Core C++ 2024

C++ Fundamentals: Object-Oriented Programming with C++

By Nathanel Ozeri Green

About me - Nathanel Ozeri Green

Programmer,
Currently working on my own venture

Trainer and Consultant At CLARITUNE Making Things Happen

Credit Note

Talk and slides are based on

Back to Basics: Object-Oriented Programming in C++ by Amir Kirsh - CppCon 2022

With Amir's kind permission, I've adapted and expanded upon his ideas to create this presentation.

Goals

C++ OOP Basics

We'll explore the fundamental concepts of Object-Oriented Programming (OOP) in C++.

Alternative Approaches

We'll discuss the alternatives to OOP and the tradeoffs involved.

Part 1

1. **Foundation**

2. **Crash Syntax Course**

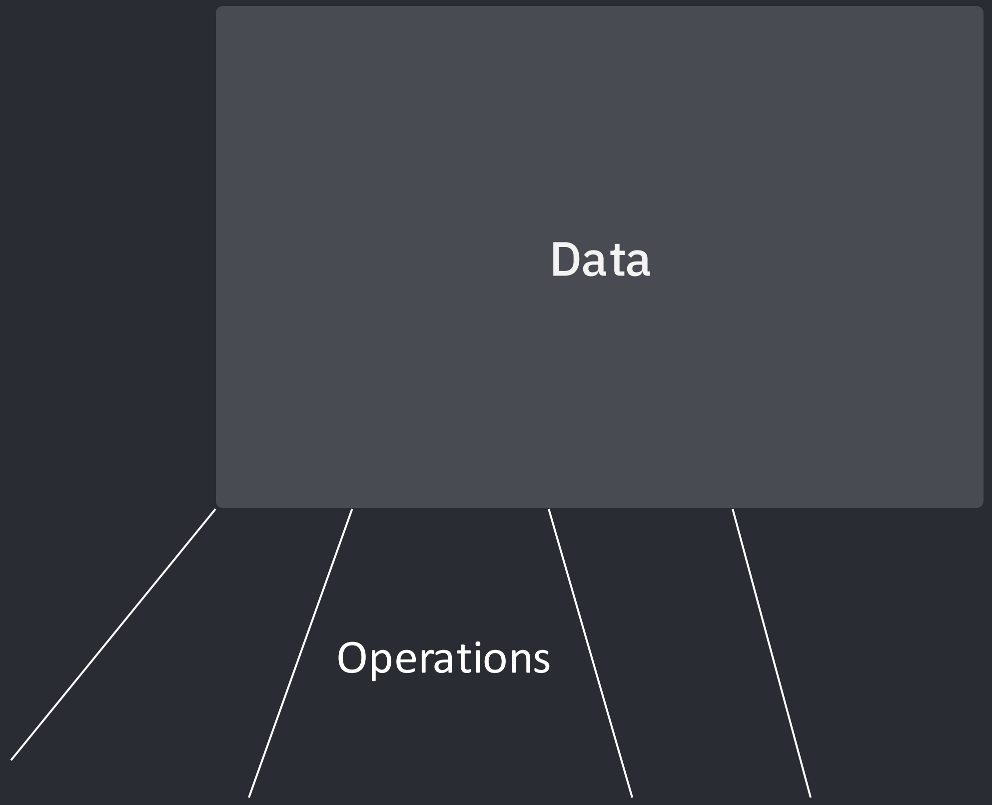
3. **Inheritance**

   We'll explore some of its concepts and how it relates to polymorphism.

Object Oriented Programming

Data

Object Oriented Programming



Classes and Objects

Class

A class is a blueprint or template for creating objects. It defines the structure and behavior of an object.

Object

An object is an instance of a class. It's a real-world entity created from the class blueprint.

```
class Widget { … }; // describes widget, nothing born yet

int main() {
	Widget w; // an actual object is created
}
```

Stick to what you do

Focus

A class should have a clear and defined purpose

Cohesion

All the members of a class should be related to its primary responsibility.

