# OOP in C++

Ilia Nechaev

3 April 2025

# Outline

# Class vs. Struct

In C++ both `class` and `struct` are used to define user-defined types. The primary difference is the default access modifier:

- `struct`: Members are **public** by default.
- `class`: Members are **private** by default.

**Example:**

```cpp
struct Point {
    double x, y;
};

class Circle {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double getArea() const {
        return 3.14159 * radius * radius;
    }
};
```

# Class structure

```cpp
class ClassName{
public: // Any modificator can be used: public, private, protected
    TypeName fieldName;

    // Optional before TypeName: static, virtual, inline, constexpr, consteval (C++20), friend
    TypeName methodName(/* Any amount of args */);
    // Optional after function arguments: noexcept, const, volatile, override, final, =default, =delete

    // Optional before ClassName: explicit, constexpr, consteval (C++20), inline
    ClassName(/* Optional constructor args */); // Constructor
    // Optional after ClassName: noexcept, =default, =delete

    // Optional before ClassName: virtual, constexpr (C++20), inline
    ~ClassName(); // Destructor
    // Optional after ClassName: noexcept, =default, =delete, override, final

    // Here all methods and fields are still public
private:
protected:
public: // You can use the same modificator as many times as you want
}
```

# How to implement?

Define class in header file: "some_name.hpp":

```cpp
class Foo{
public:
    void print_me() const;
}
```

Implement it in source file: "some_name.cpp" (name of source file can be different)

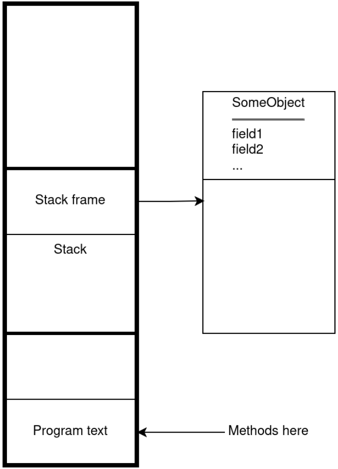
```cpp
#include <iostream>
#include "some_name.hpp"
void Foo::print_me() const {
    std::cout << "Hi\n";
}
```

Exceptions (these cases should be declared and implemented in the same file):

- Inline methods
- Constexpr and consteval methods
- Templates

# Memory layout

- If class doesn't have parent and doesn't have virtual functions, its memory layout is the same as C structure memory layout
- Pointer to object is pointer to its first field
- You are able to (but MUST NOT) get and change any object's field even if it is private and/or constant
- Methods, however, stored in program text segment and can't be simply accessed via pointer arithmetics



# Methods' names resolving

So, how does compiler understand which function to call if we have two classes with the identically named methods?

```cpp
#include <iostream>
#include <string>
struct Class1{
    const std::string name = "Class1";
    void print_me() {
        std::cout << name << "\n";
    };
};
struct Class2{
    const std::string name = "Class2";
    void print_me() {
        std::cout << name << "\n";
    };
};
int main(){
    Class1 class1;
    Class2 class2;
    class1.print_me();
    class2.print_me();
}
```

Mangling!

# Mangling and class argument

*[Wikipedia] In compiler construction, **name mangling** (also called name decoration) is a technique used to solve various problems caused by the need to resolve unique names for programming entities in many modern programming languages.*

- When you define function compiler stores it with different name than you've specified
- Function name, arguments, qualifiers (const, static, etc) and class name (if function is method) - all influence mangled function name (exact list of attributes depends on compiler's ABI)
- `Class1::print_me()` $\longrightarrow$ `_ZN6Class18print_meEv`
- But why can we use object's fields inside non-static method?
- During compilation non-static method's signature is changed in a way that object goes as first argument
  - `class1.print_me();` $\longrightarrow$ `print_me(&class1);`
  - `std::cout << name << "\n";` $\longrightarrow$ `std::cout << this->name << "\n";`

# Constructors in C++

- Constructors are special member functions used to initialize objects.
- When constructor's body is executed the object already exists.
- How to initialize fields? – Initializer list! (some complex examples later)

```cpp
class Cat{
    std::string name;
    int age;
    int amountOfLegs;
public:
    Cat() : name("Barsik"), age(1), amountOfLegs(4) {};
    Cat(std::string name) : name(name) {
        // Here age and amountOfLegs are already created, but aren't initialized (they have random value)
    };
}
```

