нет.
- Шаблон создается только в том случае, если аргументы шаблона удовлетворяют всем ограничениям.
- Любое нарушение ограничений шаблона приводит к сообщению об ошибке, которое гораздо ближе к первопричине проблемы, а именно к попытке использовать шаблон с неверными аргументами.

Начиная со стандарта C++20 в язык был добавлен оператор requires, который позволяет установить для параметров шаблонов ограничения.

```
template <параметры> requires ограничения
содержимое шаблона;
```

```
#include <iostream>

template <typename T> requires std::is_same<T, int>::value || std::is_same<T,
double>::value
T sum(T a, T b){ return a + b;}

int main()
{
    std::cout << sum(3, 4) << std::endl;
    std::cout << sum(12.5, 4.3) << std::endl;
    //std::cout << sum(5l, 7l) << std::endl;
}
```

Начиная со стандарта C++20 в язык C++ была добавлена такая функциональность как concepts (концепты). Концепты позволяют установить ограничения для параметров шаблонов (как шаблонов функций, так и шаблонов класса).

Концепт фактически представляет шаблон для именованного набора ограничений, где каждое ограничение предписывает одно или несколько требований для одного или нескольких параметров шаблона. В общем случае он имеет следующий вид:

```
template <параметры>
concept имя_концепта = ограничения;
```

Список параметров концепта содержит один или несколько параметров шаблона. Во время компиляции компилятор оценивает концепты, чтобы определить, удовлетворяет ли набор аргументов заданным ограничениям.

Простейший пример:

```
template <typename T>
concept size = sizeof(T) <= sizeof(int);
```

Концепты бывают 4 видов:

1. Простые
2. Составные
3. Типовые
4. Вложенные

Пример:

```
# include <iostream>
# include <vector>

using namespace std;

template <typename T>
concept HasBeginEnd = requires(T a)
{
    a.begin();
    a.end();
};

# define PRIM_1

# ifdef PRIM_1
template <typename T>
requires HasBeginEnd<T>
```

```
ostream& operator <<(ostream& out, const T& v)
{
    for (const auto& elem : v)
        out << elem << endl;

    return out;
}

# elif defined(PRIM_2)
template <HasBeginEnd T>
ostream& operator <<(ostream& out, const T& v)
{
    for (const auto& elem : v)
        out << elem << endl;

    return out;
}

# elif defined(PRIM_3)
ostream& operator <<(ostream& out, const HasBeginEnd  auto& v)
{
    for (const auto& elem : v)
        out << elem << endl;

    return out;
}

# endif

int main()
{
    vector<double> v{ 1., 2., 3., 4., 5. };
    cout << v;
}
```

Варианты использования концепта:

```
template <typename T>
concept Incrementable = requires(T t)
{
    {++t} noexcept;
    t++;
};

# define PRIM_1

# ifdef PRIM_1
template <typename T>
requires Incrementable<T>
auto inc(T& arg)
{
    return ++arg;
}

# elif defined(PRIM_2)
template <typename T>
auto inc(T& arg) requires Incrementable<T>
{
    return ++arg;
}

# elif defined(PRIM_3)
template <Incrementable T>
auto inc(T& arg)
{
    return ++arg;
}
```

```
# elif defined(PRIM_4)
auto inc(Incrementable auto& arg)
{
    return ++arg;
}

# elif defined(PRIM_5)
template <typename T>
requires requires(T t)
{
    {++t} noexcept;
    {t++};
}
auto inc(T& arg)
{
    return ++arg;
}

# endif

class A {};

int main() {
    int i = 0;

    cout << "i = " << inc(i) << endl;

    A obj{};
    //    cout << "obj = " << inc(obj) << endl;
}
```

16. Проблемы с динамическим выделением и освобождением памяти. Шаблон Holder. «Умные указатели» в C++: unique_ptr, shared_ptr, weak_ptr. Связь между shared_ptr и weak_ptr.

Существует проблема. Предположим, у нас есть класс A, в котором есть метод f(). Мы не знаем, что творится внутри f(), и, естественно, мы используем механизм обработки исключительных ситуаций. Внутри f() происходит исключительная ситуация, она приводит к тому, что мы перескакиваем на какой-то обработчик, неизвестно где находящийся. Это приводит к тому, что объект p не удаляется - происходит утечка памяти.

```
A* p = new A;
p->f();   // Внутри f() происходит исключительная ситуация
delete p; // Объект p не удаляется
```

Идея: обернуть объект в оболочку, которая статическая распределяет память. Эта оболочка будет отвечать за этот указатель. И соответственно, поскольку мы статически распределили, когда будет вызываться деструктор, в деструкторе мы будем освобождать память.

Шаблон Holder. Мы можем указатель p обернуть в объект-хранитель. Этот объект будет содержать указатель на объект A. Задача объекта: при выходе из области видимости объекта-хранителя будет вызываться деструктор obj, в котором мы можем уничтожить объект A.

```
Holder<A> obj(new A);
```

Для объекта хранителя достаточно определить три операции - * (получить значение по указателю), ->(обратиться к методу объекта, на который указывает указатель) и bool(проверить, указатель указывает на объект, nullptr он или нет). Чтобы можно было записать obj->f();. То есть эта оболочка должна быть "прозрачной". Её задача должна быть только вовремя освободить память, выделенную под объект. Мы работаем с объектом класса A через эту оболочку.

```
template <typename Type>
class Holder
{
private:
    Type* ptr{ nullptr };

public:
    Holder() = default;
    explicit Holder(Type* p) : ptr(p) {}
    Holder(Holder&& other) noexcept
    {
        ptr = other.ptr;
        other.ptr = nullptr;
    }
    ~Holder() { delete ptr; }

    Type* operator ->() noexcept { return ptr; }
    Type& operator *() noexcept { return *ptr; }
    operator bool() noexcept { return ptr != nullptr; }
    Type* release() noexcept
    {
        Type* work = ptr;
        ptr = nullptr;

        return work;
    }

    Holder(const Holder&) = delete;
    Holder& operator =(const Holder&) = delete;
};

class A
{
public:
    void f() { cout << "Function f of class A is called" << endl; }
};

int main()
{
    Holder<A> obj(new A{});

    obj->f();
}
```

Проблема висящего указателя. Этот хранитель решает ситуацию, связанную с обработкой исключительных ситуаций. Но предположим, что у нас есть один объект класса A и класс B держит указатель на объект класса A.

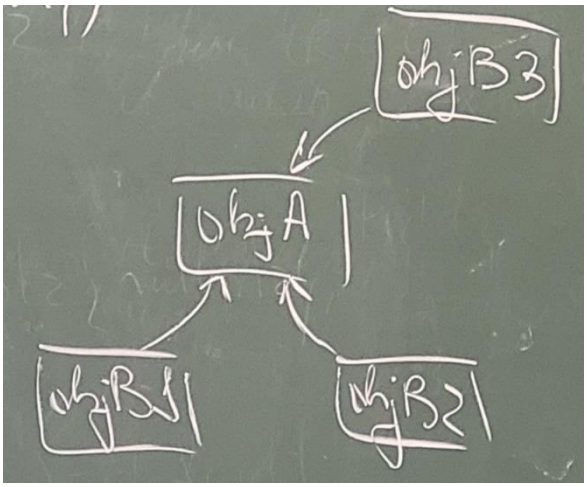
```
class A {...};

class B
{
    A* p;
}
```

Например, мы получили указатель p. Этот объект может быть удалён, и в этом случае возникает проблема: указатель, инициализированный каким-то адресом, будет указывать на удалённый объект. Можно рассматривать каждый объект, который держит указатель, как хранитель. То есть мы отдаём указатель на объект, а объект-хранитель считает, что этот объект его собственный, происходит захват.

В случае если хранитель отдаёт объект, нужно позаботиться о том, чтобы не образовался "висящий" указатель, то есть указатель на объект, которого нет.

Проблема с утечкой памяти не такая острая как проблема с висящим указателем. Утечка памяти приводит всего лишь к нехватке памяти, в то время как с висящим указателем мы можем случайно вызвать метод несуществующего объекта, что приведёт к падению системы.

Представим, что на один объект держат указатели несколько объектов. Как понять, какой из объектов должен удалять этот указатель? Если это отдавать на откуп программиста, то о надежности такого кода говорить нельзя, возможно ошибка. Допустим, мы выбрали один из объектов ответственным. Какая гарантия, что он не уничтожится раньше, чем другие два объекта?



Идея: последний, кто уезжает, выключает свет. То есть последний объект (класса B), который будет уничтожаться, он должен позаботиться об объекте класса A.

Умные указатели решают проблемы с утечкой памяти и с висящим указателем. Первоначально эту проблему пытались решить одним указателем, но скоро поняли, что одним указателем решить проблему невозможно.

Существует три вида умных указателей, каждый решает свою проблемы.

unique_ptr

Пример с Holder (ранее), по существу, представляет собой указатель unique_ptr. Жестко сберегает какой-то один объекта. Он хранит уникальную ссылку на объект и не позволяет другим указателям владеть этим объектом.

Применение unique_ptr:

```
class A
{
public:
    A() { cout << "Constructor" << endl; }
    ~A() { cout << "Destructor" << endl; }

    void f() { cout << "Function f" << endl; }
};

int main() {
    unique_ptr<A> obj1(new A{});
    unique_ptr<A> obj2 = make_unique<A>();
    unique_ptr<A> obj3(obj1.release()); // move(obj1)
```

```
    obj1 = move(obj3);

    if (!obj3) {
        A *p = obj1.release();

        obj2.reset(p);
        obj2->f();
    }
}
```

shared_ptr и weak_ptr

Если shared_ptr обеспечивает нас счетчиком, это так называемое совместное владение, то а паре с ним идет weak_ptr - слабое владение. Этот указатель не отвечает за освобождение памяти из-под объекта. Он может только проверить, есть объект или его нет. Эти два указателя связаны между собой.

У нас должен быть счетчик countS, определяющий, сколько объектов указывают на сберегаемый объект. Указателю weak_ptr тоже надо знать об этом счетчике.

Пусть есть какой-то базовый класс, от которого порождаем два класса: shared_ptr и weak_ptr. И тому и другому нужен указатель на object и нужен счетчик countS. Базовый класс содержит указатель на объект, и счетчик countS тоже должен быть доступен всем shared_ptr и weak_ptr, следовательно, счетчик countS мы тоже должны вынести, как объект.

Так как память object вынесли, по счетчику countS weak_ptr определяет, есть ли этот object или нет. Если счетчик равен нулю, то объекта нет. Когда создается новый shared_ptr на область памяти object, счетчик countS увеличивается. Удаляется shared_ptr - счетчик уменьшается. Если счетчик равен нулю - эта память должна быть освобождена.

А что будет отвечать за память счетчика countS, когда освободится память из-под object? Здесь встает необходимость считать не только количество объектов shared_ptr, но и количество объектов weak_ptr. То есть, по существу, у нас не один счетчик, а два. Счетчик weak_ptr нужен для того, что, если он станет равен нулю, и второй счетчик равен нулю, освободить эту память. Соответственно, общий класс для shared_ptr и weak_ptr может решать эту проблему. Он будет контролировать и память объектов, и область счетчика.

Memory	Владение	Операторы	Копия	Методы
unique_ptr	строгое	`*, ->, bool, []`	Нет	get, release, reset, swap
shared_ptr	совместное	`*, ->, bool, []`	Да	get, reset, use_count, unique (true, если счётчик shared равен 1, иначе false)
weak_ptr	слабое	Нет	Да	use_count, expired (возвращает признак, есть объект или его нет), reset, lock (возвращает shared_ptr, на основе weak мы создаём shared)

```cpp
class SomeClass {
public:
    void sayHello() {
        std::cout << "Hello!" << std::endl;
    }
    ~SomeClass() {
        std::cout << "~SomeClass" << std::endl;
    }
};

int main() {
    std::weak_ptr<SomeClass> wptr;
    {
        auto ptr = std::make_shared<SomeClass>();
        wptr = ptr;

        if(auto tptr = wptr.lock())
            tptr->sayHello();                          // !
        else
            std::cout << "lock() failed" << std::endl;
    }

    if(auto tptr = wptr.lock())
        tptr->sayHello();
    else
        std::cout << "lock() failed" << std::endl;     // !
}
```

17. Приведение типа в C++: static_cast, dynamic_cast, const_cast, reinterpret_cast. Контейнерные классы и итераторы. Требования к контейнерам и итераторам. Категории итераторов. Операции над итераторами. Цикл for для работы с контейнерными объектами.

В языке C++ решили использовать разные варианты приведения типов.

По умолчанию всегда указатель на производный класс приводится к указателю на базовый класс (или ссылка). А если нужно обратное преобразование?

Для обратного преобразования появились два оператора преобразования:

- Первый оператор преобразования выполняется на этапе компиляции - static_cast
- Второй оператор преобразования на этапе выполнения - dynamic_cast.

static_cast

На этапе компиляции выполняется оператор static_cast. Этот оператор используется для приведения родственных классов, находящихся по одной ветви наследования. Также используется для стандартных типов, для которых определён механизм явного приведения.

Рассмотрим следующую иерархию:

С помощью static_cast мы можем от указателя A привестись к указателю B или к указателю на класс C, или от A к D. Он не позволит нам привестись от класса указателя на B к D. Они родственные, но находятся по разным ветвям.

```cpp
A *pa = new B;               // У нас есть указатель
B *pb = static_cast<B*>(pa); // Приведение
```

Проблема — это выполняется на этапе компиляции. На этапе компиляции невозможно проверить, что это за объект, то есть приведение будет срабатывать, но указатель pa может не указывать на объект класса B. Мы можем написать такую строчку:

```
A *pa = new B;                // У нас есть указатель
C *pc = static_cast<C*>(pa); // Приведение
```

Что будет в таком случае – неизвестно.

Если приведение невозможно, будет выдаваться ошибка на этапе компиляции. Хотелось бы, чтобы это проверялось на этапе выполнения.

dynamic_cast

Оператор dynamic_cast делает проверку на этапе выполнения. Он приводит к типу, если реально указатель указывает на объект этого типа. Если нет - возвращает указатель на NULL. Приведение может быть не только указателем, но и ссылкой. Для dynamic_cast есть жесткое требование - базовый классы должен быть полиморфными (то есть либо virtual метод, либо virtual деструктор).

Пример:

```
pb = dynamic_cast<B*>(pa);
if (pb)
```

Можно проверить: если pb не равно нулю, приведение осуществилось, иначе не осуществилось. В данном случае всё будет нормально, так как pa указывает на объект класса B - приведение возможно. А приведение, например, к указателю на класс C невозможно из pa, так как pa указывает на объект другого класса.

Такое приведение типов удобно тем, что это удобно на этапе выполнение программ.

Если работаем со ссылкой в dynamic_cast - вместо NULL возникает исключение bad_cast, которое можно отловить. Исключения ловить неприятно, поэтому лучше работать с указателями.

const_cast

Мы работаем с модификатором const. Мы контролируем, что объект может быть константными, контролируем методы. Но есть проблема - мы не можем менять поля константных объектов. Иногда возникают ситуации, когда нам необходимо менять.

Предположим, у нас есть объект, который держит указатель на другой. Мы определили его, как константный, но этот указатель мы хотим отобрать от него. Чтобы отобрать, нам нужно это поле обнулить. А сделать это мы не можем, так как не можем обнулять поля константных объектов.

Чтобы убрать модификатор const, используется оператор const_cast. Есть компиляторы, которые не позволяют изменять константность объектов.

reinterpret_cast

Также существует оператор, эквивалентный С-му приведению - reinterpret_cast. Может приводить из любого типа.

```
class A {...};
A* p = new A;
char* pbyte = reinterpret_cast<char*>(p); // Можем выполнить такой бандитизм
```

Мы поставили указатель типа char на первый байт объекта класса А.

Это то же самое, как преобразование, которое было в языке Си. Небезопасное преобразование. Неизвестно, к чему это приведет.

Контейнерные классы и итераторы. Пусть у нас есть класс:

```
struct List
{
    Node *first, *last;
};
```

Идея: выделить текущий указатель в структуру и создавать по надобности:

```
struct Iterator
{
    Node *curent;
};
```

Стоит задача унификации работы с контейнерными данными. Можно идти по пути унификации интерфейса, чтобы разные контейнерные классы имели один и тот же интерфейс. Желательно, чтобы каждая сущность имела свой интерфейс.

Идея простая: унифицировать работу с разными контейнерами за счёт итератора. А у итератора будет унифицированный интерфейс.

Есть базовый шаблон - итератора, и уже конкретный итератор - специализация. 5 видов специализации. Специализация определяется тэгом. Тэг - простейшая структура.

```
struct output_iterator_tag {}; // итератор вывода
struct input_iterator_tag {}; // итератор ввода
struct forward_iterator_tag {}; // однонаправленный итератора на чтение-запись
struct bi_directional_iterator_tag {}; // двунаправленный итератор на чтение-запись
struct random_access_iterator_tag {}; // произвольный доступ
```

Первый допускает операторы - *, ++

Второй, третий и четвертый - *, ->, ++, !=, ==

Пятый – it - n, it + n, += , ==, it1 - it2, []

Раньше мы вынуждены были любой итератор порождает от этих базовых итераторов. В принципе, в современных стандартах можно свой итератор не порождать от этих стандартных итераторов.

В языке C++ появился цикл foreach:

```
for (auto elem: obj)
cout << elem;
// Цикл выше разворачивается в
for (Iterator<Type> It = obj.begin(); It != obj.end(); ++It) {
auto elem = *It;
cout << elem;
}
```

Мы обязаны в контейнерный класс добавить методы: begin, end, cbegin, cend (c - const), rbegin, rend, crbegin, crend. Также добавить операторы: !=, ++, * (++ - префиксный инкремент).

Контейнер может быть хранитель, но есть проблема. Мы на контейнеры можем создавать итераторы. По этой причине "голенький" указатель в контейнере хранить нельзя. То есть итератор должен хранить weak_ptr. Контейнер должен оборачивать данные в shared_ptr.

Существуют классы, которые содержат в себе другие классы. Например, множество, вектор и т. п. Такие классы называются контейнерными - они включают в себя другие классы.

Для работы с контейнерными классами, нам, во-первых, необходимо абстрагироваться от внутренней структуры организации контейнера - нас это не должно интересовать, а во-вторых, мы можем обрабатывать контейнер вне зависимости от типа объекта внутри контейнера. Для этого были придуманы классы-итераторы. Существуют стандартные итераторы, поэтому задача программиста - задавать общий механизм работы с итераторами. Классы, которые отвечают за просмотр содержимого в других объектах, называют итераторами.

Есть стандартный итератор - шаблон класса Iterator. У него есть специализации под разные виды работы с итераторами. Итераторы делятся на итераторы ввода (мы можем менять то, на что итератор указывает) и итераторы вывода (мы НЕ можем менять то, на что итератор указывает, то есть мы можем только читать, но не записывать).

Специализация ввода или вывода задаётся при наследовании пользовательского итератора от стандартного итератора.

Итератор может рассматривать контейнер как направленную последовательность, двунаправленную последовательность и последовательность произвольного доступа.

С помощью итератора можно просматривать содержимое контейнера.

Благодаря итераторам в C++ стало возможным создать оператор foreach. Для работы с этим оператором контейнерный класс должен содержать два метода: метод begin(), указывающий на начало последовательности, и метод end(), указывающий на конец последовательности. Конец — это не последний элемент, а ЗА последним элементом.

```
for (<тип>& <имя> : <объект>)
{
...
}
```

При реализации следует предусмотреть итераторы для работы как с константным контейнерным классом, так и с неконстантным.

