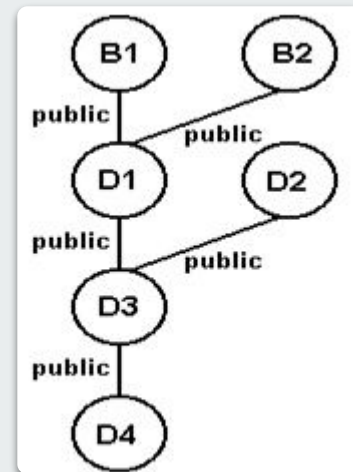


# Lesson 5-6

ООП C++.  
Классы/Объекты. Атрибуты и методы класса.  
Секции класса. Public/Private методы/атрибуты.  
ГЕТТЕРЫ/СЕТТЕРЫ.  
Конструктор. Виды конструкторов.  
Деструктор.  
Список инициализации.  
Концепции ООП. (наследование, ..)

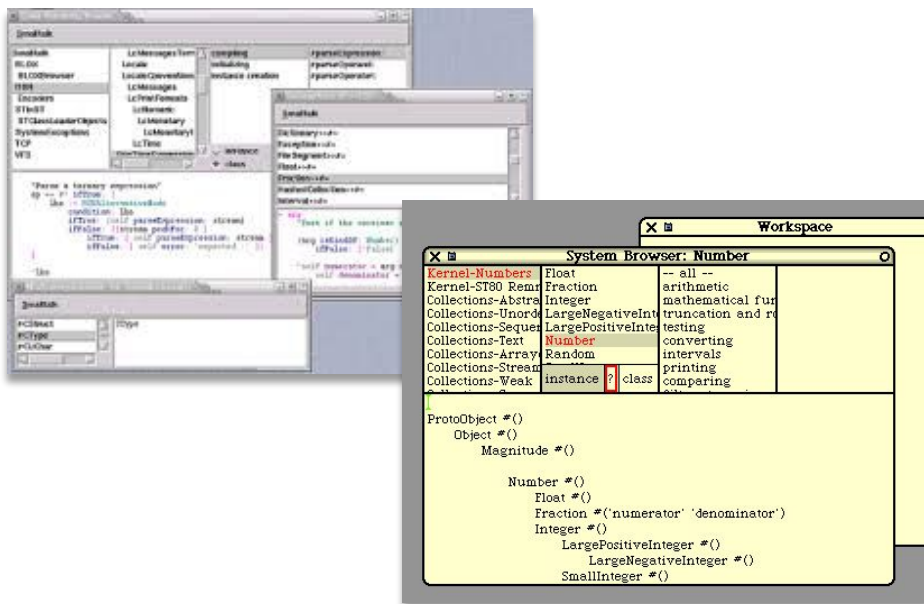


# История ООП

- 1837:** Лукас Ада сформулировал идеи алгебраического языка программирования.
- 1936:** Алан Тьюринг предложил модель универсальной машины Тьюринга, которая легла в основу теории вычислений.
- 1943-1944:** Энциклопедия Матем.Наук исследовала программы для ЭВМ.
- 1950-1960:** Эра машинных ЯП, включая Fortran, Lisp, COBOL.
- 1970-е:** Появление языков C и Pascal. Развитие идеи структурного программирования.
- 1980-е:** Программирование на уровне объектов (ООП) стало широко распространенным. ЯП, такие как **Smalltalk**, C++, и Objective-C, интегрировали концепции ООП.
- 1990-е:** Появление Java, Python, и C#, языков программирования, активно использующих ООП. Введение UML (Unified Modeling Language) для визуального представления структур и поведения систем.
- 2000-е:** Развитие фреймворков и технологий, поддерживающих ООП. Важные шаги в сторону веб-разработки.

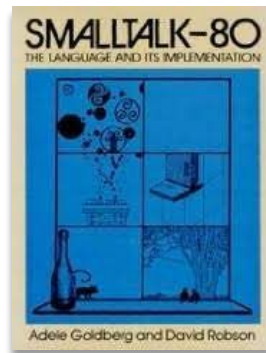


# The First OOPs Code.



# История ООП

ООП возникло как развитие идей процедурного программирования, при котором данные и методы, их обрабатывающие, формально не связаны. Взгляд на программирование «под новым углом» в 80-х годах предложили **Алан Кэй** и **Дэн Ингаллс** в языке **Smalltalk**. Здесь понятие класса стало основообразующей идеей для всех остальных конструкций.