# Types of constructors

How many types of constructors are there in C++? There are a lot of answers based on classification. We'll stick with two:

- Constructors that create object based on its components + default constructor
- Constructors that create object based on other object – copy and move constructors

```cpp
class Cat{
    std::string name;
    int age = 0;
    int amountOfFavouriteToys;
    std::string *favouriteToys; // Never do like this, use std::array or std::vector
public:
    Cat() = default; // Default constructor, deffinition will be generated automatically
                     // because of =default
    explicit Cat(std::string name); // Constructor with arguments
    Cat(const Cat &other); // Copy constructor
    Cat(Cat &&other) noexcept; // Move constructor
    // Some other methods
}
```

# Implementing constructors

```cpp
Cat::Cat(std::string name) : name(name), age(0), amountOfFavouriteToys(0), favouriteToys(nullptr) {}

Cat::Cat(const Cat &other)
: name(other.name)
, age(other.age)
, amountOfFavouriteToys(other.amountOfFavouriteToys)
, favouriteToys(nullptr)
{
    favouriteToys = new std::string[amountOfFavouriteToys];
    for (int i = 0; i < amountOfFavouriteToys; ++i)
    {
        favouriteToys[i] = other.favouriteToys[i];
    }
}

Cat::Cat(Cat &&other) noexcept
: name(std::move(other.name))
, age(other.age)
, amountOfFavouriteToys(other.amountOfFavouriteToys)
, favouriteToys(other.favouriteToys)
{
    other.amountOfFavouriteToys = 0;
    other.favouriteToys = nullptr;
}
```

# Initializer list

- As was mentioned before when constructor's body is executed all fields of object already exists.
- If field isn't listed in initializer list there are two options:
  - Default constructor is called (if field type is some class)
  - Field is uninitialized if it's primitive type (int, float, pointer)

```
class NoDefaultConstructor
{
public:
    NoDefaultConstructor()=delete; // Default constructor won't be generated (no need of =delete actually)
    NoDefaultConstructor(int a) {};
};

class WithDefaultConstructor{};

class SomeOtherClass{
    NoDefaultConstructor no_default;
    WithDefaultConstructor with_default;

    // SomeOtherClass() = default; - Error
    // SomeOtherClass() {}; - Error
    // SomeOtherClass() : with_default() {}; - Error
    SomeOtherClass() : no_default(5) {}; // OK
}
```

# Calling constructors

```cpp
1  int main(){
2      Cat cat1; // All uninitialized in default constructor fields of primitive
3                // types are in uninitialized state
4  //  Cat cat2(); it's function declaration not constructor call :)
5      Cat cat3{}; // All fields of primitive types uninitialized in default constructor  are zeroed
6      Cat *cat4 = new Cat();         // Heap allocation with default Constructor, all uninitialized fields
7                                     // of primitive types in default constructor are zeroed
8      Cat cat5("Barsik");
9      Cat cat6{"Barsik"};            // Better use this one instead of line 6
10     Cat *cat7 = new Cat("Barsik");
11     Cat *cat8 = new Cat{"Barsik"}; // Better use this one instead of line 8
12     Cat cat9(cat6);                // Copy constructor
13     Cat cat10 = cat6;              // Copy constructor
14     Cat cat11 = std::move(cat1);   // Move constructor
15     Cat cat12(std::move(cat3));    // Move constructor
16 }
```

# Simple Inheritance

C++ supports inheritance allowing a class to inherit from a base class.

```cpp
class Animal {
public:
    std::string name;
    void speak() const {
        std::cout << "Animal sound\n";
    }
};

class Dog : public Animal {
public:
    // field name and method speak are available for this class
    void bark() const {
        std::cout << "Woof!\n";
    }
};
int main(){
    Dog dog{};
    dog.name = "Sharik";
    dog.speak();
    dog.bark();
}
```

# Inheritance types

Inheritance can be:

- public – all modifiers of parent are saved in child class
- protected – all modifiers of parent are decreased by one in child class (public $\rightarrow$ protected, protected $\rightarrow$ private)
- private – all modifiers of parent are private in child class