Теоретические вопросы

1. Структурное программирование: нисходящая разработка, использование базовых логических структур, сквозной структурный контроль.

Задача технологии структурного программирования - предложить набор методов, которые позволят решать сложные задачи (повышать надежность кода, увеличивать производительность труда). В основе лежит алгоритмическая декомпозиция.

Алгоритмическая декомпозиция - идея разделения задачи по подзадачи по действию ("Что нужно делать?").

Этапы разработки ПО в структурном программировании:

- Анализ (оценка задачи, переработка технического задания)
- Проектирование (разработка алгоритмов)
- Кодирование
- Тестирование

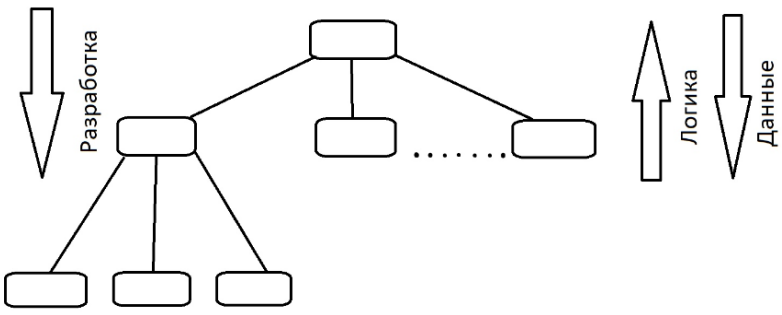
Три технологии структурного программирования:

1. Нисходящая разработка.
2. Использование базовых логических структур.
3. Сквозной структурный контроль.

Нисходящая разработка

В нисходящей разработке используются алгоритмы декомпозиции – разбиение задачи на подзадачи (из принципа "разделяй и властвуй"). Выделенные подзадачи разбиваются дальше на подзадачи. Таким образом, формируется иерархическая структура: данные нисходящие, логика восходящая, разработка нисходящая.

Нисходящая разработка используется в трех этапах разработки (проектировании, кодировании, тестировании).



Идея черного ящика, в том, что, выделив действия - есть какой-то функционал, есть что-то на входе и на выходе, нас не интересует как будет реализовано, а что делает.

После выделения подзадач можно разработать алгоритм основной задачи (планировщик), используя подзадачи.

Правила структурного программирования:

1. Данные на низком уровне, на высшем логика.
2. Для каждой полученной подзадачи создаем отладочный модуль. Готовятся тестирующие пакеты (до этапа кодирования). Принцип полного недоверия к данным.
3. Возврат результата наверх и анализ последующего результата там.
4. Явная передача данных через список параметров (не более 3х).
5. Функция может возвращать не более одного параметра. Не более 7 подзадач у задачи.
6. Глубина вложенности конструкций - не больше трёх.
7. Иерархия уровня абстракции должна соответствовать иерархии данных [Нельзя работать с полями полей структур].
8. Чем больше уровней абстракции, тем лучше.

Принципы работы с кодом:

1. Сегментирование (функция разбивается на логические куски).
2. Пошаговая реализация.
3. Вложенные конструкции (глубина не более 3х).

Заглушка - то, что должна выдавать функция при данных входных данных.

Написание кода программы с использованием заглушек:

Строится диаграмма иерархии алгоритма. Затем пишется код основной программы, в котором, вместо каждого блока диаграммы вставляется вызов подпрограммы, которая будет выполнять этот фрагмент. Вместо настоящих, работающих подпрограмм, в программу вставляются заглушки. Программа тестируется, после успешных тестов начинается написание подпрограмм с заглушками и их тестирование. На каждой стадии процесса реализации уже созданная программа должна правильно работать по отношению к более низкому уровню. Полученная программа проверяется и отлаживается. Разработка заканчивается тогда, когда не останется ни одной заглушки. Такая последовательность гарантирует, что на каждом этапе разработки программист одновременно имеет дело с обозримым и понятным ему множеством фрагментов, и может быть уверен, что общая структура всех более высоких уровней программы верна.

Использование базовых логических структур

IBM решили стандартизировать конструкцию языка, свести количество этих конструкций к достаточному минимуму и этот набор был бы удобен написание программы.

Базовые управляющие конструкции:

- Последовательность (однократное выполнение операций в том порядке, в котором они записаны)
- Ветвление (однократное выполнение одной из двух или более операций, в зависимости от выполнения заданного условия) [if, switch]
- Повторение (многократное исполнение одной и той же операции до тех пор, пока выполняется условие продолжения цикла) [while, until, for, loop - безусловный цикл]

Конструкции ветвления:

Конструкция из двух альтернатив if else и много switch, ограничив глубину вложенности не больше 3. В конструкциях проверка первых случаях должна быть короткой, а последних больше, то есть если проверка при if else, то в else кладем основной случай обработки.

Конструкции повторений:

- Цикл while с предусловием
- Цикл do while (untill) с постусловием
- Цикл for с итерации
- Новые языки for each (работа с конструкциями)

Принципы структурного программирования:

- Выход из цикла должен быть один.
- Не использовать оператора безусловного перехода goto.
- Любая программа строится из трёх базовых управляющих конструкций: последовательность, ветвление, цикл.
- В программе базовые управляющие конструкции могут быть вложены друг в друга произвольным образом.
- Повторяющиеся фрагменты программы можно оформить в виде подпрограмм (процедур и функций).
- Все перечисленные конструкции должны иметь один вход и один выход.

Сквозной структурный контроль

IBM предложила идею организации контрольных сессий.

Суть контрольных сессий: перенос контролирующей функции на плечо самих программистов, чтобы они контролировали друг друга. Сколько было контрольных сессий - не влияет на оценку работы. А сколько недочетов найдут в коде - влияет. Участвуют только программисты (без руководства).

Есть проект, количество программистов в проекте, сталкиваемся с ситуацией управление, разделяя их на группы на 7 человек, где руководитель (озадачить и проконтролировать их, в создании не участвует в таком случаи, так как технологии развиваются, а руководитель нет, то ответственность и контроль переложить на программистов)

Преимущества этой технологии:

- Самые серьезные логические ошибки исправляются на ранних стадиях разработки.
- Объединение этапов - проектирования, кодирования, тестирования (примечание в ООП будут проблемы)
- Взаимодействие с заказчиком на ранней стадии
- Легко распределяются задачи между программистами
- При таком подходе нет "кода в корзину".
- Начиная с самых ранних стадий идет взаимодействие с заказчиком.
- Объединение этапов кодирования, проектирования и тестирования (параллельно происходит).
- Комплексная отладка - тесты пишутся до этапа проектирования на основе ТЗ.
- Удобное распределение работы между программистами.
- Из-за многоуровневой абстракции возникают естественные контрольные точки за наблюдением за проектом.
- Локализация ошибок. (много уровней абстракции, легко выявить где)
- Вероятность невыполнения проекта сводится к нулю.
- Повторное использование кода, выделяются библиотеки.
- Плавное распределение ресурсов при разработке программного продукта. Нет аврала в конце проекта.
- На начальном этапе используется иерархический подход (на этапе распределения ролей), а потом операционный(разработка).
- Проще писать код, проще читать код

Иерархический - порядок программирования и тестирования модулей определяется их расположением в схеме иерархии

Операционный - модули разрабатываются в порядке их выполнения при запуске готовой программы.

Недостатки этой технологии:

- Сложно модифицировать код:
- Понижение надежности за счет внесения изменений в написанный чужой код (плюс трата времени на разбор чужого кода).
- Изменение данных, следовательно программа сыпется. Возникают моменты, когда легче написать свою программу с нуля.
- Исключительные ситуации обрабатываются вперемешку с логикой кода - это приводит к большому количеству проверок и необходимости "протаскивать" ошибку чрез весь код до того места, где её можно будет обработать.

2. Преимущества и недостатки структурного программирования. Идеи Энтони Хоара. Преимущества и недостатки объектно-ориентированного программирования.

Достоинства структурного программирования:

- Структурное программирование позволяет значительно сократить число вариантов построения программы по одной и той же спецификации, что значительно снижает сложность программы и, что ещё важнее, облегчает понимание её другими разработчиками.
- В структурированных программах логически связанные операторы находятся визуально ближе, а слабо связанные - дальше, что позволяет обходиться без блок-схем и других графических форм изображения алгоритмов (по сути, сама программа является собственной блок-схемой).
- Сильно упрощается процесс тестирования и отладки структурированных программ.
- Ясность и удобочитаемость программ

Недостатки структурного программирования:

- Локальные модификации могут нарушить работоспособность всей системы.
- Далеко не все задачи поддаются алгоритмическому описанию и тем более алгоритмической декомпозиции, как того требует структурное программирование.
- Относительно неэффективное использование памяти и увеличение времени выполнения
- Недостаток управляющих структур для реализации этой концепции в языках программирования

Чарльз Энтони Хоар в 1966 выделил базовые понятия ООП:

1. Если в программе меняется данное, то в структурном подходе мы вынуждены выявить все места, где оно используется и внести изменения в написанный код. Вот прикиньте, какой это объем работы? В ООП мы выносим действие над данным! Это не связка с другими данными!
   Эта идея называется инкапсуляцией. Инкапсуляция - объединение данных и действий над данными. Работа с данными идет только через действие. Пример: работа со структурой File в Qt и других программах. Поля разные, а функции одинаковые.
2. Изменение кода - понижение надежности. Для модификации программных сущностей использовать при возможности надстройку над существующими сущностями. Это наследование - надстройка над тем, что существует (возможность сокрытия/добавления полей/функций). Хоар не выделял это как требование, оно утвердилось позже.
   Безразличие к тому, что представляет функционал - полиморфизм. Полиморфизм — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Формирование понятия: данные + действия. Возможность изменения как самих данных, так и надстройки без изменения самих данных.

3. Перенос взаимодействия из физического мира в программу. Синхронное (access-орное) взаимодействие или асинхронное (событийное) взаимодействие. Синхронное взаимодействие - вызов метода (если объект), функции. Асинхронное взаимодействие - событие произошло, изменение объекта или его состояния произойдут со временем, на уровне ЯП реализовать невозможно. Поддержка должна быть на другом уровне. (Реализация на callback вызове).

Сначала начали реализовывать операционные оболочки. Сейчас это поддерживается на уровне ОС.

Преимущества ООП

- Все недостатки пусты перед тем, что мы можем легко модифицировать программу.
- Более гибкая система, код легче поддерживать -> возможность легкой модификации (при грамотном анализе и проектировании) -> увеличивается показатель повторного использования кода.
- Более "естественная" декомпозиция ПО, которая существенно облегчает разработку.
- Сокращение количества межмодульных вызовов и уменьшение объемов информации, передаваемой между модулями.

Недостатки ООП

- Невозможность совмещения этапов проектирования, кодирования, тестирования (что в структурном возможно). Сначала нужно построить начальную модель, потом рекурсивный дизайн, то есть эту модель мы можем развивать.
- В структурном подходе (СП) порог вхождении очень низкий, то есть вы сразу может начать писать программу используя СП. А что касается ООП порог вхождения выше и это проблема, требуется более высокая классификация программиста.
- Программист и проектировщик, и кодировщик. Другие требования к квалификации.
- Размер кода резко возрастает - резко увеличивается время выполнения, объем необходимой памяти увеличивается. Гораздо больше времени тратится на проектирование. В структурном подходе лавинный вызов метода.
- В СП легко контролировать ресурсы и правила черного ящика (если какая-то функция не может выполниться она не должна захватывать ресурс, при возвращении ресурса ответственность переходит на вызывающую программу). Что касается ООП - задача разбивается на куски, например, если проблема с утечкой памяти, ее определили и выявили место, где ошибка, то не всегда можно избавиться от утечки памяти. Раньше в ООП программах были с утечками, сейчас более-менее нормально. В принципе с C11 есть механизмы борьбы с проблемами памятью - висящие указатели.
- Возникновение "мертвого кода" при модификации. Это занимает место, висит в памяти. Борьба с мертвым кодом - компилируется только тот код, который вызывается.

3. Основные понятия ООП: инкапсуляция, наследование, полиморфизм. Понятие объекта. Категории объектов. Отношения между объектами. Понятие класса. Отношения между классами. Понятие домена.

Инкапсуляция – объединение данных и действий над ними; взаимодействие с ними только за счет действий. Не работаем непосредственно с полями данных.

Наследование – модификация за счет «надстройки», не изменяя написанный код.

Полиморфизм – возможность подменять одно другим, не изменяя написанный код. Возможность обработки разных типов данных, с помощью "одной и той же" функции, или метода.

Объект – конкретная реализация абстрактного понятия, обладающая характеристиками состояния, поведения и индивидуальности.

Модель состояний (объектов) Мура:

- Множество состояний
- Множество событий, которые приводят к изменению состояний объекта
- Правила перехода
- Действия состояний: перевод объектов в новые состояния

Категории объектов:

- Реальный объект. Абстракция фактического существования предметов в физическом мире.
- Роль. Абстракция цели или назначения человека, части оборудования и т.д. (пример: студент, сливной бачок, член семьи)
- Инцидент. Абстракция чего-либо происшедшего либо случившегося (наводнение, скачок напряжения, выборы).
- Объекты взаимодействия. Получаются при взаимоотношении одних объектов с другими (перекресток, порог, долг).
- Объекты спецификации (пассивные объекты). Стандарты, правила, критерии (ПДД, расписание движения поездов).

Отношения между объектами:

- Отношение использования (старшинства). Роли:
  - Активные объекты (объекты воздействия). Не подвержены воздействию других объектов.
  - Пассивные объекты.
  - Посредники. Могут быть активными и пассивными.
- Отношение включения. Абстрагирует от внутренней реализации объектов.

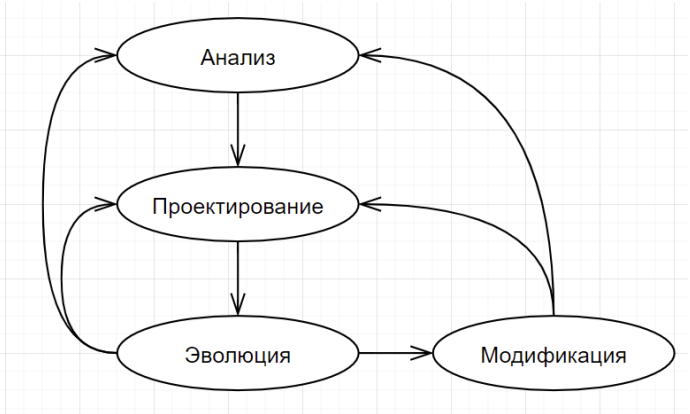
Класс – такая абстракция множества предметов реального мира, что все предметы в этом множестве обладают одними и теми же характеристиками. Все объекты подчинены и согласовываются с одним и тем же набором правил или/и поведений.

Отношения между классами:

- Наследование.
- Использование.
- Схема наполнения. Один класс содержит другой класс.

Домен – отдельный или реальный, гипотетический или абстрактный мир, населенный отчетливым набором объектов, которые ведут себя в соответствии с характерными для домена правилами и линиями поведения.

4. Цикл разработки ПО с использованием ООП: анализ, проектирование, эволюция, модификация. Рабочие продукты объектно-ориентированного анализа.



Анализ.

Этап анализа проходит вместе с заказчиком. Для нас, как программистов, это плюс - мы четко понимаем, что представляет из себя задача. Для заказчика это тоже плюс, так как на этом этапе он сам разбирается в модели той задачи, которую мы решаем.

Анализ строится на основе физического мира. Мы строим модель и как бы делаем "слепок" физического мира, то есть строить объектно-ориентированную модель, выделять сущности, выявлять взаимодействие сущностей.

Требования к модели:

- Модель должна быть полная.
- Модель должна быть понятная всем заинтересованным лицам.

Проектирование.

Этап проектирования - очень важный этап, так как нам нужна документация к нашему программному продукту.

Проектирование - перевод из документов, которые нам помогали проводить анализ, в проектные документы, на основе которых мы уже непосредственно пишем код.

Эволюция

В ООП эволюция включает в себя: кодирование, тестирование, рекурсивный дизайн.

Мы начинаем с простого, начального состояния модели нашей системы и потом её развиваем. Если у нас возникли какие-то проблемы, мы переходим или к анализу, либо перепроектируем какие-то части системы.

Преимущества эволюции:

- Предоставляется обширная обратная связь.
- Появляются различные версии нашей системы. Быстро и легко мы можем перейти от одного решения к другому.
- За счет большого количества версий системы появляется возможность плотно взаимодействовать с заказчиком.
- Как в структурном программировании, мы разворачиваем систему и уже на начальных этапах видны результаты.
- Интерфейс проектируется отдельно, до основного этапа разработки, так как на интерфейсе отрабатывается взаимодействие с заказчиком, как с пользователем системы.

Какие изменения могут быть реализованы на этом этапе:

- Добавление нового класса - вещь совершенно безболезненная, которая при нормальном анализе не приводит к изменению написанного кода.
- Изменения реализации класса. Это происходит на основе наследования, мы добавляем новый класс, подменяющий другой класс. Мы не переписываем существующий код.
- Добавление нового класса - имеется ввиду добавление своей ветви. Реализация класса - на основе наследования.
- Изменение представления класса. Мы можем с классом работать по-другому и это не должно повлиять на переписывание кода.
- Реорганизация структуры классов - болезненная часть, затрагивающая минимальное переписывание (использование паттернов, например, появление компоновщика, адаптера для расширения функционала, прокси для добавления функционала или реализации доступа к объекту).
- Самое страшное - изменение интерфейса базового класса. До такой ситуации лучше не доводить, как правило, это переделывание большого кода.

Модификация

Эволюция - непосредственно разработка программного продукта. Мы сдали заказчику готовый продукт, а следующие изменения, когда мы уже сдали, будет модификацией. Модификация затрагивает анализ и проектирование.

Разница между эволюцией и модификацией:

- До этапа сдачи - эволюция, после - модификация.
- Эти этапы выполняют совершенно разные люди.

5. Концепции информационного моделирования. Понятие атрибута. Типы атрибутов. Правила атрибутов. Понятие связи. Типы связей. Формализация связей. Композиция связей. Подтипы и супертипы. Диаграмма сущность-связь.

Объектно-ориентированный подход - подход белого ящика. Мы идем от сущности, которая у нас реально существует, и из чего состоят эти сущности. Начинается всё с информационного моделирования.

Всегда разработка начинается с физических сущностей, которые существуют. Во есть задача, рассматривается задача на примерах с физическими сущностями. Пытаемся выделить характеристики этих объектов

У нас есть объект. Мы выделяем, что характеризует этот объект. Естественно, это зависит от той задачи, которая нам требуется. Мы подходим не "что нам делать", а "чем характеризуется". Мы выделяем атрибуты объектов, характеристики. Любая характеристика, которая нас заинтересовала, абстрагируется как атрибут.

Атрибуты делятся на:

- Идентифицирующие или указывающие. Идентификатор - набор из одного или нескольких атрибутов, которые четко идентифицируют объект. Атрибуты, которые мы объединили в понятии идентификатор, являются идентифицирующими. Еще их называют указывающими атрибутами.
- Описательные. Любая характеристика объекта, которая его описывает.
- Вспомогательные. Мы их вводим для формализации связей и состояния (статус).

Для каждого атрибута мы смотрим, какие значения он может принимать, то есть определяем его множество значений. Это нужно для того, чтобы в дальнейшем понять, какого типа этот атрибут.