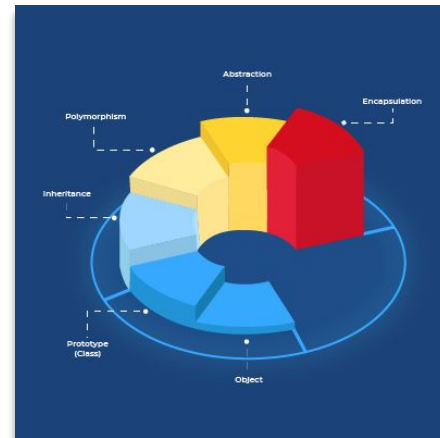


```
| s f c|
Transcript show: ' Enter a line: '.
s := stdin nextLine.
f := Bag new.
s do: [ :ch | ch isLetter
    ifTrue: [ f add: ch asLowercase]
].

1 to: 26 do: [ :i |
    c := (i+96) asCharacter.
    Transcript show: ((f occurrencesOf: c) printString, ' ')
]
```

## Цели использования ООП.

объектно-ориентированное программирование (ООП) может использоваться для решения широкого круга задач: от простых сценариев до сложных систем. С ООП проще поддерживать и развивать продукт, ООП также можно использовать для создания полноценных модулей, библиотек или даже приложений.





## Преимущества/недостатки ООП.

### Плюсы (+)

- Модульность и Повторное Использование Кода
- Инкапсуляция/Наследование/Полиморфизм
- Структурированность кода
- Удобство отладки и сопровождения
- Масштабируемость и гибкость
- Разделение программы на независимые компоненты

### Минусы(-)

- Часть программ требует больших ресурсов оборудования.
- Сложность написания кода.
- Возможно даже потеря производительности в некоторых программных реализациях.



# Fundamentals of Object Oriented Programming C++.

**ООП (Объектно-ориентированное программирование)** — методология программирования, основанная на представлении программы в виде набора объектов, каждый из которых является экземпляром определенного класса, причем классы образуют иерархию наследования.

**Классы** (явл-ся шаблонами для создания объектов) в программировании состоят из **свойств** и **методов**. **Свойства** — это любые данные, которые могут характеризовать объект класса. **Методы** — это функции, которые могут выполнять любые действия с данными класса.

*В итоге:*

1. **Методы класса** — это его функции.
2. **Свойства класса** — это его переменные (Поле (атрибут)).
3. **Интерфейс** — это набор полей и методов класса, которые могут использоваться другими классами.
4. **Объект** (экземпляр класса) — это отдельный член класса, имеющий определенные поля и методы.



# Понятие класса.

**Класс** - это шаблон для создания объектов. В классе определяются атрибуты (переменные, которые хранят данные) и методы (функции, которые могут менять данные или выполнять различные операции). Класс задается с использованием ключевого слова `class`, затем записывается имя класса (которое должно соответствовать стандартам оформления кода).

Пример создания:

```
class Car {  
    ....  
};  
  
class Animal {  
    ....  
};
```





# Понятие объекта.

**Объект** - это экземпляр класса. Когда вы создаете объект на основе класса, вы получаете экземпляр этого класса, который содержит все методы и атрибуты, определенные в классе. Для создания объекта используется ключевое слово **new**, которое выделяет память для объекта и вызывает его конструктор для инициализации. Например:

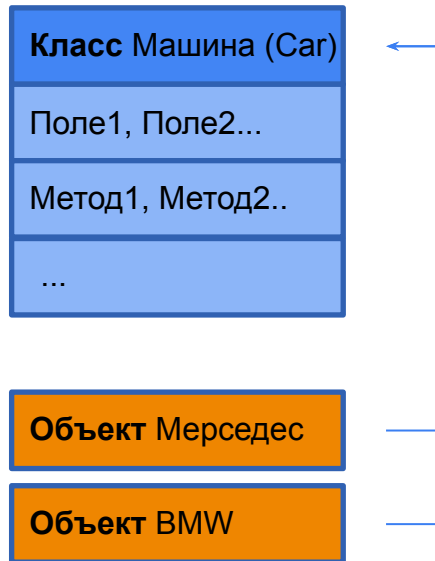
```
class A {  
    ....  
};  
  
int main() {  
    // Создание объекта и вызов конструктора  
    A* obj = new A();  
    // Удаление объекта и вызов деструктора  
    delete obj;  
    return 0;  
}
```

# Обобщение понятий класса и объекта.

**Класс** - это как чертеж или шаблон для создания чего-то нового. Давай представим, что у нас есть чертеж для изготовления машин. Этот чертеж (класс) определяет, как выглядит и как устроена машина, но сам по себе он не является машиной.