Testability

Classes with single responsibilities are easier to test

Every class takes care of its own business

Single Responsibility

A class should only have a single responsibility

A crash syntax course

Class Point

```
class Point {
        int x, y;
public:
        Point(int x1 = 0, int y1 = 0): x(x1), y(y1) {}
        void set(int x1, int y1) {
                x = x1;
                y = y1;
        }
        void move(int diffX, int diffY);
        void print() const { std::cout << "x = " << x << ", y = " << y; }
};
```

☝ don't forget the semicolon!

Class Point - usage

```
int main() {

	Point p1;
	p1.set(3, 7);
	p1.move(2, 2);
	p1.print();

	const Point p2(10, 5);
	// p2.set(10, 5);
	// p2.move(2, 2);
	p2.print();
}
```

Privileges ("Access Modifiers")

public

Public members are accessible from anywhere with proper context, like other classes or functions.

protected

Protected members are accessible only within the class itself and its derived classes.

private

Private members are only accessible from within the class definition.

Privileges - class and struct

Default Privilege

- `class` has a default private access specifier
- `struct` has a default public access specifier

Default Inheritance

- `class` inherits from its base class privately by default
- `struct` inherits from its base class publicly by default .

Data members

The data the class manages

1. **Object Data**

   Each object has its own copy of the data mebers

2. **Data Privacy**

   Data members should be private, preventing direct external access.

   ☝

3. **Initialization**

   Primitive data types must be explicitly initialized. (No default initialization)

   ☝

Member functions (= "methods")

The operations that can be preformed on an object of this type

1. **Privileges**

   Might be public/ protected or private.

2. **Scope**

   Are called with an object ("the caller")

3. **Data access**

   Can access the data members - of the calling object

4. **Size**

   Is not part of the object size

Object size

1 **Members**

The size of an object includes the size of its data members

2 **Functions**

Functions are not included in the object's size.

3 **Inheritance**

When a class inherits from a base class, its size includes the size of the base class

4 **Additional Data**

May include additional parts, e.g. pointer to vtable (discussed in another lesson)

5 **Padding**

The compiler may add padding to ensure proper alignment of data members (cppreference)

☝

header and cpp

.h file

Contains the class declaration, including the class name, member variables, and member function prototypes.

.cpp file

Contains the function definitions, where the actual implementation of the member functions is written. Includes the #include directive for the corresponding header file.

header and cpp

```
// .h file
class Point {
        int x, y;
public:
        void set(int, int); // declaration only
         void print() const { std::cout << "x = " << x << ", y = " << y; }
};

// .cpp file
#include "Point.h"
void Point::set(int x1, int y1) {
        x = x1;
        y = y1;
}
```

this

The `this` keyword is a special pointer that points to the current object.

```
struct A {
        void printAddress() { std::cout << this << std::endl; }
};

int main() {
        A a;
        std::cout << &a << std::endl;
        a.printAddress();
}
```

Constructors

Default Constructor

The compiler provides a default constructor if you don't define any.

Parameterized Constructors

Constructors can accept parameters to initialize objects with different values.

Constructor Overloading

Multiple constructors with different signatures allow flexible object initialization.

Constructor Delegation

Constructors can call other constructors in the same class (C++11).

Default Parameters

Can use default parameters - as any other method in C++

Initialization list

Used for initialization of members as well as base class(es)

Ctor init list

Efficiency

More efficient initialization, avoiding copy operations.

Correctness

Ensures data members are initialized before the constructor body executes.

Required Scenarios

Mandatory for initializing const, reference, or members with no default constructor.

Ctor init list

Init list

```
class Point {
        int x, y;
public:
        Point(int x1, int y1): x(x1), y(y1) {}
        void print() const { std::cout << "x = " << x << ", y = " << y; }
};

class Rectangle {
        Point TL, BR;
public:
        Rectangle(const Point& tl, const Point& br): TL(tl), BR(br) {}
        void print() const {
                std::cout << "TL: "; TL.print();
                std::cout << ", BR: "; BR.print();
        }
};
```

Init list

Ctor init list - Must

1

No Default Ctor

Ensures proper initialization of objects without a default constructor.

2

Const Members

Initializes const members, preventing modification after initialization.

3

Reference Members

Initializes reference members, binding them to their corresponding objects.