Мы выделяем правила атрибутов:

1. Для данного объекта каждый атрибут в любой момент времени должен иметь значение. Не может быть такого, чтобы какой-то атрибут не был определен. Это значение должно быть единственным. Если, когда мы выделили какую-то сущность, в ней появился объект, который мы отнесли к этой сущности, но у него, возможно, какой-то атрибут не определен — значит, объект относится к другой сущности.
2. Атрибут не должен содержать внутренней структуры. Он не должен быть для нас сложным. Это что-то простое, например: вес, рост и так далее.
3. Когда атрибут имеет составной идентификатор, каждый атрибут, который является частью идентификатора, представляет характеристику всего объекта, а не его части, ну и тем более не характеристику чего-либо другого. Например, для студента, ФИО — это всё характеристики самого объекта, самого студента, а не его части.
4. Если атрибут не является частью идентификатора, то он должен представлять характеристику данного объекта, указанного идентификатором, а не характеристику какого-либо другого атрибута. Это типичная ошибка - добавить атрибуты, которые характеризуют другие атрибуты или части объекта.

Между сущностями существуют связи, и эту связь мы должны задать из перспективы каждого участвующего объекта. Связи могут быть разными.

Типы связей:

- Зависимость. Изменение одной сущности может влиять на другую.
- Ассоциация. Позволяет перейти от одной сущности к другой (студент-вуз).
    - Агрегация. У сущностей не связаны жизненные циклы.
    - Композиция. Связаны жизненные циклы (человек-органы).
- Наследование.
    - Реализация. Может расширять базовый класс.
    - Обобщение. Полиморфизм.

Мы должны формализовать связь. У нас есть связь, и кто-то за эту связь должен отвечать.

На информационной модели некоторые связи могут быть следствием других связей.

Есть связь студента с кафедрой, и есть связь студента с руководителем. Получились такие связи, когда кафедра может на студента через преподавателя и на прямую. Когда получилась такая связь, мы одну из связей формализуем как композицию других. Избыточностей связей не должно быть.

R3 = R1 + R2 - композиция других связей.

Супер-класс - абстрактное понятие на этапе информационного моделирования. Необходимо выделять общие атрибуты разным сущностям в супер-класс.

Возможна такая ситуация, когда объект мы отнесли к какому-то классу, но для него этого атрибута нет. Или мы отнесли два объекта к разным сущностям, разным классам, но они имеют общие атрибуты. В этом случае мы должны объединять эти классы так называемым супер-классом. Атрибуты, являющиеся общими для разных классов, нужно выносить в этот суперкласс, а все остальные атрибуты, которыми отличаются объекты, будут в подклассах.

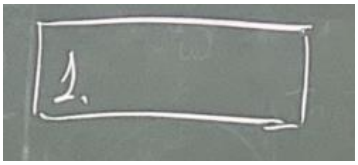
6. Модель поведения объектов. Жизненный цикл и диаграмма перехода в состояния (ДПС). Виды состояний. События, данные событий. Действия состояний. Таблица перехода в состояния (ТПС). Правила переходов.

Для описания поведения объектов мы используем модель Мура.

Она включает в себя:

- множество состояний объекта (стадий)
- множество инцидентов, которые переводят объект из одной стадии в другую (событий)
- множества действий, которые мы связываем с состоянием
- правила перехода

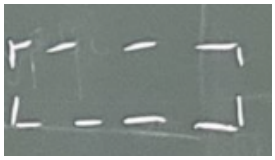
ДПС - диаграмма переходов состояний.



Состояние обозначаются прямоугольником. Каждому состоянию мы присваиваем уникальный номер и имя состояния.

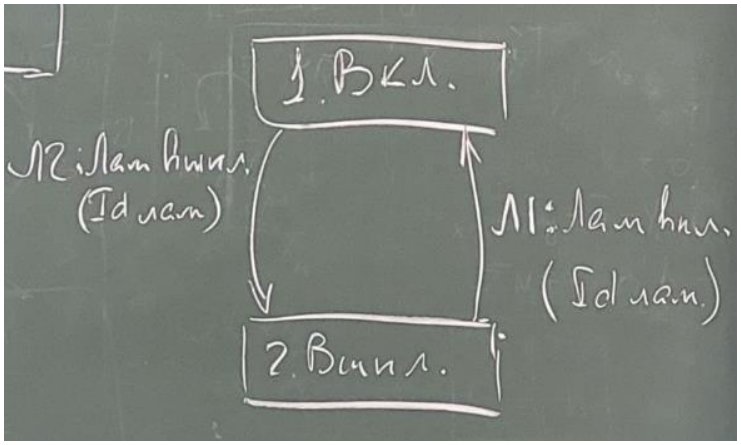
Виды состояния:

- Состояние создания - такие состояния, в которых объект появляется первый раз. Переход в такие состояния переходит не из состояния
- Заключительное состояние. Возможно два варианта заключительных состояний:
  - Объект уничтожается. Это состояние на диаграмме рисуется пунктирной линией.
  - Состояние, из которых объект уже не переходит в другие состояния. Дальнейшего поведения у объекта нет, но он не уничтожается! На диаграмме оно рисуется таким же прямоугольником, но перехода из него в другое нет.



- Текущее состояние. Чтобы определять текущее состояние, мы в класс добавляем вспомогательный атрибут, определяющий это состояние. Его называют статусом.

Пример лампочки:



Имеет два состояния: выкл. и вкл. Чтобы объект перешел из одного состояние в другое объект должно произойти событие, как внутренне, так и внешнее. Событие переносят данные. Обязательно идентификатор объекта, кроме события создания.

Из состояния выкл переходим в вкл, событие Л1 => лампочка включается. Л2: лампочка выключается.

Описание события:

- Значение события - короткое событие, отвечающее на вопрос: "Что происходит?".
- Предназначение события. Мы четко событие связываем с каким-либо объектом.
- Метка события (номер).
- Данные события. Событие может переносить данные. Данные могут быть причем двух типов: идентифицирующие данные и любые другие атрибуты, причем не обязательно только того объекта, для которого происходят события.

Правила связи события с состоянием

- Все события, которые переводят объект в определенное состояние, должны нести одни и те же данные. С каждым состоянием мы связываем действие, которое должно выполняться при переходе объекта в состояние.
- Правило состояния создания: то событие, которое переводит в состояние создания, не несет идентификатора объекта.
- Правило состояния не создания: оно должно нести идентификатор объекта.

Каждому состоянию мы ставим в соответствие действие. Задача действия - перевести объект в то состояние, которому оно соответствует. Оно может выполнять любые операции над атрибутами самого объекта. Кроме того, выполнять любые вычисления. Действие может порождать события для любого объекта любого класса, в том числе и для самого себя. В том числе порождать события для чего-либо вне области анализа нашей подсистемы. выполнять любые действия над таймером: выполнять создавать считывать, запускать таймер, очищать таймер.

Мы не накладываем ограничений. Действие имеет доступ к любым атрибутам объектов любых классов. На этом этапе мы не ограничиваем действие! Ограничения возникнут позже.

Таблица переходов состояний

С помощью ТПС мы контролируем, какие возможны переходы из одного состояния в другое. В ТПС каждая строка — это состояние, в котором может находиться объект, столбец — это событие. Состояния нужно не забыть пронумеровать, у каждого состояния есть свой уникальный номер.

Варианты заполнения ячеек:

- В какое состояние объект перейдет в результате возникновения события в этом состоянии? В ячейке пишем номер состояния, в которое переходит объект.
- У нас может быть ситуация, когда событие игнорируется, то есть в этом состоянии объект игнорирует событие. Событие происходит - никакого перехода не происходит. В этом случае мы ставим прочерк.
- Еще одно возможное заполнение ячейки (лучше, чтобы их не было) - данное событие не может произойти, если объект находится в этом состоянии. Ставим крестик (не плюсик).

Анализ по таблице переходов состояния - все ячейки должны быть заполнены.

Идея - избавиться от крестиков. В таблицу добавлять новые состояния, соответствующие этому атипичному поведению. Все атипичные ситуации должны обрабатываться точно также, как типичное поведение.

7. Модель взаимодействия объектов (МВО). Диаграмма взаимодействия объектов в подсистеме. Типы событий. Схемы управления. Имитирование. Каналы управления.

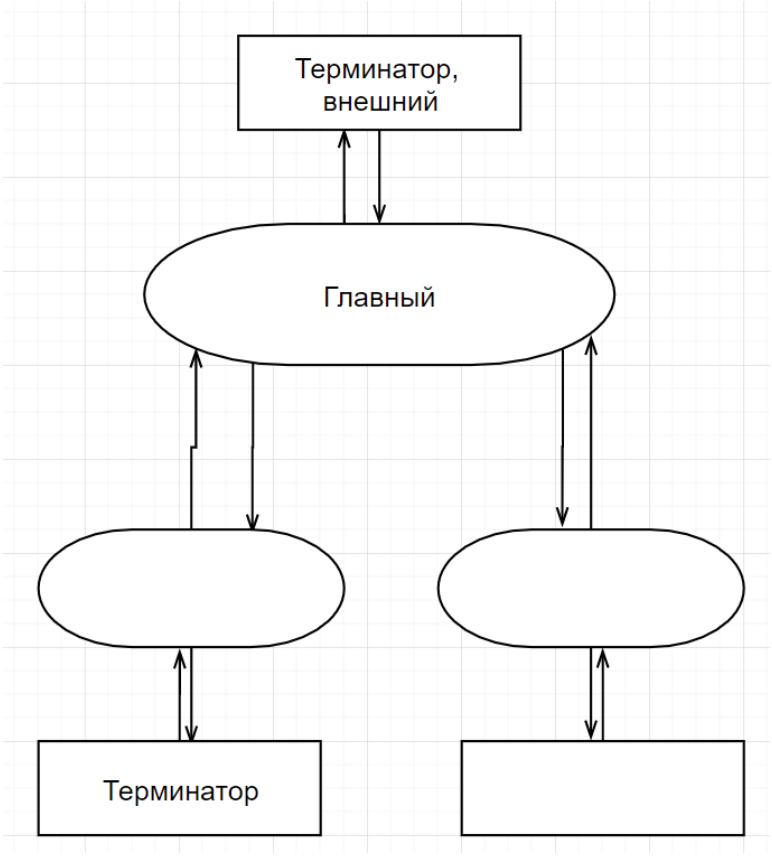
Есть объект А, объект В, рисуем стрелку А -> В, что говорит о том, что А использует аксессор объекта В, на стрелке помечаем идентификатор процесса, где взаимодействие синхронное. МВО - модель взаимодействия объектов. Мы рассматриваем только нашу подсистему или домен. Что ВОВНЕ нас не интересует, но ИЗВНЕ могут приходить события.

МВО строится таким образом, что вверху располагаются модели состояний объектов, которые боле осведомлены о нашей подсистеме. Идея - свести так, чтобы главный был один.

Все события разделяем на две группы:

- Внешние события, которые приходят извне.
- Внутренние события, которые происходят внутри нашей подсистемы.

На МВО мы ограничиваем нашу подсистему сверху и снизу. Ограничения сверху и снизу называют терминаторами. Соответственно, могут быть терминаторы верхнего уровня и терминаторы нижнего уровня (прямоугольничек).



Все модели состояний, чтобы как-то их отделить от сущности, рисуются в овалах (вытянутых овальчиках). Внутри записывается номер и указывается, какой сущности соответствует этот объект.

Приходит событие извне и объект наш может менять состояние. Мы должны четко проанализировать, какие события принимает наша модель состояния, и какие события она может порождать для других объектов. Причем эти события могут быть как вверх, так вниз. События могут уходить к терминатору.

Внешние события разделяем на две группы:

- Незапрашиваемые - не являются следствием предыдущего действия нашей подсистемы.
- Запрашиваемые - являются результатом действия нашей подсистемы.

В зависимости от того, откуда пришло не запрашиваемое событие, рассматриваем схемы:

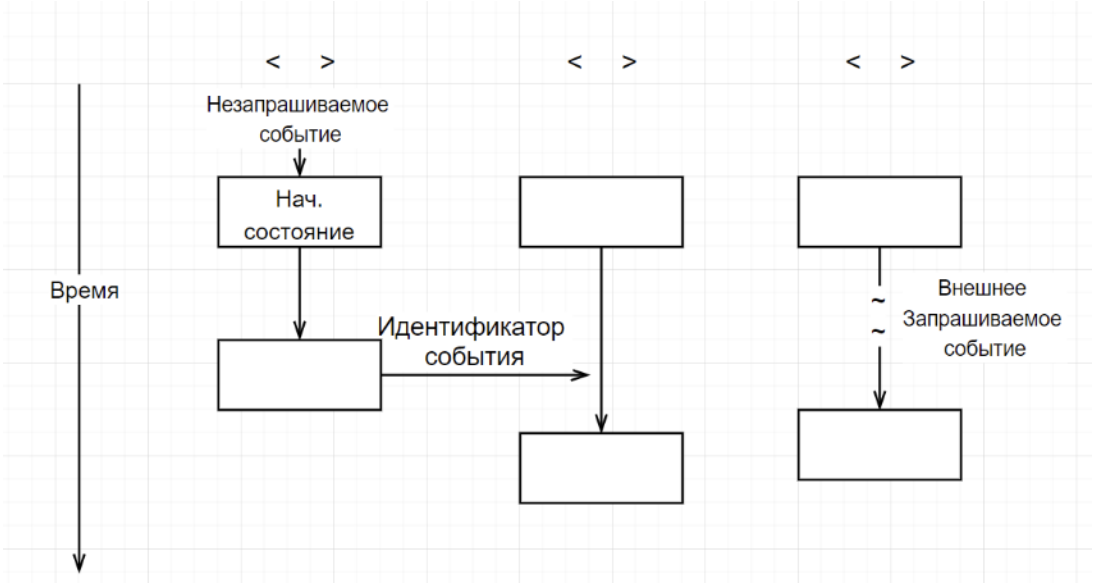
- Схема верхнего управления - не запрашиваемое событие пришло от терминатора верхнего уровня.
- Схема нижнего управления - не запрашиваемое событие пришло от терминатора нижнего уровня.

Мы нарисовали модель взаимодействия объектов в нашей подсистеме, но хотелось бы проследить, как изменяются состояния объектов нашей подсистемы с приходом какого-либо не запрашиваемого события извне. Для того чтобы проверить корректность нашей модели, мы должны перебрать все возможные варианты.

Для удобства моделирования строятся каналы управления.

МДО - взаимодействие синхронное, а в МВО - асинхронное. Выделили модели состояний, модель взаимодействия, но нам надо проверить все же нормально, для этого проводится процесс проектирования результатов, строятся КУ, как у нас происходит процесс при незапрашиваемое событие, причем для каждого событие строится свои каналы управления. Дело в том, что у нас может быть несколько начальных состояний нашей подсистемы. Первым делом, что мы должны сделать - выделить в каких начальных состояний могут находится наши подсистемы, далее смотрим какие незапрашиваемые событие могут приходить, неважно от каких терминаторов и смотрим как система реагирует на эти событие.

Также могут быть и запрашиваемые событие, они также включаются в КУ.



Правила построения КУ:

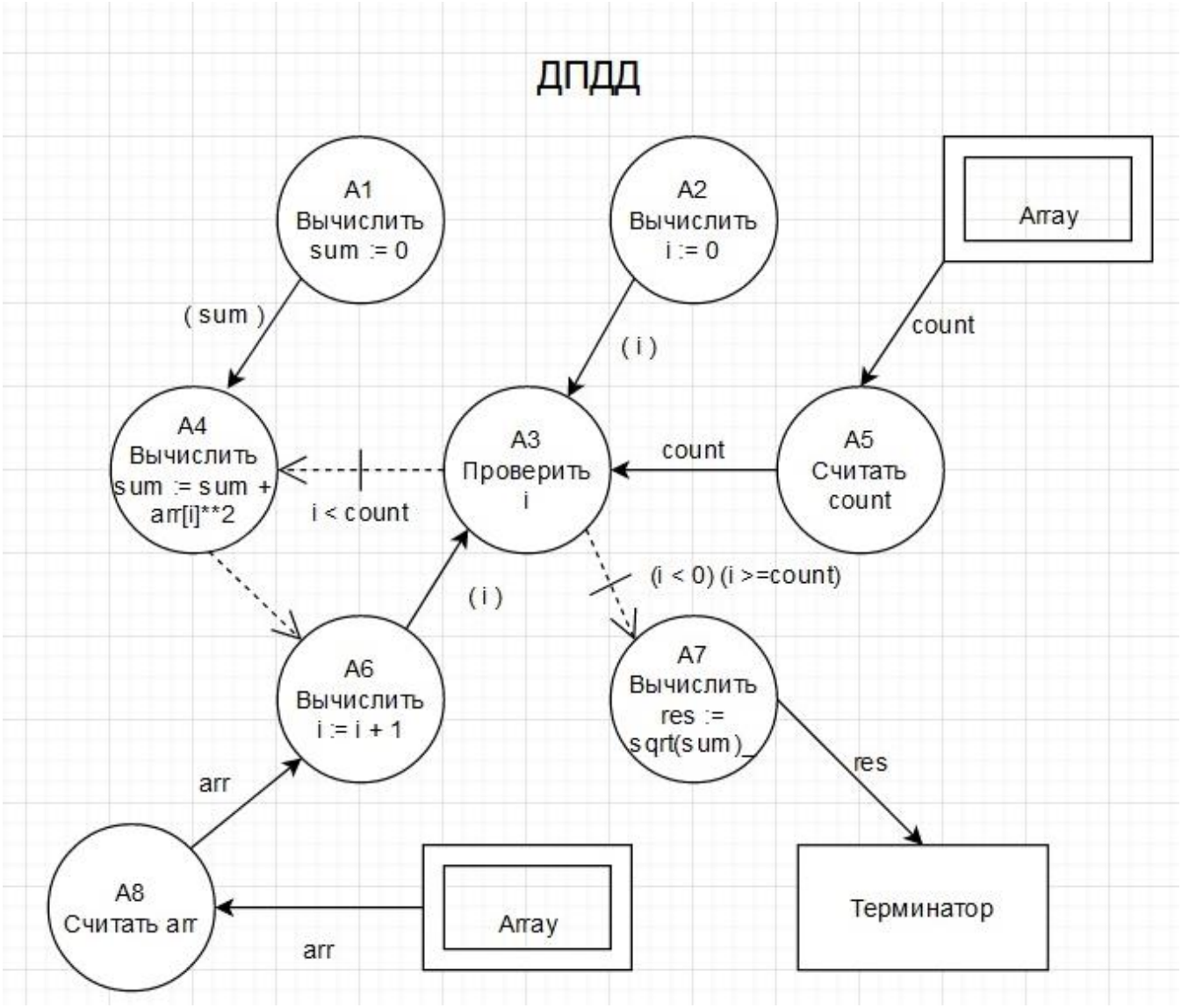
- КУ должны иметь конец.
- Как правило при построении выделяют сначала главную модель состояний, и затем уже остальные. Тут также неважно будут реагировать или изменять модель состояний, которые мы просматриваем на данном этапе, все равно выделяем все состояния нашей подсистемы.
- Название состояний, берется с ДПС.
- Всегда помечаем, какое незапрашиваемое событие пришло, в результате которого модель состояния может перейти в другое
- При моделировании, тут будет важно время, временная школа сверху вниз
- Для каждого состояния выделяются два времени (не указываем, если несущественное):
    - Время выполнения действия - время, которое переводит объект в это состояние. Это время записывается внутри прямоугольника.
    - Время задержки - это то время, которое объект должен находиться в этом состоянии. Например, для дверей есть состояние "двери закрываются". Закрытие дверей не происходит мгновенно, это происходит за какое-то время. Это время записывается вне прямоугольника.
    -

Процесс имитирования каналов управления:

- Сгенерировать все начальные состояния.
- Для каждой генерации принять все незапрашиваемые события.
- Мы должны прийти в какие-то окончательные состояния.