**Объект** - это конкретная машина, созданная по этому чертежу. Например, если мы используем чертеж машины "Мерседес" (класс "Машина"), то конкретная машина "Мерседес" (объект) будет создана на основе этого чертежа.

Таким образом, **класс** - это описание, как создать что-то, а **объект** - это конкретная вещь, созданная с использованием этого описания.





# Атрибуты и методы класса.

**Атрибуты (они же поля класса)** - это данные, которые хранятся в объекте (переменные, и др.). Есть возможность задавать статические атрибуты (поля) - **static** означает, что эта переменная принадлежит не конкретному объекту, а всему классу в целом.

Например, если у нас есть класс `Car` и у нас есть статическая переменная `count`, которая отслеживает, сколько машин было создано. Таким образом, даже если мы создадим много машин, `count` будет одним для всего класса `Car`.

**Методы** - это функции, которые определены в классе и могут менять данные объекта или выполнять другие операции.

```
class A {  
    int x = 1;  
  
    void foo() {  
        std::cout << "foo";  
    }  
  
    int get_v(const int value) {  
        return (x * value) + 100;  
    }  
};
```



# Вызов атрибутов/методов.

## Два основных способа:

- **Оператор . (точка):** Используется, когда объект доступен напрямую, то есть когда у нас есть сам объект, а не указатель на него.
- **Оператор -> (стрелка):** Используется, когда мы работаем с указателем на объект. Если у нас есть указатель на объект, мы используем оператор ->, чтобы получить доступ к его методам и атрибутам.

```
// через точку
```

```
A obj;
```

```
obj.foo();    // Вызов метода через объект
```

```
obj.x = 5;    // изменение значения у атрибута
```

```
// через ->
```

```
A* obj = new A();
```

```
obj->foo();    // Вызов метода через
```

```
указатель
```

```
obj->x = 5;    // изменение значения у
```

```
атрибута через указатель
```

```
delete ptr;   // Освобождение памяти
```



# Ключевое <спец> слово THIS.

**Ключевое слово `this`** используется внутри методов класса для обращения к текущему объекту, к которому принадлежит метод. Оно представляет собой указатель на объект, для которого вызван метод. Обычно используется в контексте, когда имена аргументов метода совпадают с именами атрибутов класса, чтобы явно указать, что мы обращаемся к атрибуту объекта.

```
class A {  
public:  
    int data;  
  
    void setData(int data) {  
        // Используем this для доступа к атрибуту data объекта  
        this->data = data;  
    }  
};
```



## Секции класса в C++.

В C++ секции класса используются для определения уровня доступа к его членам (полям и методам):

1. **public:** Элементы класса, объявленные в этой секции, доступны из любого места программы. Это означает, что они могут быть использованы внутри класса, в других классах и вне класса.
2. **private:** Элементы класса, объявленные в этой секции, доступны только внутри самого класса. Это означает, что они не могут быть использованы в других классах или вне класса. Это обеспечивает инкапсуляцию, позволяя скрывать детали реализации класса от внешнего мира.
3. **protected:** Элементы класса, объявленные в этой секции, доступны как из самого класса, так и из его производных классов. Таким образом, они могут быть использованы внутри класса, в его производных классах, но не вне класса или его производных.