4

Base Class

Calls the base class constructor, ensuring correct derived object state.

Constructor delegation (C++11)

C++98

Temporary objects often lead to redundant code and performance issues.

C++11

Delegation reduces code duplication and improves initialization efficiency.

Ctor Inheritance

C++11 allows derived classes to inherit constructors from base classes.

```
class Rectangle {
        Point TL, BR;
public:
        Rectangle(const Point& tl, const Point& br): TL(tl), BR(br) {}
        Rectangle(int x1, int y1, int x2, int y2)
                   : Rectangle(Point(x1, y1), Point(x2, y2)) {}
};
```

temporary object (C++98)

ctor delegation (C++11)

C++11 also added ctor inheritance

Copy C'tor

Correct Signature

```
A::A(const A& a);
```

Use Case

- Used when creating a copy
- Called automatically when passing objects of this class by VALUE

Default Copy C'tor

If you don't define a copy constructor, the compiler automatically provides a default one. This performs a member-wise copy

☝

Copy C'tor

Problematic Signature

```
A::A(A a);
```

Why?

Assignment Operator

Signature

```
A& A::operator=(const A& a);
```

Assigning

Used when assigning an object of the same type.

Not a copy C'tor

Don't confuse with Copy C'tor! They are very similar but not the same.

Default Assignment

If you don't implement your own - you get a default one by the compiler, which does memberwise-assignment.

☝

Assignment Operator

Can we implement assignment as a global function?

```
A& operator=(A& a1, const A& a2);
```

Global Function

Implementing the assignment operator as a global function breaks encapsulation and leads to potential issues with accessing private members.

Member Function

The assignment operator should be implemented as a member function within the class. This provides the correct context for accessing and modifying the object's data.

Assignment Operator - By Value

Can we get by value?

```
A& A::operator=(A a);
```

C'tor used for Casting

Implicit Casting

C++ allows implicit casting when a constructor with a single parameter is defined.

Const Reference

Implicit casting works when passing an object by `const` reference, allowing for convenient type conversion.

By Value

Implicit casting also works when passing an object by value, enabling seamless type conversions.

By Reference

However, implicit casting doesn't work when passing an object by non-`const` reference. This prevents accidental modifications to the original object.

C'tor used for Casting

```
class A {
        int i;
public:
        A(int i1):i(i1){}
};

void f(const A& a);

// implicit casting works only for ‘const ref’ or for byval but not for byref!
int main() {
        A a1(1);
        A a2 = 2;
        f(A(1)); // works
        f((A)1); // works
        f(1);    // works!
        a1 = 3;  // works!
}
```

explicit casting

Using `explicit` promotes code clarity and reduces the risk of unintended conversions, leading to more stable and predictable code.
Use `explicit` when the c’tor doesn't get the full state! (How can you tell? Equallity)

```
class A {
        int i;
public:
        explicit A(int i1):i(i1){}
};

void f(const A& a);
```

```
int main() {
        A a1(1);    // ok
        // A a2=2;  // can't...
        f(A(1));    // ok
        f((A)1);    // ok
        // f(1);    // can't...
        // a1 = 3;  // can't...
        a1 = A(3);  // ok
}
```

const + mutable members

const

The `const` keyword prevents accidental modification of data members within a class, promoting data integrity.

`const` member functions cannot modify the object's data members, ensuring predictable behavior.

mutable

The `mutable` keyword allows specific data members to be modified within `const` functions, even though the object's state remains unchanged.

When to use?

```
class Array {
	int arr[SIZE]{};
	mutable int sum = 0;
	mutable bool isSumUpdated = true;
	void calcSum() const;
public:
	Array() {}
	// …
```

Destructor

1 Automatic Call

The destructor is automatically called when an object is destroyed.

2 No arguments

Takes no arguments, thus there is only one per class

```
~<ClassName>();
```

3 Usage

Usually used for resource de-allocations (but can actually do anything)

4 Executed Point

When an object is destroyed, its destructor is called to perform cleanup tasks

Destructor - When Object dies

1 Stack Objects

When a stack object goes out of scope, its destructor is automatically invoked.