8. Диаграмма потоков данных действий (ДПДД). Типы процессов: аксессоры, генераторы событий, преобразования, проверки. Таблица процессов (ТП). Модель доступа к объектам (МДО).

Мы разбиваем наш алгоритм на процессы, которые происходят для выполнения обработчика состояний. Диаграмма строится для каждого состояния, каждой модели состояний. И каждый из них отвечает на вопрос, что он делает. Каждое действие можно выделить как процесс, который происходит.



Виды поток данных ДПДД:

- Условные (штрих-пунктирная с обозначением условия перехода);
- С передачей данных (прямые);
- Безусловные (штрих-пунктирная без обозначения условия);

Выделяют 4 типа процессов

- Аксессоры – единственная цель – получить доступ к архиву данных 4 вида:
  - Создания/Уничтожения
  - Чтения/Записи
- Генераторы событий – процесс, который создает на выходе событие:
  - Принять событие
  - Породить событие
- Преобразования/Вычисления – процесс, который выполняет какие-либо преобразования данных
- Проверки – результат – условный переход

Мы объединяем в таблицу процессов состояний (выделяем действия и выносим их для общего использования). Возможно несколько представлений этой таблицы. Сначала мы разбиваем эту таблицу по дпдд: “на данном дпдд происходят такие процессы, другая дпдд - другие процессы и тд.”. Потом пытаемся проанализировать и объединить все те процессы, которые проходят в одной модели состояния. Чтобы было удобнее, можно преобразовать это по типу действия, которое происходит. То есть можно сгруппировать аксессоры, преобразователи и тд. и посмотреть есть ли что-то общее. Обработав так процессы для каждой модели состояния, мы можем объединить это в одно целое. Так мы четко выделяем общие процессы.

Аксессоры определяются для того объекта, к архиву данных которого происходит доступ. Аксессор создания — это конструктор самого объекта. Аксессор уничтожения - деструктор самого объекта. Аксессоры чтения/записи - методы объекта.

Есть объект A, объект B, рисуем стрелку A -> B, что говорит о том, что A использует аксессор объекта B, на стрелке помечаем идентификатор процесса, где взаимодействие синхронное. Определив таким образом взаимосвязи между объектами, получим модель доступа объеков.

9. Домены. Модели доменного уровня. Типы доменов. Мосты, клиенты, сервера.

Домен - отдельный реальный, абстрактный или гипотетический мир, населенный отчетливым набором объектов, которые ведут себя соответственно определёнными доменом правилам или линиям поведения. Нашу систему мы разбиваем на домены.

Классификация доменов:

- Прикладной домен – непосредственно та задача, которую мы решаем. Его часто называют бизнес-логикой нашей системы.
- Сервисные домены – обеспечивают какие-то сервисные функции.

Например, пользовательский интерфейс. Домен, который реализует взаимодействие пользователя с прикладным доменом, выделяем как сервисный домен. Мы можем читать данные из базы данных из архива — это тоже сервисный домен, то есть то, что необходимо нашей задаче.

- Архитектурный домен – предоставляет общие механизмы и структуры управления данными и в общем всей системы, как единым целым.

Задача, который решает архитектурный домен: придать однородность структуре ПО.

Когда в домене большое количество классов (где-то 50 штук), его разбивают на подсистемы по принципу минимума связей между выделенными подсистемами.

Для домена (или для каждой подсистемы):

- ДСС – диаграмма сущность-связь (информационная модель).
- МВО – модель взаимодействия объектов (событийная модель). Можем реализовать как синхронно, так и асинхронно (нужно стремиться к асинхронной модели.
- МДО – модель доступа к объектам (аксессорная модель). Можем реализовать только синхронно.

Если мы домен разбиваем на подсистемы, нужно для домена реализовать соответствующие диаграммы:

- МСП - модель связи подсистем
- МВП - модель взаимодействия подсистем
- МДП - модель доступа подсистем

Получается такое соотношение:

- ДСС ---> МСП
- МВО ---> МВП
- МДО ---> МДП

Желательно чтобы все эти диаграммы можно было накладывать друг на друга. То есть расположение всех этих подсистем на трех диаграммах одинаковое. Отличаются только связями: МСП - общее представление о связях, МВП - на уровне событий, МДП - аксессерное взаимодействие.

На модели связей подсистем: можно изобразить множество связей, как одну связь, и перечислить все связи, которые входят в это множество. На модели взаимодействия подсистем: необходимо указать все события (стрелочка от порождающего к принимающему). На модели доступа подсистем: -//-, но стрелочки — это не события, а аксессорные процессы между подсистемами.

Внутри прямоугольника, обозначающего подсистему, характеризуем ее: имя, если есть, и диапазон номеров объектов, которые входят в эту подсистему.

Домен, который предоставляет что-то другому домену, называется серверным. О связи между доменами: существует мост между одним доменом и другим. Клиенту не важно, кто предоставляет ему возможности, которые он использует. Серверу не важно, кто использует возможности, которые он предоставляет.

10. Объектно-ориентированное проектирование. Диаграмма класса. Структура класса. Диаграмма зависимостей. Диаграмма наследования.

Используется язык объектно-ориентированного проектирования (Object-Oriented design language - OODLE)

В этой нотации выделяется 4 диаграммы:

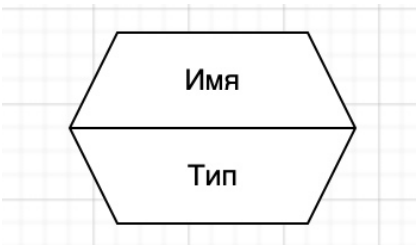
- Диаграмма класса - диаграмма внешнего представления одного класса
- Схема структуры класса – чтоб оказать внутреннюю структуру класса и доступа к данным
- Диаграмма зависимостей – показывает дружественные связи и клиент-серверный доступ
- Диаграмма наследования

Диаграмма класса

Мы проектируем класс вокруг данных, атрибутов этого класса. У нас есть информационная модель, в которой мы выделили каждую сущностью, и каждая сущность для нас должна быть классом. У нас есть имя этой сущности(имя класса) , есть данные, есть атрибуты этого класса, которые становятся компонентами или членами класса при проектировании.

Когда мы выделяли атрибуты, мы четко перечисляли значения, которые может принимать атрибут для разных объектов. Это перечисление нам теперь необходимо для того, чтобы понять какого типа будет компонент.

Имя компонента берём то, которое уже было на информационной модели. И на основе тех значений, которые может принимать атрибут, определяем тип компонента. На диаграмме рисуется так:



Так делаем для каждого компонента.

Для каждой сущности мы выделяем идентификатор. Если идентификатор формальный, то на диаграмме классов мы его не записываем, то есть при проектировании мы его убираем. А если это не формальный атрибут, то на диаграмме мы это преобразуем в компонент.

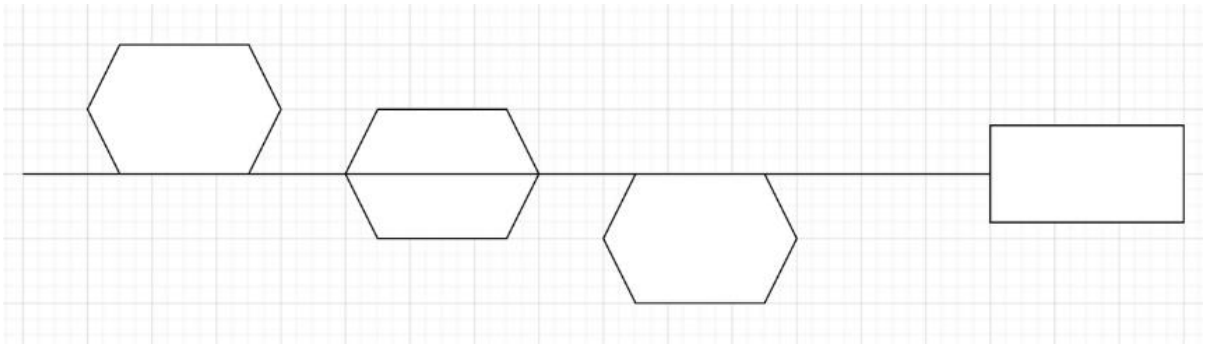
Таким образом мы рассмотрели то, что внутри диаграммы класса.

Теперь наша задача выявить те методы, которые мы можем выполнять над нашим объектом. Стоит сказать, что все методы мы можем разделить на 2 группы:

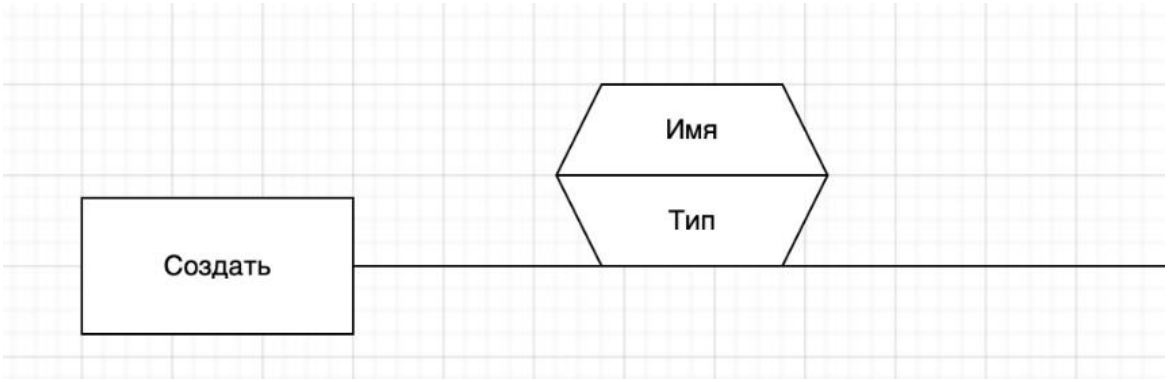
- Операции над объектом
- Операции над классом. С правой стороны рисуются операции над классом, с левой стороны над объектом.

Примером операции над классом является операция “создать”. Дело в том, что, когда мы используем эту операцию, объекта ещё нет. Объект появляется после выполнения этой операции. Это может быть конструктор или какой-то статический метод, который порождает объект данного класса.

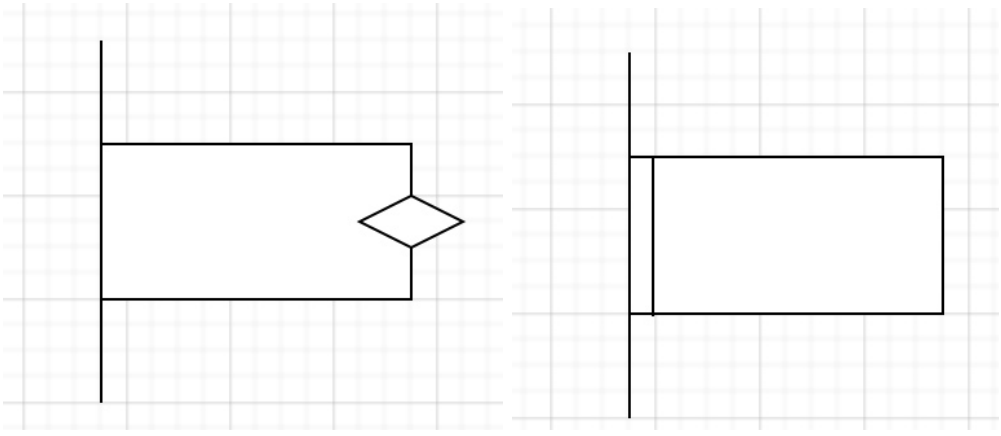
Для операции мы рисуем линию, на которой мы должны показать какие данные получает процесс, какие изменяет, какие возвращают. Над линией мы показываем данные, которые получает. На самой линии изменяемые параметры. Под линией возвращаемые параметры:



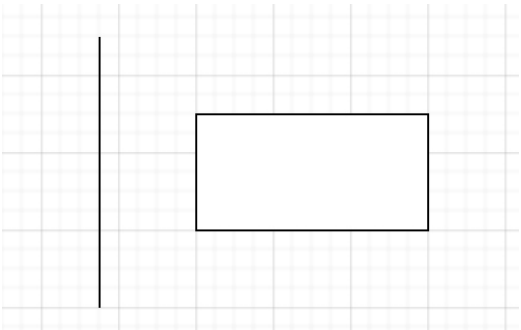
Когда речь идет об операции создания, нас интересуют только получаемые данные. Результатом является объект, но мы это не указываем на диаграмме. Получаемые компоненты мы рисуем аналогично членам самого класса.



Если метод обрабатывает исключительную ситуацию, то мы у него рисуем ромбик, если метод связан и вызывается во время выполнения, мы помечаем его чертой. Это обработчик состояний:



Если метод скрытый, то в принципе его можно помечать на диаграмме, но нас тут интересует интерфейс. Но отметить некоторые операции, которые происходят внутри, можно. В таком случае мы не рассматриваем данные, которые они получают. Если класс перегружен, то можно вообще эти классы не зарисовывать.



Возможно такое, что метод принимает или возвращает несколько данных. Тогда мы рисуем тот же “гробик” и приписываем N:

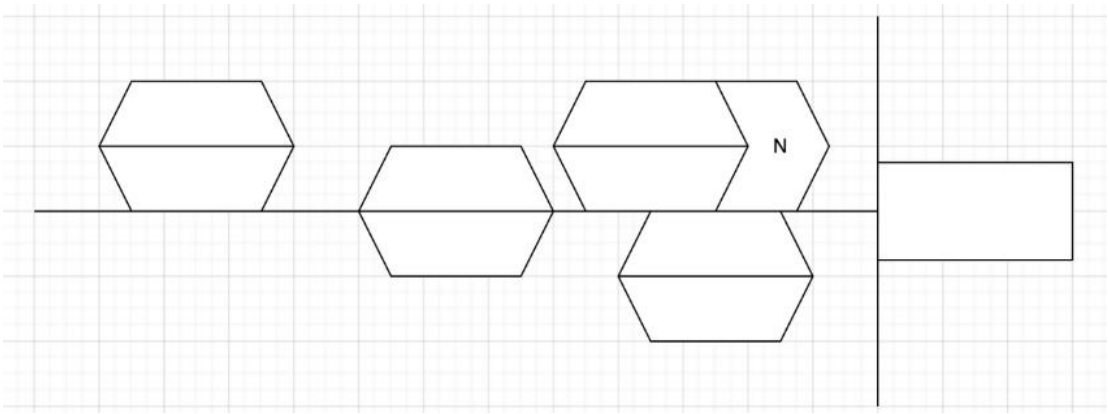
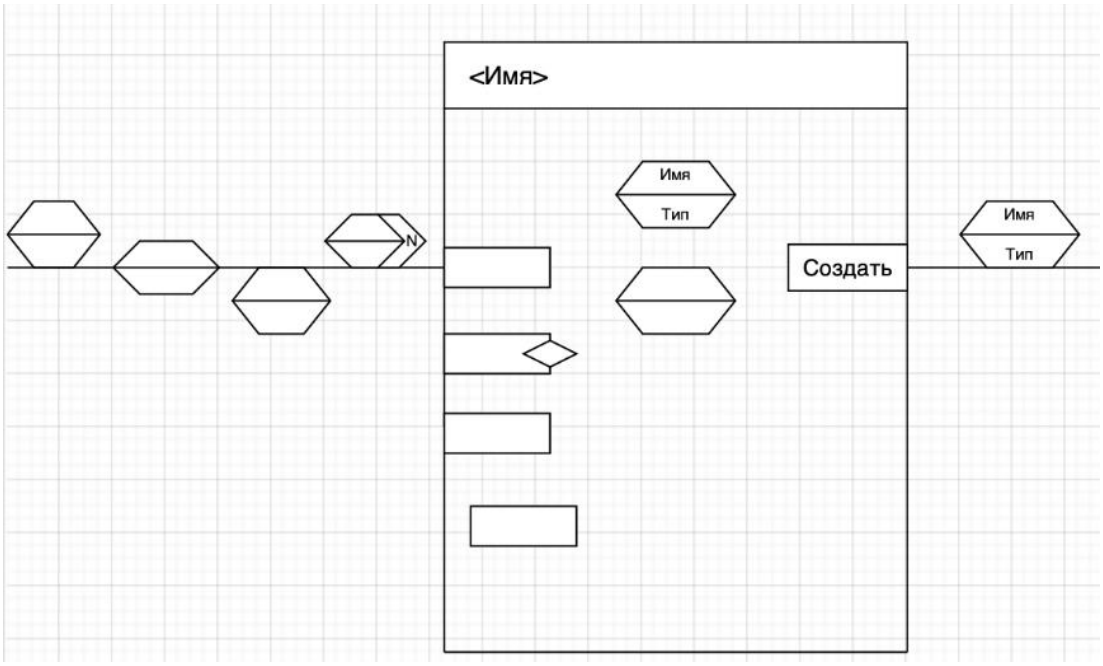
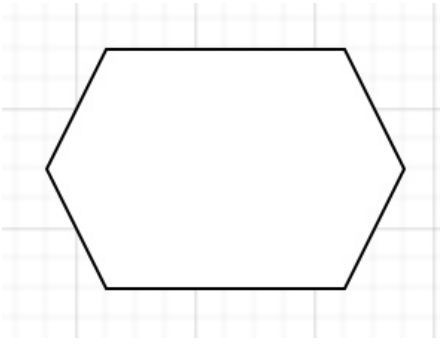


Диаграмма внешнего представления одного класса



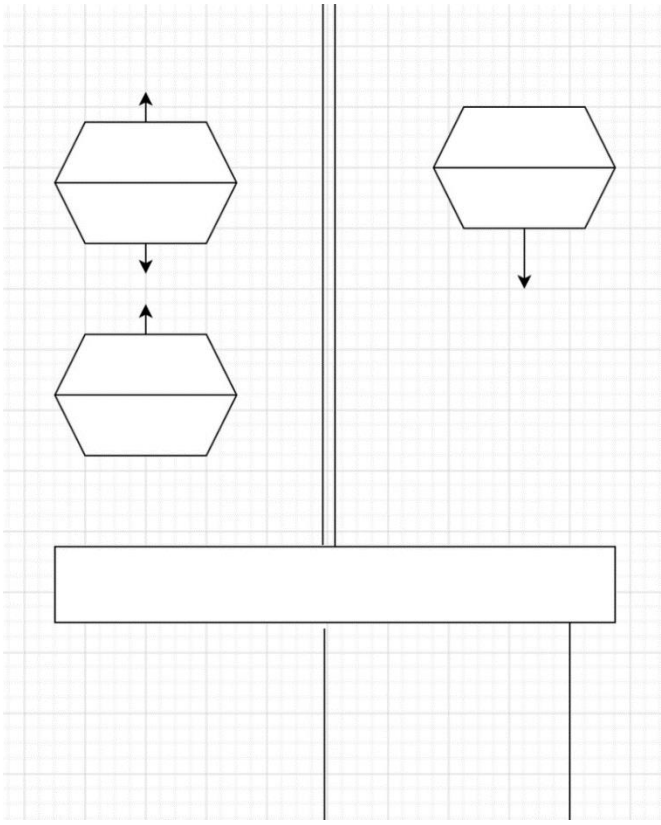
Структура класса.