Эти секции позволяют контролировать уровень доступа к членам класса, обеспечивая безопасность и эффективность кода.



public.

```
class A{
```

```
public:
```

```
    int x = 1;  
    void foo1();  
    void foo2();  
    ...  
    void fooN();
```

```
};
```

```
obj->|
```

```
// уда foo
```

```
inline void A::foo()
```

```
delete x
```

```
std::cout << obj->x; // вывод: 1
```

## private.

```
class A{
```

```
private:
```

```
    int x = 1;  
    void foo1();  
    void foo2();  
    ...  
    void fooN();
```

```
};
```

```
std::cout << obj->x;  
obj->foo();
```

```
public:  
    int get_x() {  
        return x;  
    }  
    void set_x(const int value) {  
        x = value;  
    }
```

```
std::cout << obj->get_x();
```

для изменения приватной переменной следует использовать спец.методы СЕТТЕРЫ/ГЕТТЕРЫ;

**SET** - изменение private переменной, **GET** - получения private переменной.



## protected.

```
class A{
```

```
protected:
```

```
    int x = 1;  
    void foo1();  
    void foo2();  
    ...  
    void fooN();
```

```
};
```

```
std::cout << obj->x;  
obj->foo();
```

*protected может использоваться  
чаще при наследовании..*



## ИТОГ по секциям.

Syntax for creating a class:

```
class <class name> : [<base class list>]
{
    <list of class members>
} [<list of variables>;
```

A class has three main access specifiers:

1. **Private** — приватный член класса, доступ к такому члену класса имеют только члены-функции того же класса, а также дружественные функции, методы и классы;
2. **protected** — защищенный член класса, доступ к такому члену класса имеют только функции-члены того же класса, дружественные функции, методы и классы, а также функции-члены классов-потомков;
3. **public** — открытый член класса, доступ к такому члену класса возможен везде, где доступен сам класс;

**!! IMPORTANT!!! Если спецификатор доступа не указан, поле метода или класса автоматически становится закрытым.**



# PUBLIC/PRIVATE методы.

## Публичные методы (public):

- Доступны извне класса и могут вызываться из других частей программы.
- Определяют интерфейс класса, который доступен для внешнего использования.
- Обычно используются для выполнения действий, которые представляют общедоступное поведение объекта.

## Приватные методы (private):

- Доступны только внутри класса и не могут быть вызваны извне.
- Используются для внутренней реализации класса и скрытия деталей реализации от внешнего мира.
- Могут вызываться только другие методы этого класса, но не из других частей программы.




## Пример. Public method.

```
class A {  
    public:  
        void foo() {  
            // Код публичного метода  
        }  
};
```



## Пример. Private method.

```
class A {  
    private:  
        void _foo() {  
            // Код приватного метода  
        }  
};
```



## PUBLIC/PRIVATE атрибуты (поля).

### Публичные атрибуты (public):

- Доступны извне класса и могут быть изменены из других частей программы.
- Представляют общедоступные данные объекта.
- Используются, когда нет необходимости в скрытии данных и при необходимости обеспечения доступа к данным извне класса.

### Приватные атрибуты (private):

- Доступны только внутри класса и не могут быть изменены извне.
- Скрыты от внешнего мира и могут быть изменены только методами этого класса.
- Обычно используются для скрытия реализации и предотвращения прямого доступа к данным.



## Пример. Public attribute.

```
class A {  
    public:  
        int publicAttribute;  
};
```



## Пример. Private attribute.

```
class A {  
    private:  
        int _privateAttribute;  
};
```

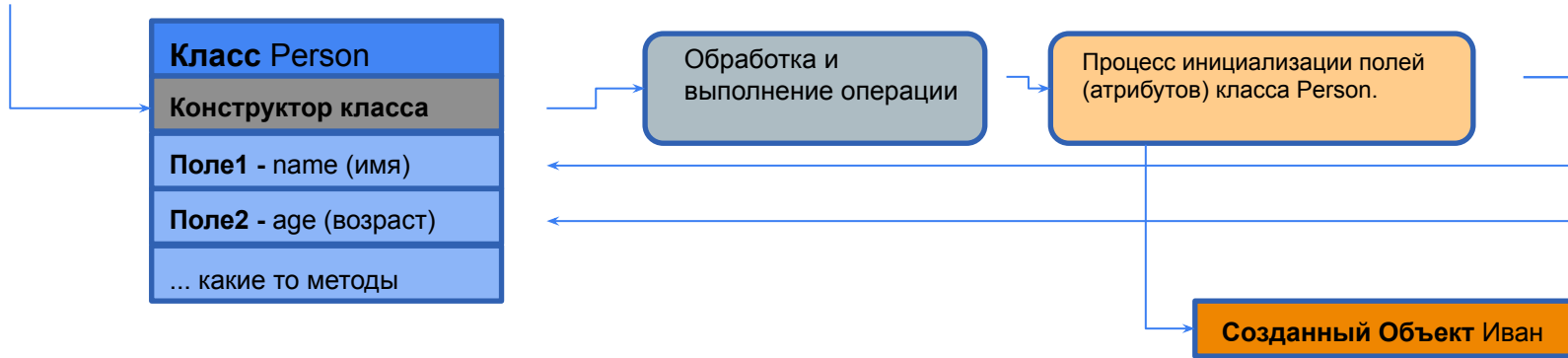
*\*m\_ иногда исп-ся для визуального оформления  
приватного поля класса*