2 Heap Objects

When a heap object is explicitly deleted using `delete`, its destructor is called before freeing the memory.

3 Global and Static

Global or static objects are destroyed when the program terminates, triggering their destructors for final cleanup.

4 Temporary Objects

Temporary objects created during expression evaluation are destroyed at the end of the statement that created them.

```
message("hello", Point(10,10));
```

Rule of Zero

It's the best if your class doesn't need any resource management

- No need for D'tor, Copy C'tor, Assignment Operator
- Defaults do the job (managed)
- [That includes defaults for move operations]

To Achieve that - Use properly managed data members - std::string, std containers, std::unique_ptr, std::shared_ptr

Rule of Three

1

Destructor Needed?

If your class needs a destructor to manage resources, take action.

2

Block Copy Operations

Immediately block the copy constructor and assignment operator.

(No TODO's)

3

Implement if Necessary

If you later determine you need the copy operations, implement them.

```
MyClass(const MyClass&) = delete;
MyClass& operator=(const MyClass&) = delete;
```

Rule of Five

If you implement or block any one of the five, you lose the defaults for the move operations

- Make sure to ask back for the defaults if they are fine

```
MyClass(MyClass&&) = default;
MyClass& operator=(MyClass&&) = default;
```

- [We are not going to cover RValue reference and Move semantics in this talk]

Inheritance

Inheritance - Why?

Code Reuse

Inheritance allows you to reuse existing code, reducing development time and effort.

We want to use both the ‘old’ class and the ‘new’ class - so we can’t change the code of the old one

Polymorphism

We want to hold and manage objects of either type without having to handle them differently (Person & Student)

Inheritance - ctor

```
class Person {
// ...
public:
        Person(const string& name);
// ...
};

class Student: public Person {
// ...
public:
        Student(const string& name): Person(name){}
// ...
};
```

calling base ctor

Inheritance - dtor

```
struct A {
        ~A() { cout << "~A" << endl; }
};

// B is inherited from A for non-polymorphic usage
struct B: public A {
        ~B() { cout << "~B" << endl; }
};

int main() {
        B b;
}
```

Output?

~B

~A

Polymorphism in C++

Polymorphism is the ability to treat different types similarly

```
class Pet {
public:
    virtual void eat(const Food& food) = 0;
        // …
};
```

```
pet.eat(food); //run time dispatching based on the calling object
```

any type of Pet

any (proper) type of Food

virtual functions

```
class Pet {
//...
public:
	virtual void makeSound() const = 0;
	virtual ~Pet() {}
};
```

If Make sound is const - it must be const in all the classes to preserve the same signature

```
class Dog: public Pet {
//...
public:
	void makeSound() const override {
		cout << "Raf raf";
	}
	~Dog() override {}
};
```

```
class Cat: public Pet {
//...
public:
	void makeSound() const override {
		cout << "mewo";
	}
	~Cat() override {}
};
```

abstract classes

```
class Pet {
//...
public:
      virtual void makeSound() const = 0;
      virtual ~Pet() {}
};
```

makeSound method is pure virtual at Pet, which makes Pet an abstract class

```
class Dog: public Pet {
//...
public:
      void makeSound() const override {
            cout << "Raf raf";
      }
      ~Dog() override {}
};
```

```
int main() {
        // Pet pet;  // Can't create
        Dog d;
        Pet* p = &d
       p->makeSound();
}
```

Usage Example - Command Pattern

- Encapsulate the information needed to perform an action
- Classical for implementing Undo/Redo stack