# Dynamic Polymorphism

Dynamic polymorphism in C++ is achieved via virtual functions. Only functions marked as virtual can be overridden.

```cpp
class Base {
public:
    virtual void print() const { std::cout << "Base\n"; }
    void non_virtual_print() const { std::cout << "Base\n"; }
};

class Derived : public Base {
public:
    // override is optional, however it's good practice to have it
    void print() const override { std::cout << "Derived\n"; }
    void non_virtual_print() const { std::cout << "Derived\n"; }
};

void execute(const Base& obj) {
    obj.print();
}
void execute_non_virtual(const Base& obj) {
    obj.non_virtual_print();
}
int main(){
    const Base& obj = Derived();
    execute(obj); // Derived
    execute_non_virtual(obj); // Base
}
```

# Dynamic polymorphism. Syntax

Important keywords:

- `virtual` - mark function as virtual, so function can be overridden by derived class
- `override` - mark function as overridden. Omitting this keyword will not lead to error, but using it can prevent errors in the future:
  - Compiler will check the signature of the method marked with `override` that it corresponds the method in the base class
  - `override` shows other developers that the method is overridden
- `final` - prevent method overriding in the derived class

# Dynamic polymorphism. Virtual table

- Dynamic polymorphism is achieved via virtual table (vtable) - special per class memory region with pointers to functions
- Each object of the class which has virtual methods has a pointer (or a few in some cases) to a vtable
- When object is used via base class pointer or reference all calls to methods marked `virtual` go through vtable
- Vtable IS NOT a part of standard
- Methods that are not marked with `virtual` are called directly without using vtable

# Destructors and Virtual Destructors

In C++, destructors are responsible for cleaning up an object's resources when it is destroyed.

- When dealing with inheritance, a base class pointer may refer to a derived object.
- If the base class destructor is not declared `virtual`, only the base's destructor is called when deleting through the base pointer.
- This can lead to resource leaks or undefined behavior.

# Consequences of Non-Virtual Destructors

Consider a class hierarchy where a derived class manages a resource using a raw pointer:

```cpp
class Base {
public:
    // Non-virtual destructor - BAD PRACTICE!
    ~Base() {
        std::cout << "Base destructor\n";
    }
};

class Derived : public Base {
public:
    int* data;
    Derived() {
        data = new int[10]; // Allocate resource
    }
    ~Derived() {
        delete[] data; // Release resource
        std::cout << "Derived destructor\n";
    }
};

int main() {
    Base* obj = new Derived();
    delete obj; // Only Base destructor is called; Derived destructor is skipped!
    return 0;
}
```

# Virtual Destructors

To avoid the issues shown above:

```cpp
class Base {
public:
    virtual ~Base() {
        std::cout << "Base destructor\n";
    }
};

class Derived : public Base {
public:
    int* data;
    Derived() {
        data = new int[10];
    }
    ~Derived() override {
        delete[] data;
        std::cout << "Derived destructor\n";
    }
};

int main() {
    Base* obj = new Derived();
    delete obj; // Both Derived and Base destructors are called correctly.
    return 0;
}
```

# Abstract Classes: Overview

Abstract classes in C++ allow you to define an interface that must be implemented by derived classes. An abstract class cannot be instantiated directly.

- Contains at least one pure virtual function.
- Serves as a base for polymorphic hierarchies.
- Helps enforce a contract for derived classes.

# Pure Virtual Functions

A pure virtual function is declared by assigning `= 0` in its declaration.

```cpp
class Shape {
public:
    // Pure virtual function makes Shape an abstract class
    virtual void draw() const = 0;
};
```

- The `= 0` syntax marks the function as pure virtual.
- Any class with at least one pure virtual function becomes abstract.

# Instantiation and Derived Classes

Abstract classes cannot be instantiated. Derived classes must override all pure virtual functions.

```cpp
class Shape {
public:
    virtual void draw() const = 0;
};

class Circle : public Shape {
public:
    // Override the pure virtual function
    void draw() const override {
        std::cout << "Drawing Circle\n";
    }
};

int main(){
    // Shape shape; // Error: cannot instantiate abstract class
    Circle circle;
    circle.draw(); // Calls Circle's implementation of draw()
}
```

- Derived classes provide concrete implementations.