# Конструктор класса.

**Конструктор в C++** - это специальный метод класса, который автоматически вызывается при создании объекта этого класса. Он используется для инициализации объектов и установки начальных значений их членов.

```
Person* obj = Person("Ivan", 27)
```





## Виды конструкторов.

- **Конструктор по умолчанию** - конструктор, который не принимает аргументов или принимает значения по умолчанию для всех параметров. Если вы не определите явно конструктор для класса, компилятор автоматически создаст конструктор по умолчанию.
- **Параметрический конструктор** - конструктор, который принимает аргументы и используется для установки начальных значений членов класса в соответствии с переданными значениями.
- **Конструктор копирования** - создает новый объект на основе существующего объекта. Он копирует значения членов одного объекта в другой.



## Пример. Параметрический конструктор.

```
class A {  
public:  
    A() {  
        // Конструктор по умолчанию  
    }  
};
```



## Пример. Конструктор по-умолчанию.

```
class Car {  
public:  
    std::string brand;  
    int year;  
  
    Car(const std::string& brand, int year) {  
        // параметрический конструктор  
        this->brand = brand;  
        this->year = year;  
    }  
};
```



## Пример. Конструктор копирования.

```
class A {  
public:  
    int data;  
  
    A() {  
        // конструктор по-умолчанию  
        this->data = 1;  
    }  
    A(const A& other) {  
        // конструктор копирования  
        this->data = other.data;  
    }  
};
```



# Список инициализации.

## Список инициализации в C++

представляет собой специальный синтаксис, который используется для инициализации элементов класса или базовых классов при создании объекта. Он позволяет задать начальные значения для членов класса непосредственно в момент создания объекта, что может быть более эффективным и удобным, чем установка значений по умолчанию в конструкторе.

```
class A {  
public:  
    int data;  
    double value;  
  
    // Синтаксис списка инициализации в конструкторе  
    A(int data, double value) : data(data), value(value) {  
        // Тело конструктора  
    }  
};
```



## Когда удобно исп-ть список инициализации?

1. **При инициализации элементов класса:** Это позволяет установить начальные значения атрибутов класса непосредственно при создании объекта, что может быть более эффективно и удобно, чем установка значений в теле конструктора.
2. **При инициализации базовых классов:** Когда у нас есть наследование, и мы хотим инициализировать базовый класс с определенными значениями.
3. **При работе с неизменяемыми членами класса или ссылками:** Список инициализации обязателен для инициализации неизменяемых членов класса (const) и ссылок.
4. **При работе с объектами, которые имеют конструкторы без параметров:** Если у нас есть классы с конструкторами без параметров, то использование списка инициализации позволяет инициализировать объекты этих классов при создании другого объекта.



# Деструктор класса.

**Деструктор в C++** - это специальный метод класса, который автоматически вызывается при уничтожении объекта этого класса. Он используется для освобождения ресурсов, которые могли быть выделены во время жизни объекта.

Зачем нужен деструктор:

- Освобождение ресурсов
- Предотвращение утечек памяти
- Совершение завершающих операций

```
class A {  
public:  
    ~A() {  
        // Тело деструктора  
    }  
};
```






## Ключевое <спец> слово default.

**Ключевое слово = default в C++** используется для явного указания компилятору использовать реализацию по умолчанию для определенного метода класса. Оно применяется к специальным методам класса, таким как конструкторы, деструкторы, конструкторы копирования и операторы присваивания.