Наша задача проследить потоки данных и доступ к непосредственно данным самого класса. На данной диаграмме всё строится вокруг данных объектов. Это то, что на дпдд мы называли архивом данных. Вот он:

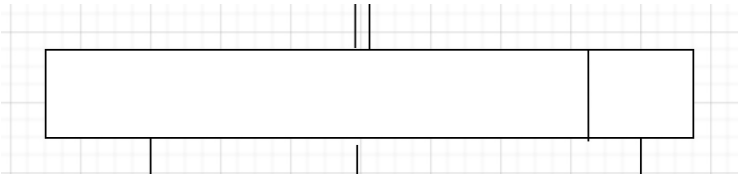
Допустим у нас есть методы. Здесь задача четко посмотреть какие это методы. Нас интересовать будут не только методы, которые получают данные извне, но ещё и методы, которые используют методы, которые вызываются извне. Например, есть три метода.

Мы соединяем их линиями. Линии — это линии по доступу передачи данных. На каждой линии мы показываем какие данные получает метод и какие возвращает. Опять рисуем гробики. Если стрелочка вниз, это параметр, которые принимает метод. Если стрелочка вверх, то возвращает. Если и вверх, и вниз, то и, то, и другое:

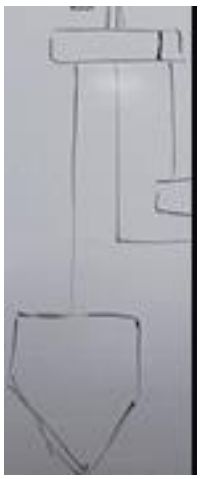


Мы четко выделяем слои. Слой методов, которые вызываются извне – общедоступные, и методы скрытные, которые не вызываются извне. Желательно, чтоб методы интерфейса не вызывали друг друга. Если есть что-то общее в разных методах, то мы выделяем метод скрытого слоя, который объединяет эти методы общедоступного слоя. Четко прослеживаются потоки данных

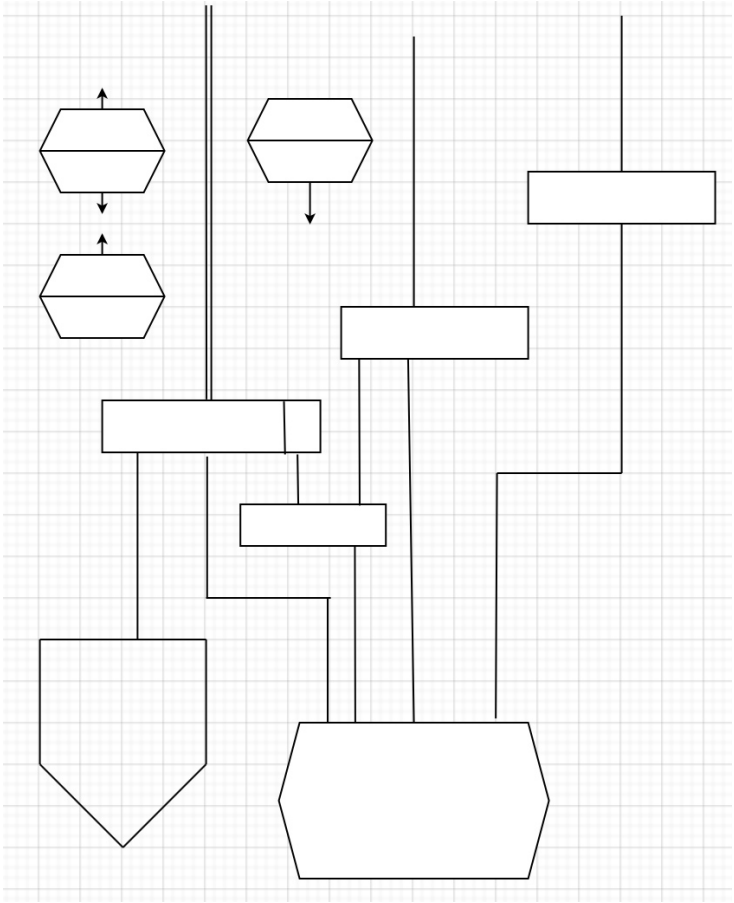
Если метод обрабатывает исключения, выделяем квадратиком:



Если метод, получает или передает как результат какие-то данные во внешний модуль, то мы показываем это таким образом:



Общий вид схемы:



На схеме структуры класса четко прослеживаются потоки данных.

Диаграмма зависимостей

На диаграмме зависимостей на нужно имя класса и возможно те методы, которые участвуют во взаимодействии.

Двойная стрелочка – дружественные связи, одна – схема использования. На этой диаграмме можно показать, какой метод имеет доступ к методу другого класса и какой метод вызывается

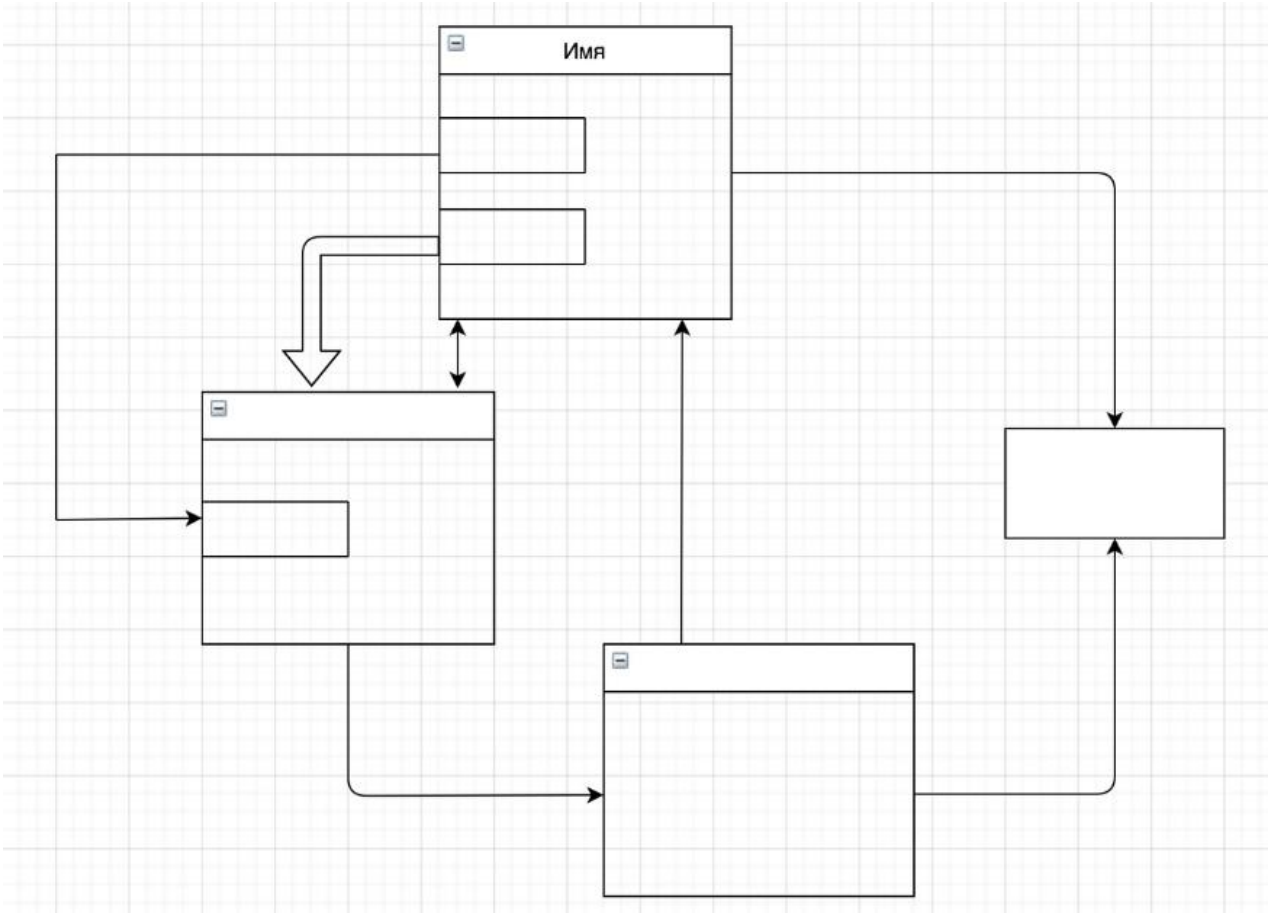
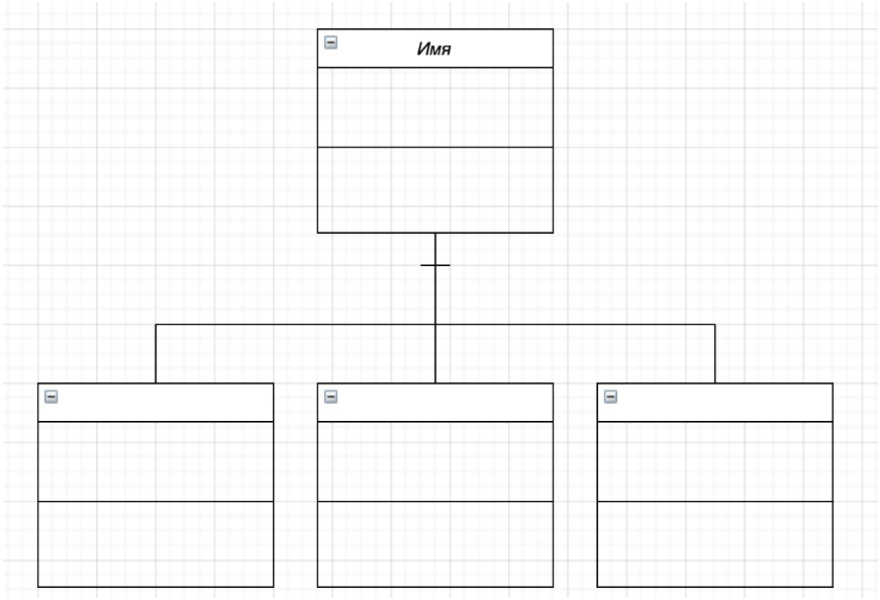


Диаграмма наследования

На данной диаграмме нас интересуют какие методы и компоненты объектов.

Нам надо четко понимать какие компоненты есть у суперкласса и подклассов, и какие методы. Напомним, что в объектно-ориентированном анализе у нас суперкласс всегда абстрактный. К этому надо стремиться и при проектировании.



11. Структурные паттерны: адаптер (Adapter), декоратор (Decorator), компоновщик (Composite), мост (Bridge), заместитель (Proxy), фасад (Facade). Их преимущества и недостатки.

Структурные паттерны вязаны с композицией классов и объектов. Они используют наследование для создания интерфейсов. Структурные паттерны создания объектов определяют способы компоновки объектов для получения новой функциональности.

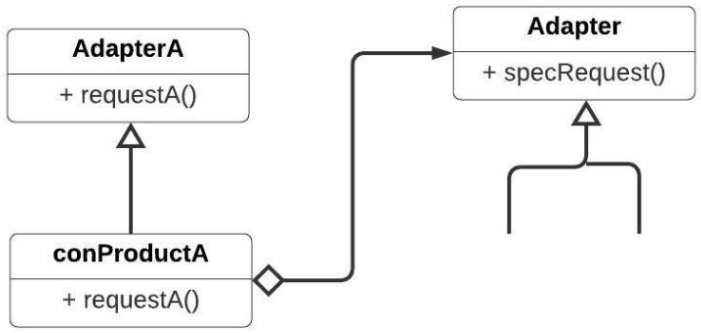
Adapter

Идея: у нас один и тот же объект выполняет несколько ролей, физически это один объект. В разных местах мы работаем с ним по-разному, т.е. разные интерфейсы. Выделить простой класс, а посредники будет представлять нужный класс для работы.

Адаптер надо применять в следующих ситуациях:

- Один объект может выступать в нескольких ролях.
- Нам нужно внедрить в систему сторонние классы, имеющие другой интерфейс.
- Мы, используя полиморфизм, сформировали интерфейс для базового класса. Но, какие-то определенные сущности, определенные от базового класса, должны поддерживать еще какой-то функционал. Мы не можем расширить этот функционал и изменять написанный код. Мы можем эту проблему решить добавлением функционала на основе того, что есть, за счет адаптера. Причем это можно сделать не для всей иерархии, а для определенного объекта определенного класса или группы классов.

У нас есть наш объект, и мы хотим подменить интерфейс этого объекта. Мы будем работать с этим объектом через объект другого класса. Таким образом, мы любой объект можем внедрить в любое место, поменяв его интерфейс.



Базовый класс - AdapterA, задача которого - подменить интерфейс.

Базовый класс, который нам надо адаптировать - Adapter, его интерфейс надо подменить.

Класс, который решает эту проблему - ConAdapterA. Он будет подменять интерфейс.

Достоинства

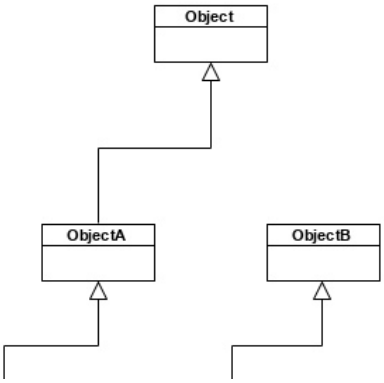
- Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.
- Позволяет адаптировать интерфейс к требуемому
- Позволяет разделить роли сущности
- Можно независимо развивать различные ответственности сущности
- Расширение интерфейса можно отнести к адаптеру

Недостатки

- Нужно плодить много классов поэтому увеличивается количество времени и памяти,необходимых для исполнения программы
- Дублирование кода(в различных конкретных адаптерах может требоваться одна и та же реализация методов)
- У различных сущностей с одной базой может быть работа с различными данными (с которыми адаптер может не уметь работать и прийдется перенести зависимости на один уровень ниже)
- Часто адаптер должен иметь доступ к реализации класса

```cpp
class BaseAdaptee {
public:
    virtual ~BaseAdaptee() = default;

    virtual void specificRequest() = 0;
};

class ConAdaptee : public BaseAdaptee {
public:
    virtual void specificRequest() override { cout << "Method ConAdaptee;" << endl; }
};

class Adapter {
public:
    virtual ~Adapter() = default;

    virtual void request() = 0;
};

class ConAdapter : public Adapter {
private:
    shared_ptr<BaseAdaptee>  adaptee;

public:
    ConAdapter(shared_ptr<BaseAdaptee> ad) : adaptee(ad) {}

    virtual void request() override;
};

# pragma region Methods
void ConAdapter::request(){
    cout << "Adapter: ";

    if (adaptee)
        adaptee->specificRequest();
    else
        cout << "Empty!" << endl;
}

# pragma endregion

int main() {
    shared_ptr<BaseAdaptee> adaptee = make_shared<ConAdaptee>();
    shared_ptr<Adapter> adapter = make_shared<ConAdapter>(adaptee);

    adapter->request();
}
```

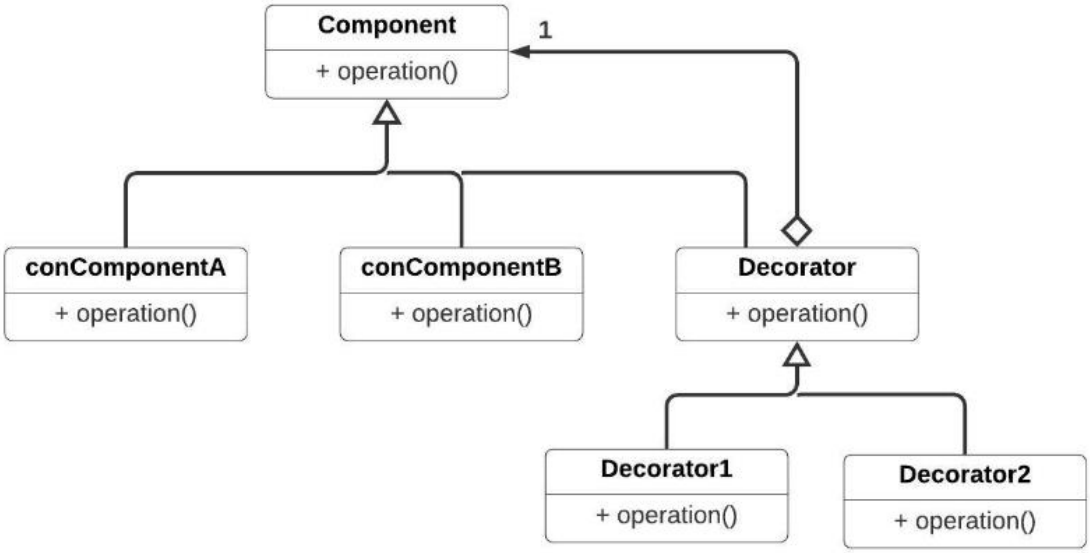
Decorator



Идея: пусть у нас есть следующие объекты и мы хотим добавить им общий функционал.

Если мы так сделаем, у нас будет повторяющийся код, и резко возрастет иерархия наследования.

Чтобы это исправить выделяют отдельный класс – декоратор:



Есть класс Component. От него производные классы - какие-то конкретные компоненты: ComponentA и ComponentB. Мы создаем класс Decorator, который обладает таким же интерфейсом, но, кроме всего прочего, он имеет ссылку на базовый класс Component. От него мы уже можем порождать конкретные декораторы - Decorator1 и Decorator2. Decorator имеет ссылку на компонент, и вызывая метод компонента, на который держится ссылка, добавляет свой функционал. Таким образом, мы можем добавить к любому компоненту единый функционал.

Достоинства

- В случае ,если необходимо изменить сущность, можно во время выполнения программы продекорировать , таким образом меняя поведения объектов
- Отсутствие разрастания иерархии
- Отсутствия дублирования кода, данный код просто уходит в конкретный декоратор

Недостатки

- Без декораторов можно просто напрямую пользоваться методом класса , а тут работа с декоратором который может долго, полиморфно делать цепочку вызовов(долгая работа)
- Никто не отвечает за оборачивания декораторов: за порядок их вызова и оборачивания(вся ответственность на программисте)
- В случае если в цепочке вызовов декоратора необходимо изменить какую-либо обертку или удалить ее ,придется заново оборачивать исходный объект

```
class Component
{
public:
    virtual ~Component() = default;

    virtual void operation() = 0;
};

class ConComponent : public Component
{
public:
    void operation() override { cout << "ConComponent; "; }
};

class Decorator : public Component
{
protected:
    shared_ptr<Component> component;

public:
    Decorator(shared_ptr<Component> comp) : component(comp) {}
};

class ConDecorator : public Decorator
{
public:
    using Decorator::Decorator;

    void operation() override;
};

# pragma region Method
void ConDecorator::operation()
{
    if (component)
    {
        component->operation();

        cout << "ConDecorator; ";
    }

}

# pragma endregion

int main()
{
    shared_ptr<Component> component = make_shared<ConComponent>();
    shared_ptr<Component> decorator1 = make_shared<ConDecorator>(component);

    decorator1->operation();
    cout << endl;

    shared_ptr<Component> decorator2 = make_shared<ConDecorator>(decorator1);

    decorator2->operation();
    cout << endl;
}
```

Composite

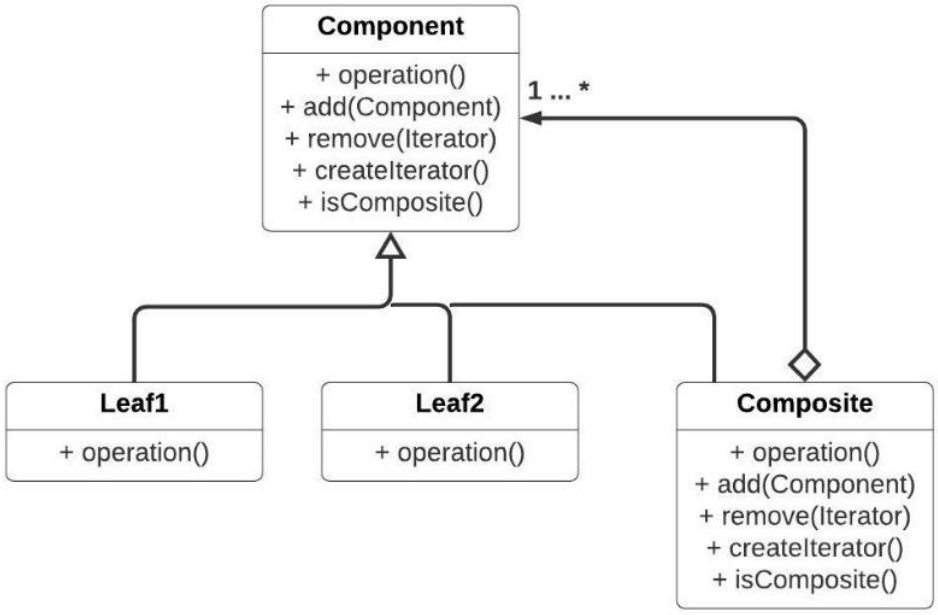
Идея: вынести интерфейс, который предлагает контейнер (объект, включающий в себя другие объекты), на уровень базового класса

Компоновщик компонует объекты в древовидную структуру, в которой над всей иерархией объектов можно выполнять такие же действия, как над простым элементом или над его частью.