# Abstract Classes and Virtual Tables

- Abstract classes still have a virtual table (vtable) if they declare virtual functions.
- The vtable stores pointers to virtual functions, including placeholders for pure virtual functions.
- Calls via base class pointers or references use the vtable for dynamic dispatch.

```cpp
class Base {
public:
    // Pure virtual function, yet Base has a vtable structure.
    virtual void func() const = 0;
};
```

- Even though you cannot instantiate `Base`, its vtable is set up for derived classes.

# Key Points and Best Practices

- Use abstract classes to define clear interfaces.
- Always provide a virtual destructor in abstract classes to ensure proper cleanup.
- Derived classes must implement all pure virtual functions.
- Avoid over-abstraction; use abstract classes when a common interface is necessary.

# Multilevel Inheritance

Multilevel inheritance occurs when a class inherits from a base class, which itself inherits from another base class.

```cpp
class Animal {
public:
    Animal() { std::cout << "Animal constructor\n"; }
    virtual ~Animal() { std::cout << "Animal destructor\n"; }
};

class Mammal : public Animal {
public:
    Mammal() { std::cout << "Mammal constructor\n"; }
    ~Mammal() override { std::cout << "Mammal destructor\n"; }
};

class Dog : public Mammal {
public:
    Dog() { std::cout << "Dog constructor\n"; }
    ~Dog() override { std::cout << "Dog destructor\n"; }
};
```

# Constructors and Destructors Call Order

When creating an object in a multilevel hierarchy:

- Constructors are called top-down (base to derived).
- Destructors are called bottom-up (derived to base).

```cpp
int main(){
    Dog dog;
    return 0;
}
```

**Output:**

```
Animal constructor
Mammal constructor
Dog constructor
Dog destructor
Mammal destructor
Animal destructor
```

# Implementation Details

- Each level of inheritance adds complexity to the memory layout.
- Fields of base classes are laid out sequentially in memory.
- If virtual methods are involved, only one pointer to the vtable (per base-class chain) is added per object instance.
- However, multiple inheritance (not shown here) may introduce multiple vtable pointers.
- Non-virtual methods remain directly callable, while virtual methods require vtable lookup.

# Multiple Inheritance

Multiple inheritance occurs when a class inherits from two or more base classes.

```cpp
class Flyer {
public:
    void move() { std::cout << "Flying\n"; }
};

class Swimmer {
public:
    void move() { std::cout << "Swimming\n"; }
};

class Duck : public Flyer, public Swimmer {
public:
    void move() {
        Flyer::move();
        Swimmer::move();
    }
};
```

# Resolving Method Conflicts

When base classes have methods with identical names, you must explicitly qualify which method to call:

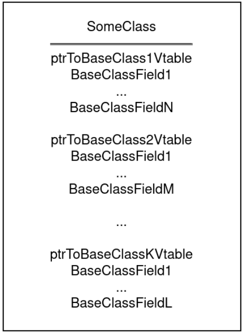
```
int main(){
    Duck duck;
    duck.move(); // Calls Duck's version

    duck.Flyer::move();  // Explicit call
    duck.Swimmer::move(); // Explicit call
    return 0;
}
```

**Output:**

```
Flying
Swimming
Flying
Swimming
```

# Memory Layout: Multiple Inheritance

- When class inherits from multiple classes, object layout includes separate sub-objects of each base class.
- Each base class sub-object is stored sequentially.
- If base classes contain virtual methods, multiple vtable pointers may exist (one per base).



# Diamond Problem

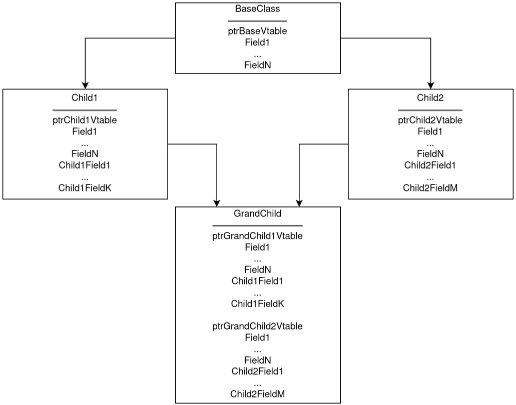
**Diamond Inheritance** creates ambiguity and redundancy when a derived class inherits from two classes that both inherit from the same base class.

```cpp
class Animal {
public:
    void speak() { std::cout << "Animal sound\n"; }
};

class Mammal : public Animal {};
class Bird : public Animal {};

class Bat : public Mammal, public Bird {};

int main(){
    Bat bat;
// bat.speak(); // Error: ambiguous call
    bat.Mammal::speak(); // Resolving explicitly
    bat.Bird::speak();
}
```