Используется **= default** в следующих случаях:

- **Конструкторы и деструкторы:** Когда нам нужен конструктор или деструктор, который выполняет то же действие, что и конструктор или деструктор по умолчанию, можно использовать **= default**, чтобы явно указать это.
- **Конструкторы копирования и операторы присваивания:** Если требуется реализация по умолчанию для этих методов, то **= default** может использоваться для явного указания компилятору использовать их.



## Пример. =default.

```
class A {  
public:  
    // Конструктор по умолчанию  
    A() = default;  
  
    // Деструктор по умолчанию  
    ~A() = default;  
};
```



## CONST/STATIC методы.

**Константные методы (const)** - не изменяют состояние объекта, на котором они вызываются, и могут быть вызваны как на константных, так и на не-константных объектах класса.

**Статические методы (static)** - принадлежат классу, а не конкретным объектам, и могут быть вызваны без создания экземпляра класса. Они не могут обращаться к нестатическим членам класса напрямую.

❖ **Константные методы (const):**

- Безопасны для вызова, не меняют состояние объекта.
- Могут быть вызваны на константных объектах класса.
- Предоставляют только чтение интерфейса.

❖ **Статические методы (static):**

- Не требуют доступа к состоянию объекта.
- Оперируют общей операцией для всех объектов класса.
- Могут использоваться без создания экземпляра класса.



## Примеры. CONST/STATIC.

### 1) CONST

```
class A {  
    public:  
        int getValue() const {  
            return value;  
        }  
  
    private:  
        int value = 10;  
};
```

### 2) STATIC


```
class Math {  
    public:  
        static int add(int a, int b) {  
            return a + b;  
        }  
};
```



## Виртуальный метод класса. <virtual>

**Виртуальный метод в C++** - это метод класса, который может быть переопределен в производных классах и вызывается в соответствии с типом объекта во время выполнения (*позднее связывание или dynamic dispatch*).

Чтобы объявить метод виртуальным, используется ключевое слово **virtual** в базовом классе. Когда этот метод вызывается через указатель или ссылку на базовый класс, компилятор определяет, какая версия метода должна быть вызвана, в зависимости от типа объекта, на который указывает указатель или ссылка.



## Пример <virtual>

```
class Base {
public:
    virtual void display() {
        std::cout << "Base::display() called" << std::endl;
    }
};

class Derived : public Base {
public:
    void display() override {
        std::cout << "Derived::display() called" << std::endl;
    }
};

int main() {
    Base* ptr = new Derived(); // Создаем объект Derived, но используем указатель на Base
    ptr->display(); // Вызываем виртуальный метод display()
    delete ptr;
    ...
}
```



# OOP concepts.

## Четыре основные концепции (принципа) ООП:

1. **Абстракция** — это способ выделения набора значимых характеристик объекта путем исключения из рассмотрения несущественных. Соответственно, абстракция — это совокупность всех таких характеристик.
2. **Инкапсуляция** — это свойство системы, позволяющее объединять данные и методы, работающие с ними, в классе и скрывать от пользователя детали реализации.
3. **Наследование** — системное свойство, позволяющее описать новый класс на основе существующего с частично или полностью заимствованным функционалом. Класс, от которого осуществляется наследование, называется базовым или родительским классом. Новый класс является потомком, наследником или производным классом.
4. **Полиморфизм** — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.



## Пример инкапсуляция.

```
class BankAccount {
private:
    std::string owner;
    double balance;

public:
    BankAccount(const std::string& owner, double balance)
        : owner(owner), balance(balance) {}

    void deposit(double amount) {
        balance += amount;
    }

    void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
        } else {
            std::cout << "Insufficient funds\n";
        }
    }

    void display() const {
        std::cout << "Owner: " << owner << ", Balance: " << balance << std::endl;
    }
};
```





## Пример наследования.

```
class Animal {
public:
    virtual void makeSound() const = 0;
};

class Dog : public Animal {
public:
    void makeSound() const override {
        std::cout << "Woof!\n";
    }
};

class Cat : public Animal {
public:
    void makeSound() const override {
        std::cout << "Meow!\n";
    }
};
```

# Таблица наследования.

Спецификатор доступа для элемента в БК	Тип наследования		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	Не доступен	Не доступен	Не доступен

Такие комбинации определяют, как элементы базового класса наследуются и доступны в производных классах в зависимости от уровня доступа и типа наследования.