Каждый композит может включать в себя как простые компоненты, так и другие композиты. Получается древовидная структура. Системе неважно, композит это или не композит. Она работает с любым элементом.

Задача методов интерфейса для компонентов - пройтись по списку компонент и выполнить соответствующий метод.

Базовый класс - Component. Нам должно быть безразлично, с каким объектом мы работаем: то ли это один компонент, то ли это объект, включающий в себя другие объекты (контейнер). Если это контейнер, то нам надо работать с содержимым контейнера, удалять, добавлять в него объекты. Идея - вынести этот интерфейс на уровень базового класса (добавление компонента - add(Component), удаление компонента - remove(Iterator), createIterator()). Нам надо четко понимать, когда мы работаем с каким-то компонентом, чем он является: один объект или контейнер. Для этого нам нужен метод isComposite(). То, что мы можем делать с Component - operation() - чисто виртуальные методы. Остальные (add, remove, и т. д.) мы будем реализовывать.



Leaf - простой компонент, его задачей будет только реализовать остальные методы - operation.

Composite - составной класс. Ралузует все те методы, что есть в компоненте. Он содержит в себе список компонент.

Достоинства

- Упрощает архитектуру клиента при работе со сложным деревом компонентов.
- Облегчает добавление новых видов компонентов.

Недостатки

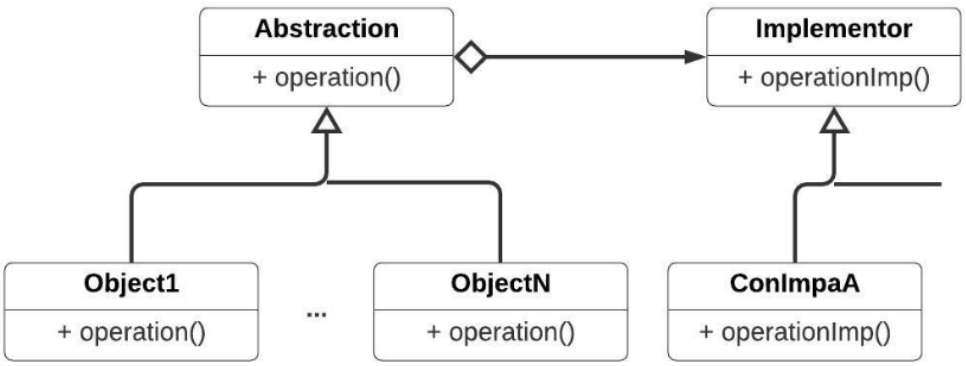
- Нет того кто контролирует композит и ответственен за его создание(принимает решение из чего будет состоять композит)
- Пусть композит состоит из композитов и объектов, что если необходимо отобрать только композиты?
- Что если из вложенного композита что-то нужно удалить?

```
class Component;

using PtrComponent = shared_ptr<Component>;
using VectorComponent = vector<PtrComponent>;

class Component
{
public:
    using value_type = Component;
    using size_type = size_t;
    using iterator = VectorComponent::const_iterator;
    using const_iterator = VectorComponent::const_iterator;

    virtual ~Component() = default;

    virtual void operation() = 0;

    virtual bool isComposite() const { return false; }
    virtual bool add(initializer_list<PtrComponent> comp) { return false; }
    virtual bool remove(const iterator& it) { return false; }
    virtual iterator begin() const { return iterator(); }
    virtual iterator end() const { return iterator(); }
};

class Figure : public Component
{
public:
    virtual void operation() override { cout << "Figure method;" << endl; }
};

class Camera : public Component
{
public:
    virtual void operation() override { cout << "Camera method;" << endl; }
};

class Composite : public Component
{
private:
    VectorComponent vec;

public:
    Composite() = default;
    Composite(PtrComponent first, ...);

    void operation() override;

    bool isComposite() const override { return true; }
    bool add(initializer_list<PtrComponent> list) override;
    bool remove(const iterator& it) override { vec.erase(it); return true; }
    iterator begin() const override { return vec.begin(); }
    iterator end() const override { return vec.end(); }
};

# pragma region Methods
Composite::Composite(PtrComponent first, ...)
{
    for (shared_ptr<Component>* ptr = &first; *ptr; ++ptr)
        vec.push_back(*ptr);
}

void Composite::operation()
{
    cout << "Composite method:" << endl;
    for (auto elem : vec)
        elem->operation();
}
```

```
bool Composite::add(initializer_list<PtrComponent> list)
{
    for (auto elem : list)
        vec.push_back(elem);

    return true;
}

# pragma endregion

int main()
{
    using Default = shared_ptr<Component>;
    PtrComponent fig = make_shared<Figure>(), cam = make_shared<Camera>();
    auto composite1 = make_shared<Composite>(fig, cam, Default{});

    composite1->add({ make_shared<Figure>(), make_shared<Camera>() });
    composite1->operation();
    cout << endl;

    auto it = composite1->begin();

    composite1->remove(++it);
    composite1->operation();
    cout << endl;

    auto composite2 = make_shared<Composite>(make_shared<Figure>(), composite1,
Default());

    composite2->operation();
}
```

Bridge

Паттерн Мост (или Bridge) отделяет саму абстракцию, сущность, от реализаций. Мы можем независимо менять логику (сами абстракции) и наращивать реализацию (добавлять новые классы реализации).



Таким образом, для каждого объекта мы можем менять реализацию динамически в любой момент во время выполнения. Как логика самого объекта может меняться, так и реализация может меняться. Мы независимо от сущности добавляем реализации и наоборот.

Используется, когда:

- Нам нужно во время выполнения менять реализацию

- Когда у нас большая иерархия, и по разным ветвям этой иерархии идут одинаковые реализации. Дублирование кода мы выносим в дерево реализаций. Такой подход дает возможность независимо изменять управляющую логику и реализацию.

Недостатки

- При вызове метода сущности вызывается метод реализации что увеличивает время работы
- Не всегда можно свести к связи между абстрактными(базовыми) понятиями, в таком случае придётся вводить связи между конкретными сущностями и лишиться подмены

Достоинства

- Убирает дублирование кода
- Позволяет иерархии не разрастаться
- Во время выполнения программы можем менять реализацию
- Получаем еще один слой защиты нашей реализации

```
class Implementor
{
public:
    virtual ~Implementor() = default;

    virtual void operationImp() = 0;
};

class Abstraction
{
protected:
    shared_ptr<Implementor> implementor;

public:
    Abstraction(shared_ptr<Implementor> imp) : implementor(imp) {}
    virtual ~Abstraction() = default;

    virtual void operation() = 0;
};

class ConImplementor : public Implementor
{
public:
    virtual void operationImp() override { cout << "Implementor;" << endl; }
};

class Entity : public Abstraction
{
public:
    using Abstraction::Abstraction;

    virtual void operation() override { cout << "Entity: "; implementor-
>operationImp(); }
};

int main()
{
    shared_ptr<Implementor> implementor = make_shared<ConImplementor>();
    shared_ptr<Abstraction> abstraction = make_shared<Entity>(implementor);

    abstraction->operation();
}
```

Proxy

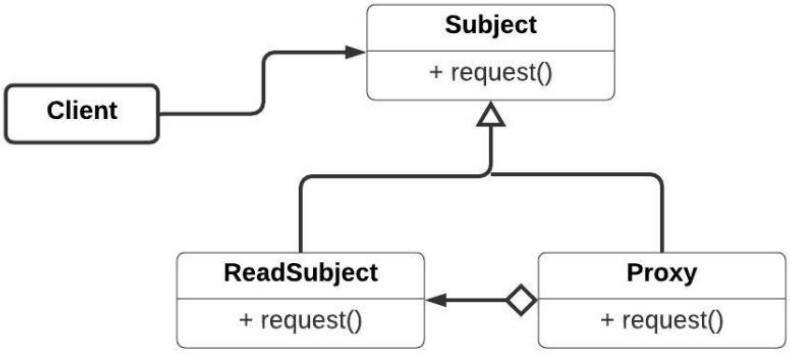
Заместитель (или Proxy) позволяет нам работать не с реальным объектом, а с другим объектом, который подменяет реальный. В каких целях это можно делать?

1. Подменяющий объект может контролировать другой объект, задавать правила доступа к этому объекту. Например, у нас есть разные категории пользователей. В зависимости от того, какая у пользователя категория категория, определять, давать доступ к самому объекту или не давать. Это как защита.

2. Так как запросы проходят через заместителя, он может контролировать запросы, заниматься статистической обработкой: считать количество запросов, какие запросы были и так далее.

3. Разгрузка объекта с точки зрения запросов. Дело в том, что реальные объекты какие-то операции могут выполнять крайне долго, например, обращение "поиск в базе чего-либо" или "обращение по сети куда-то". Это выполняется долго. Proxy может сохранять предыдущий ответ и при следующем обращении смотреть, был ответ на этот запрос или не был. Если ответ на этот вопрос был, он не обращается к самому хозяину, он заменяет его тем ответом, который был ранее. Естественно, если состояние объекта изменилось, Proxy должен сбросить ту историю, которую он накопил

Это очень удобно, когда у нас тяжелые объекты, операции которых выполняются долго. Зачем еще раз спрашивать, если мы уже спрашивали об этом? Proxy может выдать нам этот ответ.



Базовый класс Subject, реальный объект RealObject и объект Proxy, который содержит ссылку на объект, который он замещает.

Когда мы работаем через указатель на базовый объект Subject, мы даже не можем понять, с кем мы реально работаем: с непосредственно объектом RealSubject или с его заместителем Proxy. А заместитель может выполнять те задачи, которые мы на него возложили.

Если состояние RealObject изменилось, Прокси должен сбросить историю, которую он накопил.

Преимущества

- Возможность контролировать какой-либо объект незаметно от клиента.
- Proxy может работать, когда объекта нет. Например, мы запросили объект, может быть объект был, но ответ для Proxy устарел, этого объекта нет. Proxy может вернуть нам этот ответ, указав, что он устаревший.
- Proxy может отвечать за жизненный цикл объекта. Он его может создавать и удалять.

Недостатки

- Может увеличиться время доступа к объекту (так как идет дополнительная обработка через Proxy).
- Proxy должен хранить историю обращений, если такая задача стоит. Это влияет на память, она расходуется на хранение запросов и ответов на них.

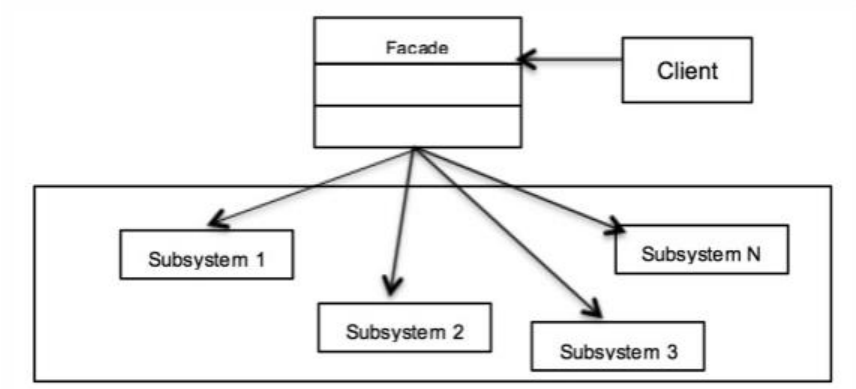
```
class Subject
{
public:
    virtual ~Subject() = default;

    virtual pair<bool, double> request(size_t index) = 0;
    virtual bool changed() { return true; }
};

class RealSubject : public Subject
{
private:
    bool flag{ false };
    size_t counter{ 0 };

public:
    virtual pair<bool, double> request(size_t index) override;
    virtual bool changed() override;
};

class Proxy : public Subject
{
protected:
    shared_ptr<RealSubject> realsubject;

public:
    Proxy(shared_ptr<RealSubject> real) : realsubject(real) {}
};

class ConProxy : public Proxy
{
private:
    map<size_t, double> cache;

public:
    using Proxy::Proxy;
    virtual pair<bool, double> request(size_t index) override;
};

#pragma region Methods
bool RealSubject::changed()
{
    if (counter == 0)
        flag = true;

    if (++counter == 7)
    {
        counter = 0;
        flag = false;
    }
    return flag;
}


pair<bool, double> RealSubject::request(size_t index)
{
    random_device rd;
```

```
    mt19937 gen(rd());

    return pair<bool, double>(true, generate_canonical<double, 10>(gen));
}

pair<bool, double> ConProxy::request(size_t index)
{
    pair<bool, double> result;

    if (!realsubject)
    {
        cache.clear();

        result = pair<bool, double>(false, 0.);
    }
    else if (!realsubject->changed())
    {
        cache.clear();
        result = realsubject->request(index);
        cache.insert(map<size_t, double>::value_type(index, result.second));
    }
    else
    {
        map<size_t, double>::const_iterator it = cache.find(index);

        if (it != cache.end())
        {
            result = pair<bool, double>(true, it->second);
        }
        else
        {
            result = realsubject->request(index);
            cache.insert(map<size_t, double>::value_type(index, result.second));
        }
    }

    return result;
}
#pragma endregion

int main()
{
    shared_ptr<RealSubject> subject = make_shared<RealSubject>();
    shared_ptr<Subject> proxy = make_shared<ConProxy>(subject);

    for (size_t i = 0; i < 21; ++i)
    {
        cout << "( " << i + 1 << ", " << proxy->request(i % 3).second << " )" << endl;

        if ((i + 1) % 3 == 0)
            cout << endl;
    }
}
```

Facade

У нас есть группа объектов, связанных между собой. Причем эти связи довольно жёсткие. Чтобы извне не работать с каждым объектом в отдельности, мы можем эти все объекты объединить в один класс - фасад, который будет представлять интерфейс для работы со всем этим объединением.

Нам не нужно извне работать с мелкими объектами. Кроме того, фасад может выполнять такую роль, как следить за целостностью нашей системы. Извне, мы, работая с фасадом, работаем, как с простым объектом. Он представляет интерфейс одной сущностью, а внутри у нас целый мир мир из объектов. Таким образом, упрощается взаимодействие, уменьшается количество связей за счет объединений фасада.



Достоинства

- Таким образом мы работаем с одним объектом
- Доступ к подсистеме обычно делается через 1 метод(run,excecute)
- Оболочка скрывает целостность подсистемы

Недостатки

- Фасад рискует стать объектом привязанным ко всем классам программы(God-object)

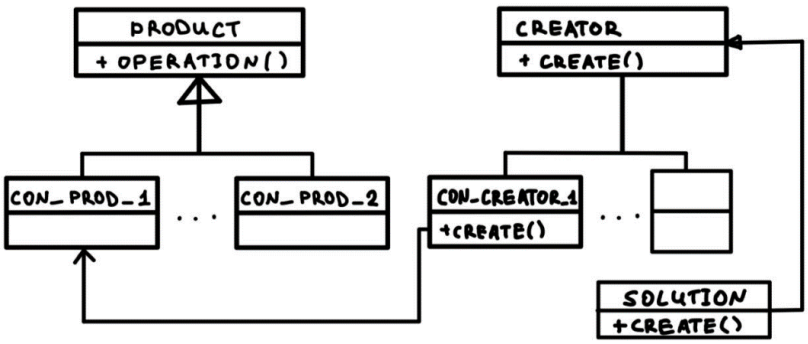
12. Порождающие паттерны: фабричный метод (Factory Method), абстрактная фабрика (Abstract Factory), строитель (Builder). Их преимущества и недостатки.

Порождающие паттерны связаны с созданием классов или объектов.

Factory method.

Основная идея заключается в том, чтобы избавиться от явного создания объектов. Эти будут заниматься «креаторы». При этом то, какой конкретно объект будет создан, т.е. то, какой креатор использовать будут говорить нам особенная таблица. Иными словами, будет выделяться отдельная сущность, которая будет принимать решение о том, какой объект создавать.

Это дает возможность выбирать объект, который создавать, во время выполнения программы. Так же, во время выполнения программы, мы можем подменять создание одного объекта, на создание другого.



За счет креатора можно использовать созданные объекты в разных местах.

Достоинства

- Также таким образом мы можем разделять ответственности:
  - **Creator** отвечает за создание сущностей
  - Умный указатель отвечает за саму сущность
  - Можно создать еще одну сущность которая будет отвечать за выбор того, какую сущность создавать, будет возвращать нам конкретный **creator(solution)**
- Мы можем влиять на создание объекта во время выполнения программы
- Можно реализовать повторное использование одних и тех же объектов
- Упрощает добавление новых классов без изменения написанного кода

Solution должен получить на вход набор параметров для выбора соответствующего **creator**, также должна быть произведена регистрация имеющихся **creator**

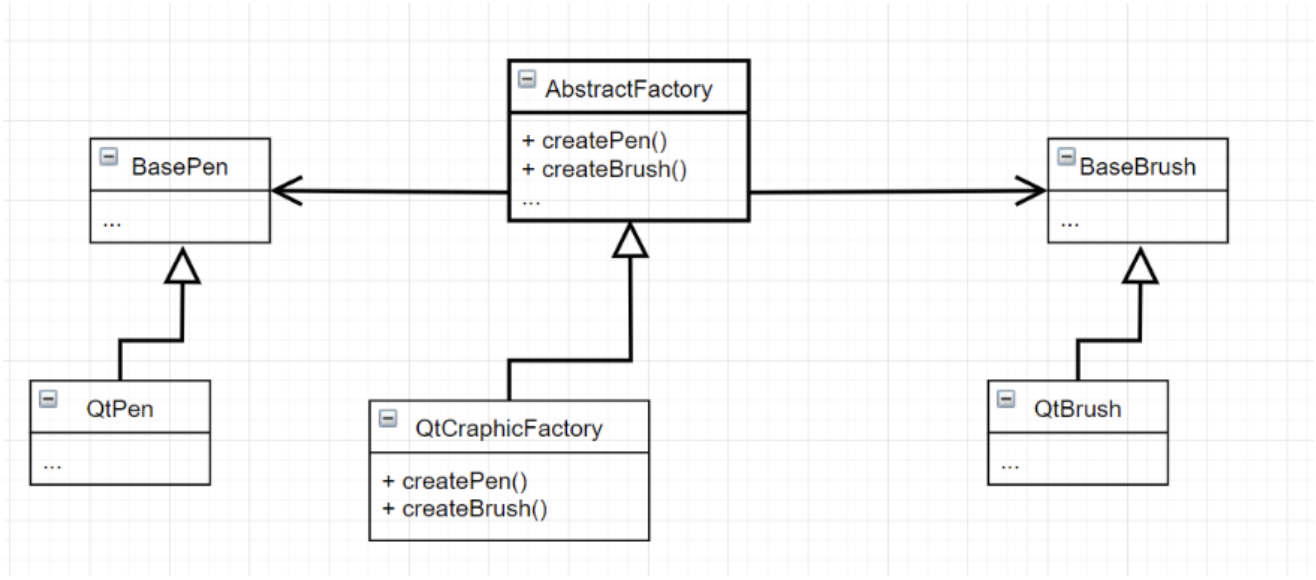
Недостатки

- Резко разрастается количество кода
- Используется полиморфизм и время работы программы растет
- Можно сделать на шаблонах , но после одного изменения придётся перекомпилировать все

```cpp
# pragma region Product
class Product
{
public:
    virtual ~Product() = default;

    virtual void run() = 0;
};

class ConProd1 : public Product
{
public:
    ConProd1() { cout << "Calling the ConProd1 constructor;" << endl; }
    ~ConProd1() override { cout << "Calling the ConProd1 destructor;" << endl; }

    void run() override { cout << "Calling the run method;" << endl; }
};

# pragma endregion

class Creator
{
public:
    virtual ~Creator() = default;

    virtual unique_ptr<Product> createProduct() = 0;
};
```

```cpp
template <typename Derived, typename Base>
concept Derivative = is_abstract_v<Base> && is_base_of_v<Base, Derived>;

template <Derivative<Product> Tprod>
class ConCreator : public Creator
{
public:
    unique_ptr<Product> createProduct() override
    {
        return unique_ptr<Product>(new Tprod());
    }
};

class User
{
public:
    void use(shared_ptr<Creator>& cr)
    {
        shared_ptr<Product> ptr = cr->createProduct();

        ptr->run();
    }
};

int main()
{
    shared_ptr<Creator> cr = make_shared<ConCreator<ConProd1>>();

    unique_ptr<User> us = make_unique<User>();

    us->use(cr);
}
```

Abstract Factory.