# Memory Layout: Diamond Inheritance Problem

Consider diamond inheritance without virtual inheritance:

```cpp
class Animal { public: int age; };
class Mammal : public Animal {};
class Bird : public Animal {};
class Bat : public Mammal, public Bird {};
```




Object of `Bat` contains two separate instances of `Animal`:

- Duplicate memory for each `Animal`'s fields
- Ambiguity accessing fields from `Animal`

# Virtual Inheritance

**Virtual inheritance** solves the diamond problem by ensuring a single shared base object.

```cpp
class Animal {
public:
    void speak() { std::cout << "Animal sound\n"; }
};

class Mammal : virtual public Animal {};
class Bird : virtual public Animal {};

class Bat : public Mammal, public Bird {};

int main(){
    Bat bat;
    bat.speak(); // No ambiguity now
}
```

Important:

- **Only one instance** of the virtual base class is created.
- This requires additional overhead at runtime (pointer indirection).
- Constructors of virtually inherited base classes are called explicitly from the most derived class.

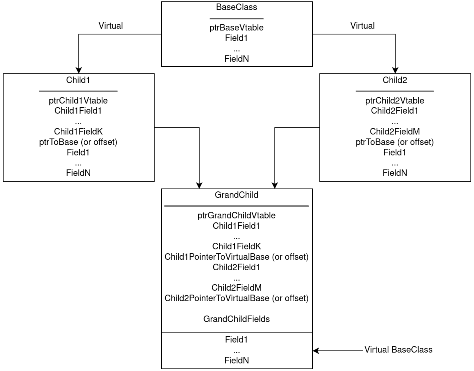
# Virtual inheritance. Syntax

```cpp
struct Tetragon {
    Tetragon(std::pair<int, int>&& a, std::pair<int, int>&& b, std::pair<int, int>&& c, std::pair<int, int>&& d)
    : a(a), b(b), c(c), d(d){};
    virtual ~Tetragon()=default;
    const std::pair<int, int> a, b, c, d;
};
struct Rectangle : public virtual Tetragon {
    Rectangle(std::pair<int, int>&& a, std::pair<int, int>&& b, std::pair<int, int>&& c, std::pair<int, int>&& d)
    : Tetragon(std::move(a), std::move(b), std::move(c), std::move(d)){};
    ~Rectangle() override=default;
};
struct Rombus : public virtual Tetragon {
    Rombus(std::pair<int, int>&& a, std::pair<int, int>&& b, std::pair<int, int>&& c, std::pair<int, int>&& d)
    : Tetragon(std::move(a), std::move(b), std::move(c), std::move(d)){};
    ~Rombus() override=default;
};
struct Square final : Rombus, Rectangle
{
    Square(std::pair<int, int>&& a, std::pair<int, int>&& b, std::pair<int, int>&& c, std::pair<int, int>&& d)
    : Tetragon(std::move(a), std::move(b), std::move(c), std::move(d))
    , Rombus(std::move(a), std::move(b), std::move(c), std::move(d))
    , Rectangle(std::move(a), std::move(b), std::move(c), std::move(d)){};
    ~Square() override=default;
};
```

# Solving Diamond Problem: Virtual Inheritance

Diamond inheritance with virtual inheritance:

```cpp
class Animal { public: int age; };
class Mammal : virtual public Animal {};
class Bird : virtual public Animal {};
class Bat : public Mammal, public Bird {};
```




Using virtual inheritance:

- Single shared `Animal` instance
- No duplication of base class memory
- Slight overhead due to additional pointers

# Implementation Details

- Multiple inheritance may introduce multiple vtable pointers if virtual methods are involved.
- Virtual inheritance adds complexity to object memory layout:
  - Additional pointers to shared base classes.
  - Increased overhead for object construction and method calls.
- Use multiple and virtual inheritance cautiously; prefer composition or abstract classes to avoid complexity.