- Элементы класса:
  - ◆ **public**: Доступны из любого места в программе.
  - ◆ **protected**: Доступны внутри класса и его производных классов.
  - ◆ **private**: Доступны только внутри класса.
- Типы наследования:
  - ◆ **public**: Элементы базового класса становятся public в производном классе.
  - ◆ **protected**: Элементы базового класса становятся protected в производном классе.
  - ◆ **private**: Элементы базового класса становятся private в производном классе.

Таким образом:

- Элементы базового класса, объявленные как public, остаются доступными в производных классах с таким же уровнем доступа.
- Если наследование происходит как protected или private, члены базового класса становятся, соответственно, protected или private в производных классах.
- Элементы производного класса, унаследованные как protected или private, недоступны извне этого класса (т.е. извне производных классов).



## Пример полиморфизм.

```
class Shape {
public:
    virtual void draw() const {
        std::cout << "Drawing a shape\n";
    }
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a circle\n";
    }
};

class Square : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a square\n";
    }
};
```

```
int main() {
    Circle circle;
    Square square;
    Shape shape;

    Shape* shapes[] = {&circle, &square, &shape};

    for (auto s : shapes) {
        s->draw();
    }

    return 0;
}
```



## Пример абстракция. #1 <shape>

```
// Абстрактный класс "Фигура"
class Shape {
public:
    // Виртуальные методы, которые будут переопределены в производных классах
    virtual double getArea() const = 0; // Получение площади
    virtual double getPerimeter() const = 0; // Получение периметра
};
```



## Пример абстракция. #2 <circle>

// Производный класс "Круг", который наследует от "Фигура"

```
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    // Переопределение виртуальных методов
    double getArea() const override {
        return 3.14 * radius * radius;
    }
    double getPerimeter() const override {
        return 2 * 3.14 * radius;
    }
};
```



## Пример абстракция. #3 <rectangle>

```
// Производный класс "Прямоугольник", который наследует от "Фигура"
class Rectangle : public Shape {
private:
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    // Переопределение виртуальных методов
    double getArea() const override {return width * height;}
    double getPerimeter() const override {return 2 * (width + height);}
};
```



## Пример абстракция. #4 <main>

```
int main() {  
    Circle circle(5);  
    Rectangle rectangle(4, 6);  
    std::cout << "Площадь круга: " << circle.getArea() << std::endl;  
    std::cout << "Периметр круга: " << circle.getPerimeter() << std::endl;  
    std::cout << "Площадь прямоугольника: " << rectangle.getArea() << std::endl;  
    std::cout << "Периметр прямоугольника: " << rectangle.getPerimeter() << std::endl;  
    return 0;  
}
```



# Дружественные функции.

**Дружественные функции в C++** - это функции, которые имеют доступ к закрытым и защищенным членам класса, хотя они не являются членами этого класса. Это достигается путем объявления такой функции как friend внутри класса, чем обеспечивается доступ к закрытым членам класса.

используются для:

- **Улучшение эффективности.** Иногда необходимо предоставить доступ к закрытым данным класса для функций, которые имеют тесную связь с этим классом. Это может сократить необходимость создания множества геттеров и сеттеров.
- **Перегрузка операторов.** Дружественные функции часто используются для перегрузки операторов в классе. Например, для перегрузки оператора вывода (<<) для вывода объекта класса в поток вывода.
- **Доступ к приватным членам другого класса.** Иногда может возникнуть ситуация, когда один класс должен иметь доступ к закрытым членам другого класса.



# Пример friend func.

```
class B; // Предварительное объявление класса B

class A {
private:
    int dataA;

public:
    A(int _dataA) : dataA(_dataA) {}

    friend void showDataA(const A& a, const B& b); //
Объявление дружественной функции

    // Другие методы класса...
};
```

```
class B {
private:
    int dataB;

public:
    B(int _dataB) : dataB(_dataB) {}

    friend void showDataA(const A& a, const B& b);
// Объявление дружественной функции

    // Другие методы класса...
};

// Определение дружественной функции
void showDataA(const A& a, const B& b) {
    cout << "Data in A: " << a.dataA << endl;
    cout << "Data in B: " << b.dataB << endl;
}

int main() {
    A objA(5);
    B objB(10);

    showDataA(objA, objB); // Доступ к закрытым членам классов A и B
    return 0;
}
```