Содержит несколько креаторов для разных семейств иерархии подобъектов.

Достоинства

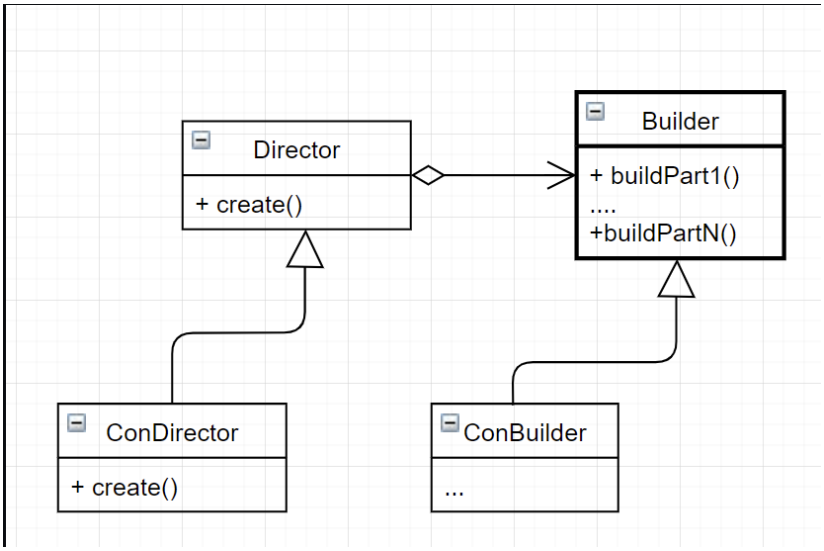
- Облегчает контроль за созданием семейства объектов
- Клиентский код не привязан к конкретным продуктам

Недостатки

- У абстрактной фабрики должен быть базовый интерфейс, если у библиотек используются различные понятия, выделить базовое для них не всегда возможно

```
class Image {};
class Color {};

class BaseGraphics
{
public:    virtual ~BaseGraphics() = 0;
};
BaseGraphics::~BaseGraphics() {}

class BasePen {};
class BaseBrush {};

class QtGraphics : public BaseGraphics
{
public:
    QtGraphics(shared_ptr<Image> im) { cout << "Calling the QtGraphics constructor;" <<
endl; }
    ~QtGraphics() override { cout << "Calling the QtGraphics destructor;" << endl; }
};

class QtPen : public BasePen {};
class QtBrush : public BaseBrush {};

class AbstractGraphFactory
{
public:
    virtual ~AbstractGraphFactory() = default;

    virtual unique_ptr<BaseGraphics> createGraphics(shared_ptr<Image> im) = 0;
    virtual unique_ptr<BasePen> createPen(shared_ptr<Color> cl) = 0;
    virtual unique_ptr<BaseBrush> createBrush(shared_ptr<Color> cl) = 0;
};

class QtGraphFactory : public AbstractGraphFactory
{
public:
    unique_ptr<BaseGraphics> createGraphics(shared_ptr<Image> im) override
    {
        return make_unique<QtGraphics>(im);
    }

    unique_ptr<BasePen> createPen(shared_ptr<Color> cl) override
    {
        return make_unique<QtPen>();
    }

    unique_ptr<BaseBrush> createBrush(shared_ptr<Color> cl) override
    {
        return make_unique<QtBrush>();
    }
};

int main()
{
    shared_ptr<AbstractGraphFactory> grfactory = make_shared<QtGraphFactory>();

    shared_ptr<Image> image = make_shared<Image>();

    shared_ptr<BaseGraphics> graphics1 = grfactory->createGraphics(image);
}
```

Builder



Позволяет создавать объект поэтапно.

Также нужен отдельный класс для контролирования создания сложного объекта.

Builder создает объект, *Director* контролирует создание (подготавливает данные для создания и отдает объект) - разделение ответственности.

Когда надо использовать? Для поэтапного создания сложных объектов. Когда создание объекта разнесено в коде, объект создается не сразу (например, данные подготавливаются поэтапно).

Достоинства

- За создание и контроль отвечают различные сущности
- Позволяет использовать один и тот же код для создания различных продуктов.
- Позволяет создавать продукты пошагово.
- Позволяет менять количество этапов создания объекта(введением нового Директора)

Недостатки

- Усложняет код программы из-за введения дополнительных классов

Проблема: с данными - конкретные строители базируются на одних и тех же данных (чтобы могли подменить один объект одного строителя другим).

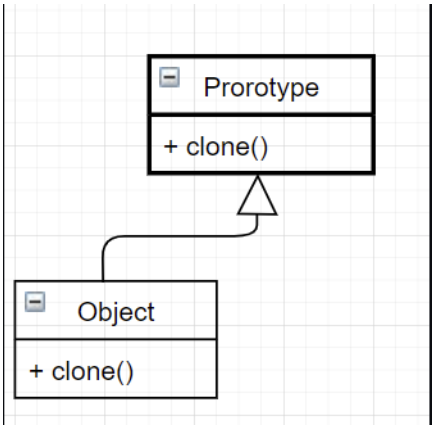
```
class Product
{
public:
   Product() { cout << "Calling the ConProd1 constructor;" << endl; }
   ~Product() { cout << "Calling the ConProd1 destructor;" << endl; }

   void run() { cout << "Calling the run method;" << endl; }
};

class Builder
{
public:
   virtual ~Builder() = default;

   virtual bool buildPart1() = 0;
   virtual bool buildPart2() = 0;

   shared_ptr<Product> getProduct();

protected:
   virtual shared_ptr<Product> createProduct() = 0;

   shared_ptr<Product> product;
};
```

```
class ConBuilder : public Builder
{
public:
   bool buildPart1() override
   {
      cout << "Completed part: " << ++part << ";" << endl;
      return true;
   }
   bool buildPart2() override
   {
      cout << "Completed part: " << ++part << ";" << endl;
      return true;
   }

protected:
   virtual shared_ptr<Product> createProduct() override;

private:
   size_t part{ 0 };
};

class Director
{
public:
   shared_ptr<Product> create(shared_ptr<Builder> builder)
   {
      if (builder->buildPart1() && builder->buildPart2()) return builder->getProduct();

      return shared_ptr<Product>();
   }
};

# pragma region Methods
shared_ptr<Product> Builder::getProduct()
{
   if (!product) { product = createProduct(); }

   return product;
}

shared_ptr<Product> ConBuilder::createProduct()
{
   if (part == 2) { product = make_shared<Product>(); }

   return product;
}
# pragma endregion

int main()
{
   shared_ptr<Builder> builder = make_shared<ConBuilder>();
   shared_ptr<Director> director = make_shared<Director>();

   shared_ptr<Product> prod = director->create(builder);

   if (prod)
      prod->run();
}
```

13. Порождающие паттерны: одиночка (Singleton), прототип (Prototype), пул объектов (Object Pool). Их преимущества и недостатки.

Порождающие паттерны связаны с созданием классов или объектов.

Prototype



Позволяет создавать объекты на основе других, не вызывая *creator*.

Мы добавляем в базовый класс метод *clone*(), возвращающий указатель на себя, производные классы реализуют *clone*() под себя, возвращая указатель на подобный объект.

Достоинства

- Позволяет клонировать объекты, не привязываясь к их конкретным классам.
- Ускоряет создание объектов.

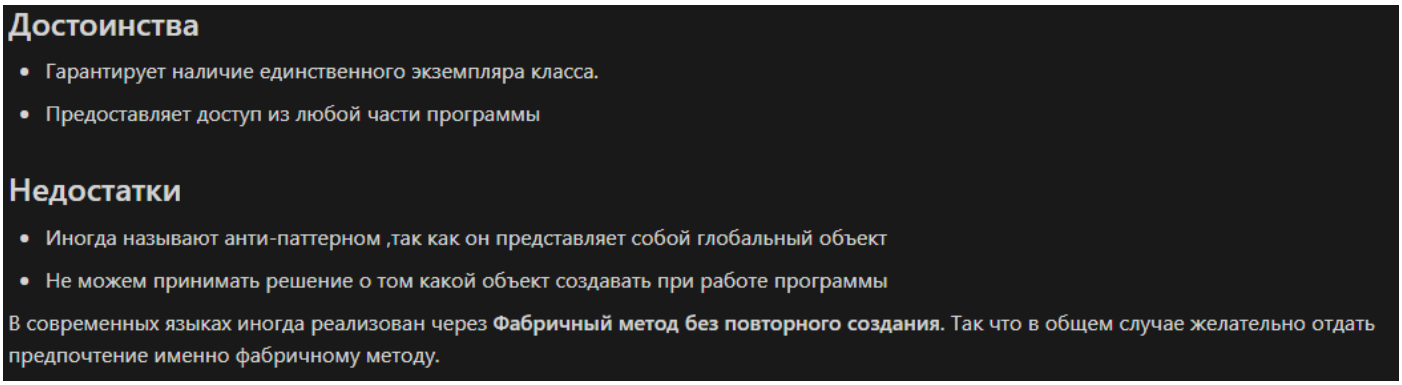
Недостатки

- Сложно клонировать составные объекты, имеющие ссылки на другие объекты,а также если во внутреннем представлении объекта присутствуют другие объекты

```cpp
class BaseObject
{
public:
    virtual ~BaseObject() = default;

    virtual unique_ptr<BaseObject> clone() = 0;
};

class Object1 : public BaseObject
{
public:
    Object1() { cout << "Calling the default constructor;" << endl; }
    Object1(const Object1& obj) { cout << "Calling the Copy constructor;" << endl; }
    ~Object1() override { cout << "Calling the destructor;" << endl; }

    unique_ptr<BaseObject> clone() override
    {
        return make_unique<Object1>(*this);
    }
};

int main()
{
    shared_ptr<BaseObject> ptr1 = make_shared<Object1>();

    auto ptr2 = ptr1->clone();
}
```

Singleton

Подойдет, если нам необходимо иметь во всех системе объект только в единственном экземпляре.



Проблема: не накладываем ограничения на объект (может иметь конструкторы), но если мы работаем через эту оболочку, это гарантирует то, что этот объект только 1.

Альтернатива: фабричный метод, который создает объект только один раз.

```
using namespace std;

class Product
{
public:
    static shared_ptr<Product> instance()
    {
        class Proxy : public Product {};

        static shared_ptr<Product> myInstance = make_shared<Proxy>();

        return myInstance;
    }
    ~Product() { cout << "Calling the destructor;" << endl; }

    void f() { cout << "Method f;" << endl; }

    Product(const Product&) = delete;
    Product& operator =(const Product&) = delete;

private:
    Product() { cout << "Calling the default constructor;" << endl; }
};

int main()
{
    shared_ptr<Product> ptr(Product::instance());

    ptr->f();
}
```

Object Pool

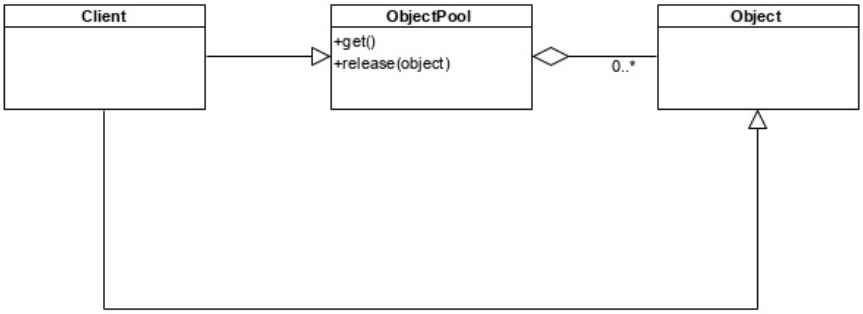
Если нам необходим, возможно ограниченный, набор определенных объектов. При запросе объекта мы даем его из этого набора, после использования возвращаем назад, при этом вернув в исходное состояние.

Используется, когда создание или уничтожение какого-либо объекта - трудоемкий процесс и надо "держать" определенное количество объектов в системе.

Задачи:

- Он держит эти объекты.
- Может их создавать (то есть может расширяться).
- По запросу отдает объект.
- Если клиенту этот объект не нужен, он может его вернуть в пул.
- Исходя из пунктов 3 и 4, для каждого включенного объекта в пул мы должны установить, используется он или не используется.

Клиент может принимать объект и возвращать его в пул объектов.



Достоинства

- Улучшает производительность(так как у нас не создаются/уничтожаются множества объектов)

Недостатки

- После использования объекта мы возвращаем его в пул, и здесь возможна так называемая утечка информации. Мы работали с этим объектом. Вернув его в пул, он находится в том состоянии, с которым мы с ним перед этим работали. Его надо либо вернуть в исходное состояние, либо очистить, чтобы при отдаче этого объекта другому клиенту не произошла утечка информации.

Пулл объектов - контейнерный класс, удобно использовать итераторы.

Необходимо знать, занят объект или нет, используем пару: ключ (bool) и объект.

```
template <typename T>
concept PoolObject = requires(T t)
{
    t.clear();
};

class Product
{
private:
    static size_t count;

public:
    Product() { cout << "Constructor(" << ++count << ");" << endl; }
    ~Product() { cout << "Destructor(" << count-- << ");" << endl; }

    void clear() { cout << "Method clear: 0x" << this << endl; }
};
```

```
size_t Product::count = 0;

template <PoolObject Type>
class Pool
{
public:
    static shared_ptr<Pool<Type>> instance();

    shared_ptr<Type> getObject();
    bool releaseObject(shared_ptr<Type>& obj);
    size_t count() const { return pool.size(); }

    Pool(const Pool&) = delete;
    Pool& operator =(const Pool&) = delete;

private:
    vector<pair<bool, shared_ptr<Type>>> pool;

    Pool() {}

    pair<bool, shared_ptr<Type>> create();

    template <typename Type>
    friend ostream& operator << (ostream& os, const Pool<Type>& pl);
};

# pragma region ObjectPool class Methods
template <PoolObject Type>
shared_ptr<Pool<Type>> Pool<Type>::instance()
{
    static shared_ptr<Pool<Type>> myInstance(new Pool<Type>());

    return myInstance;
}

template <PoolObject Type>
shared_ptr<Type> Pool<Type>::getObject()
{
    size_t i;
    for (i = 0; i < pool.size() && pool[i].first; ++i);

    if (i < pool.size())
    {
        pool[i].first = true;
    }
    else
    {
        pool.push_back(create());
    }

    return pool[i].second;
}

template <PoolObject Type>
bool Pool<Type>::releaseObject(shared_ptr<Type>& obj)
{
    size_t i;
    for (i = 0; pool[i].second != obj && i < pool.size(); ++i);

    if (i == pool.size()) return false;

    obj.reset();
    pool[i].first = false;
    pool[i].second->clear();

    return true;
}
```

```
template <PoolObject Type>
pair<bool, shared_ptr<Type>> Pool<Type>::create()
{
    return { true, make_shared<Type>() };
}

# pragma endregion

template <typename Type>
ostream& operator << (ostream& os, const Pool<Type>& pl)
{
    for (auto elem : pl.pool)
        os << "{" << elem.first << ", 0x" << elem.second << "} ";

    return os;
}

int main()
{
    shared_ptr<Pool<Product>> pool = Pool<Product>::instance();

    vector<shared_ptr<Product>> vec(4);

    for (auto& elem : vec)
        elem = pool->getObject();

    pool->releaseObject(vec[1]);

    cout << *pool << endl;

    shared_ptr<Product> ptr = pool->getObject();
    vec[1] = pool->getObject();

    cout << *pool << endl;
}
```

15. Паттерны поведения: стратегия (Strategy), посетитель (Visitor), опекун (Memento), шаблонный метод (Template Method), хранитель (Holder), итератор (Iterator), свойство (Property). Их преимущества и недостатки.

Паттерны поведения связаны с взаимодействием объектов класса.

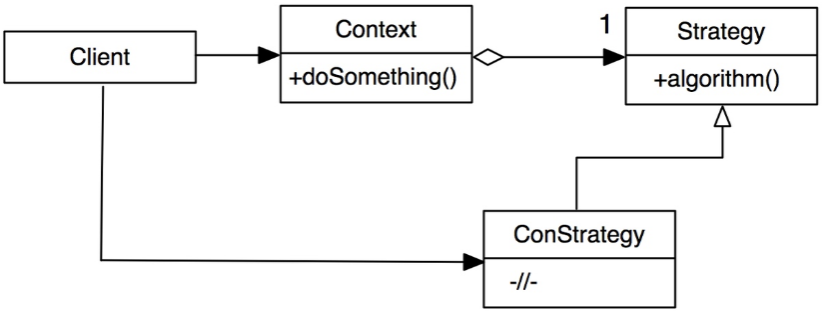
Strategy

Идея: нам во время выполнения надо менять реализацию какого-либо метода. Мы можем делать производные классы с разными реализациями и осуществлять "миграцию" между классами во врем выполнения — это неудобно, ибо мы начинаем работать с конкретными типами (классами).

То, что может меняться (это действие) вынести в отдельный класс, который будет выполнять только это действие. Мы можем во время выполнения подменять одно на другое.

Паттерн Стратегия определяет семейство всевозможных алгоритмов.

Клиент может установить для нашего класса конкретную стратегию (алгоритм) и, работая с объектом, он будет вызывать этот конкретный алгоритм. Во время работы мы можем этот алгоритм поменять.



Преимущества:

- Избавление от разрастания иерархии (в случаях различия методов у нескольких классов) => подмена метода внутри единственного класса.
- Избавление от дублирования кода
- Возможность легкого добавления функционала во время выполнения

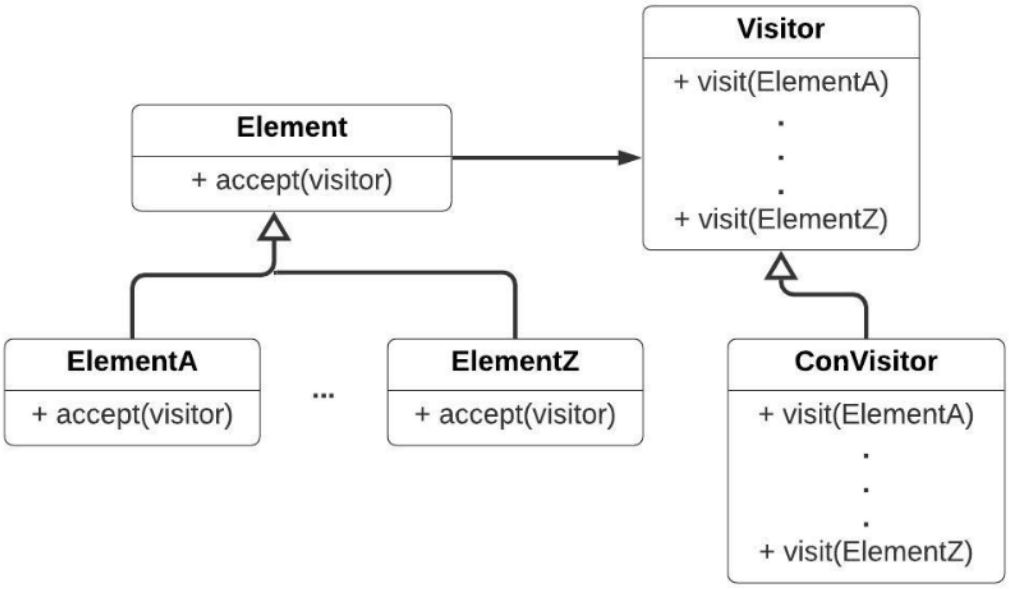
Недостатки:

- Конкретная стратегия может не работать с данными определенного класса.
- Необходимость установления дружественных связей

```
class Strategy {
public:
    virtual ~Strategy() = default;

    virtual void algorithm() = 0;
};

class ConStrategy1 : public Strategy {
public:
    void algorithm() override { cout << "Algorithm 1;" << endl; }
};

class ConStrategy2 : public Strategy {
public:
    void algorithm() override { cout << "Algorithm 2;" << endl; }
};

class Context {
protected:
    unique_ptr<Strategy> strategy;

public:
    explicit Context(unique_ptr<Strategy> ptr = make_unique<ConStrategy1>())
            : strategy(move(ptr)) {}
    virtual ~Context() = default;
    virtual void algorithmStrategy() = 0;
};

class Client1 : public Context {
public:
    using Context::Context;
    void algorithmStrategy() override { strategy->algorithm(); }
};

int main() {
    shared_ptr<Context> obj = make_shared<Client1>(make_unique<ConStrategy2>());

    obj->algorithmStrategy();
}
```

Visitor

Есть проблема, связанная с изменением интерфейса объектов. Если мы используем полиморфизм, мы не можем в производном кассе ни сузить, ни расширить интерфейс, так как он должен четко поддерживать интерфейс базового класса. Если нам необходимо расширить интерфейс, можно использовать паттерн Посетитель. Он позволяет во время выполнения (в отличие от паттерна Адаптера, который решает эту проблему до выполнения) подменить или расширить функционал.



Чтобы можно было поменять/расширять функционал, а базовом классе добавляем метод accept(Visiter). Соответственно, все производные классы могут подменять этот метод.

Посетитель один функционал собирает в одно место для разных классов. Для каждого такого класса/подкласса есть свой метод, который принимает элемент этого подкласса. Конкретный посетитель уже реализует этот функционал.

Преимущества

- Объединение разных иерархий в одну
- Значительное упрощение схемы
- Отсутствие оберточных функций

Недостатки

- Расширяется иерархия, добавляются новые классы. Проблема связи на уровне базовых классов
- Может меняться иерархия. Тогда посетитель не срабатывает.
- Необходимо знать реализацию объектов - установление дружественных связей.

```
class Visitor {
public:
    virtual ~Visitor() = default;

    virtual void visit(Circle& ref) = 0;
    virtual void visit(Rectangle& ref) = 0;
};

class Shape
{
public:
    virtual ~Shape() = default;

    virtual void accept(shared_ptr<Visitor> visitor) = 0;
};

class Circle : public Shape
{
public:
    void accept(shared_ptr<Visitor> visitor) override { visitor->visit(*this); }
};

class Rectangle : public Shape
{
public:
    void accept(shared_ptr<Visitor> visitor)  override { visitor->visit(*this); }
};

class ConVisitor : public Visitor
{
public:
    void visit(Circle& ref) override { cout << "Circle;" << endl; }
    void visit(Rectangle& ref) override { cout << "Rectangle;" << endl; }
};

class Figure : public Shape
{
    using Shapes = vector<shared_ptr<Shape>>;

private:
    Shapes shapes;

public:
    Figure(initializer_list<shared_ptr<Shape>> list)
    {
        for (auto&& elem : list)
            shapes.emplace_back(elem);
    }

    void accept(shared_ptr<Visitor> visitor)  override
    {
        for (auto& elem : shapes)
            elem->accept(visitor);
    }
};

int main()
{
    shared_ptr<Shape> figure = make_shared<Figure>(
            initializer_list<shared_ptr<Shape>>(
                    { make_shared<Circle>(), make_shared<Rectangle>(),
make_shared<Circle>() }
            )
    );
    shared_ptr<Visitor> visitor = make_shared<ConVisitor>();
    figure->accept(visitor);
}
```

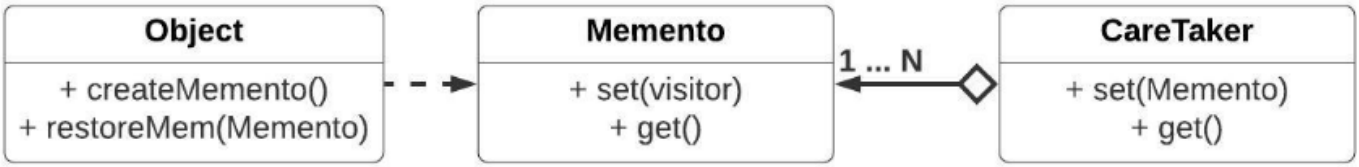
Memento

Когда мы выполняем много операций, объект изменяется, но, те изменения, которые у нас произошли, могут нас не устраивать, и мы можем вернуться к предыдущему состоянию нашего объекта (откат).

Как вариант - хранить те состояния, которые были у объекта. Если возложить эту задачу на объект - он получится тяжелым.

Идея - выделить эту обязанность другому объекту, задача которого - хранить предыдущие состояния нашего объекта, который, если нужно, позволит нам вернуться к какому-либо предыдущему состоянию.

Memento - снимок объекта в какой-то момент времени. Опекун отвечает за хранение этих снимков и по возможности вернуться к предыдущему состоянию объекта



Преимущества

- Позволяет не грузить сам класс задачей сохранять предыдущие состояния.
- Предоставляет возможность откатываться на несколько состояний назад.

Недостатки

- Опекуном надо управлять. Он наделал снимков, а они нам не нужны. Кто-то должен их очищать. Какой механизм очистки? Много памяти тратится.

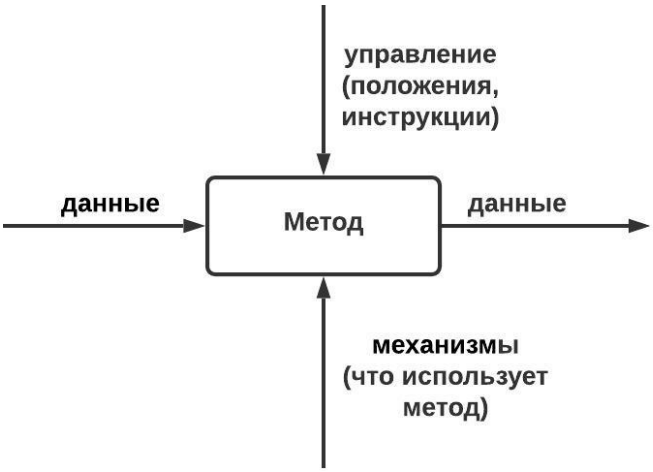
```
class Memento;

class Caretaker
{
public:
    unique_ptr<Memento> getMemento();
    void setMemento(unique_ptr<Memento> memento);

private:
    list<unique_ptr<Memento>> mementos;
};

class Originator
{
public:
    Originator(int s) : state(s) {}

    const int getState() const { return state; }
    void setState(int s) { state = s; }

    std::unique_ptr<Memento> createMemento() { return make_unique<Memento>(*this); }
    void restoreMemento(std::unique_ptr<Memento> memento);

private:
    int state;
};
```

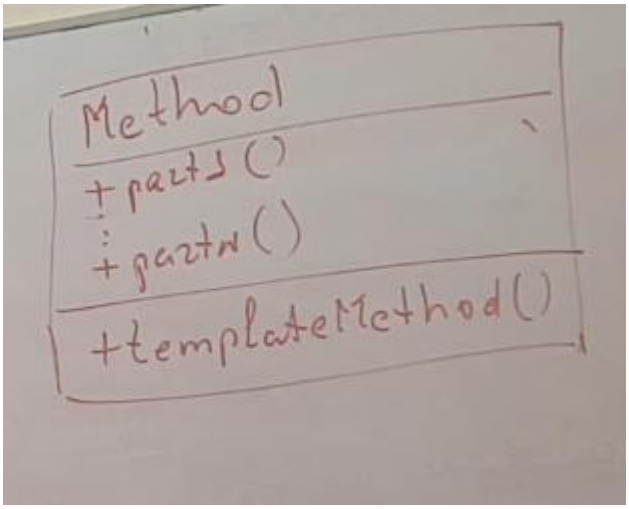
```
class Memento
{
    friend class Originator;

public:
    Memento(Originator o) : originator(o) {}

private:
    void setOriginator(Originator o) { originator = o; }
    Originator getOriginator() { return originator; }

private:
    Originator originator;
};

# pragma region Methods Caretaker
void Caretaker::setMemento(unique_ptr<Memento> memento)
{
    mementos.push_back(move(memento));
}

unique_ptr<Memento> Caretaker::getMemento() {

    unique_ptr<Memento> last = move(mementos.back());

    mementos.pop_back();

    return last;
}
# pragma endregion

# pragma region Method Originator
void Originator::restoreMemento(std::unique_ptr<Memento> memento)
{
    *this = memento->getOriginator();
}
# pragma endregion

int main()
{
    auto originator = make_unique<Originator>(1);
    auto caretaker = make_unique<Caretaker>();

    cout << "State = " << originator->getState() << endl;
    caretaker->setMemento(originator->createMemento());

    originator->setState(2);
    cout << "State = " << originator->getState() << endl;
    caretaker->setMemento(originator->createMemento());
    originator->setState(3);
    cout << "State = " << originator->getState() << endl;
    caretaker->setMemento(originator->createMemento());

    originator->restoreMemento(caretaker->getMemento());
    cout << "State = " << originator->getState() << endl;
    originator->restoreMemento(caretaker->getMemento());
    cout << "State = " << originator->getState() << std::endl;
    originator->restoreMemento(caretaker->getMemento());
    cout << "State = " << originator->getState() << std::endl;
}
```

Template Method

Этот паттерн является скелетом какого-либо метода. Мы любую задачу разбиваем на этапы - формируем шаги, которые мы выполняем для того, чтобы то, что мы получили на входе, преобразовать в результат, который нам нужен.

Диаграмма, которая, по существу, задает нам эти методы, и реальная диаграмма:





Достоинства:

- Даёт возможность предоставления собственной реализации метода
- Позволяет избежать дублирования кода в субклассах, т.к. общая логика алгоритма вынесена в абстрактный класс
- Упрощает расширение функциональности, т.к. добавлене новых шагов алгоритма или изменение порядка выполнения шагов может быть реализовано в субклассах без изменения общей структуры алгоритма.

Недостатки:

- Может привести к созданию большого количества субклассов, если алгоритм имеет много шагов и каждый шаг должен быть изменен в отдельном субклассе.
- Может усложнить понимание кода

```
class AbstractClass
{
public:
    void templateMethod()
    {
        primitiveOperation();
        concreteOperation();
        hook();
    }
    virtual ~AbstractClass() = default;

protected:
    virtual void primitiveOperation() = 0;
    void concreteOperation() { cout << "concreteOperation;" << endl; }
    virtual void hook() { cout << "hook Base;" << endl; }
};

class ConClassA : public AbstractClass
{
protected:
    void primitiveOperation() override { cout << "primitiveOperation A;" << endl; }
};
```

```
class ConClassB : public AbstractClass
{
protected:
    void primitiveOperation() override { cout << "primitiveOperation B;" << endl; }
    void hook() override { cout << "hook B;" << endl; }
};

int main()
{
    ConClassA ca;
    ConClassB cb;

    ca.templateMethod();
    cb.templateMethod();
}
```

Holder

Существует проблема. Предположим, у нас есть класс A, в котором есть метод f(). Мы не знаем, что творится внутри f(), и, естественно, мы используем механизм обработки исключительных ситуаций. Внутри f() происходит исключительная ситуация, она приводит к тому, что мы перескакиваем на какой-то обработчик, неизвестно где находящийся. Это приводит к тому, что объект p не удаляется - происходит утечка памяти.

```
A* p = new A;
p->f();   // Внутри f() происходит исключительная ситуация
delete p; // Объект p не удаляется
```

Идея: обернуть объект в оболочку, которая статическая распределяет память. Эта оболочка будет отвечать за этот указатель. И соответственно, поскольку мы статически распределили, когда будет вызываться деструктор, в деструкторе мы будем освобождать память.

Шаблон Holder. Мы можем указатель p обернуть в объект-хранитель. Этот объект будет содержать указатель на объект A. Задача объекта: при выходе из области видимости объекта-хранителя будет вызываться деструктор obj, в котором мы можем уничтожить объект A.

```
Holder<A> obj(new A);
```

Для объекта хранителя достаточно определить три операции - * (получить значение по указателю), ->(обратиться к методу объекта, на который указывает указатель) и bool(проверить, указатель указывает на объект, nullptr он или нет). Чтобы можно было записать obj->f();. То есть эта оболочка должна быть "прозрачной". Её задача должна быть только вовремя освободить память, выделенную под объект. Мы работаем с объектом класса A через эту оболочку.

Преимущества:

- Помогает избежать утечек памяти.

Недостатки:

- Опасность наличия операции взятия адреса (есть риск возникновения висячих указателей).

```
template <typename Type>
class Holder
{
private:
    Type* ptr{ nullptr };

public:
    Holder() = default;
    explicit Holder(Type* p) : ptr(p) {}
    Holder(Holder&& other) noexcept
    {
        ptr = other.ptr;
        other.ptr = nullptr;
    }
    ~Holder() { delete ptr; }

    Type* operator ->() noexcept { return ptr; }
    Type& operator *() noexcept { return *ptr; }
    operator bool() noexcept { return ptr != nullptr; }
    Type* release() noexcept
    {
        Type* work = ptr;
        ptr = nullptr;

        return work;
    }

    Holder(const Holder&) = delete;
    Holder& operator =(const Holder&) = delete;
};

class A
{
public:
    void f() { cout << "Function f of class A is called" << endl; }
};

int main()
{
    Holder<A> obj(new A{});

    obj->f();
}
```

Iterator

Пусть у нас есть класс:

```
struct List
{
    Node *first, *last;
};
```

Идея: выделить текущий указатель в структуру и создавать по надобности:

```
struct Iterator
{
    Node *curent;
};
```

Стоит задача унификации работы с контейнерными данными. Можно идти по пути унификации интерфейса, чтобы разные контейнерные классы имели один и тот же интерфейс. Желательно, чтобы каждая сущность имела свой интерфейс.

Идея простая: унифицировать работу с разными контейнерами за счёт итератора. А у итератора будет унифицированный интерфейс.

```
struct output_iterator_tag {}; // итератор вывода
struct input_iterator_tag {}; // итератор ввода
struct forward_iterator_tag {}; // однонаправленный итератора на чтение-запись
struct bi_directional_iterator_tag {}; // двунаправленный итератор на чтение-запись
struct random_access_iterator_tag {}; // произвольный доступ
```

Преимущества:

- Упрощает работу с коллекциями, позволяя обходить их элементы без знания о внутренней структуре коллекции.
- Позволяет работать с различными типами коллекций независимо от их реализации
- Позволяет реализовывать различные алгоритмы обхода коллекций

Недостатки:

- Добавление новых типов коллекций может потребовать изменения кода итератора
- Итератор может стать ненадежным, если коллекция изменяется во время обхода

Property

Не сколько паттерн поведения, сколько шаблон, который нам дает возможность формализовать свойства. В современных языках есть понятие свойство. Предположим, у нас есть какой-то класс, и у его объекта есть свойство V.

```
A obj1
obj V = 2;
int i = obj.V;
```

Доступ осуществляется через методы, один устанавливает - set(), другой возвращает - get(). Тогда мы будем рассматривать V как открытый член, но не простое данное. Если рассматривать V как объект, мы для него должны определить оператор присваивания.

Нам нужен класс, в котором есть перегруженный оператор = и оператор приведения типа. Этот перегруженный оператор = должен вызывать метод установки set, а этот get.

Удобно создать шаблон свойства. Первый параметр - тип объекта, для которого создается шаблон, а второй параметр - тип объекта, к которому приводится и ли инициализируется значение. В данном случае это целый тип. Getter - метод класса который возвращает Type, а Setter - установка для вот этого объекта.

Преимущества:

- Реализация принципа инкапсуляции
- Обеспечивает контроль доступа к данным, что позволяет избежать ошибок при работе с объектами.
- Позволяет добавлять дополнительную логику при доступе к данным, например, проверку на корректность значений или логирование

Недостатки:

- Может привести к увеличению количества кода, так как для каждого поля объекта требуется создание методов доступа.
- Может усложнить понимание кода, т.к. доступ к данным может быть скрыт в методах

```
template <typename Owner, typename Type>
class Property {
    using Getter = Type(Owner::*)() const;
    using Setter = void (Owner::*)(const Type&);
private:
    Owner* owner;
    Getter methodGet;
    Setter methodSet;

public:
    Property() = default;
    Property(Owner* const owr, Getter getmethod, Setter setmethod) : owner(owr),
methodGet(getmethod), methodSet(setmethod) {}

    void init(Owner* const owr, Getter getmethod, Setter setmethod)
    {
        owner = owr;
        methodGet = getmethod;
        methodSet = setmethod;
    }

    operator Type() { return (owner->*methodGet)(); }            // Getter
    void operator=(const Type& data) { (owner->*methodSet)(data); }    // Setter

// Property(const Property&) = delete;
// Property& operator=(const Property&) = delete;
};

class Object
{
private:
    double value;

public:
    Object(double v) : value(v) { Value.init(this, &Object::getValue,
&Object::setValue); }

    double getValue() const { return value; }
    void setValue(const double& v) { value = v; }

    Property<Object, double> Value;
};

int main() {
    Object obj(5.);

    cout << "value = " << obj.Value << endl;

    obj.Value = 10.;

    cout << "value = " << obj.Value << endl;

    unique_ptr<Object> ptr = make_unique<Object>(15.);

    cout << "value =" << ptr->Value << endl;

    obj = *ptr;
    obj.Value = ptr->Value;
}
```