Advantages

- Encapsulates and hides the action itself, easier to code and maintain

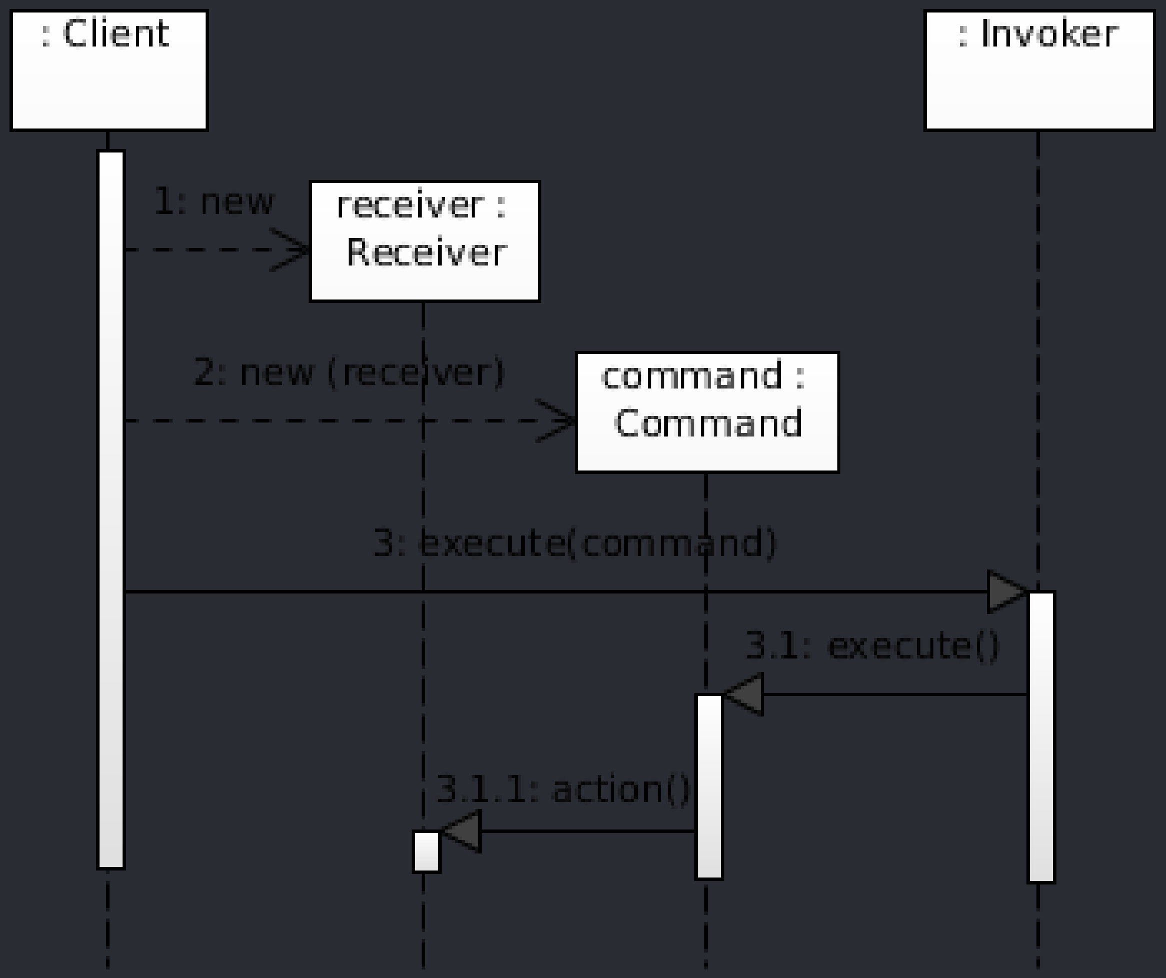


Image source

OO Low-Level Design Principles

Single Responsibility

A class should have a single, clearly defined purpose.

Break Down Complexity

Large, complex entities should be divided into smaller, more manageable classes.

Composition & Inheritance

Use composition when a class needs to use another class. Use inheritance when a class extends the functionality of another class.

Abstraction

Design your classes to be generic and reusable, focusing on interfaces rather than specific implementations.

Data Hiding

Protect your data members and member functions.

Clear API

Provide a simple and well-defined interface for your classes.

Rule of Zero

Aim to make your classes resource-free, minimizing the need for explicit memory management.

Part 2

1

Beyond the Basics

We'll explore the limitations of classic OOP in C++.

2

Alternative Approaches

We'll discuss design patterns and alternative strategies.

Beyond the "Classic" Model

Not Just OOP

C++ is not *Just* an Object Oriented Language (Bjarne Stroustrup)

Alternatives and Limitations

- When and way not to use the classic encapsulation
- When to avoid or delay inheritance

Array of Structs vs. Structs of Arrays

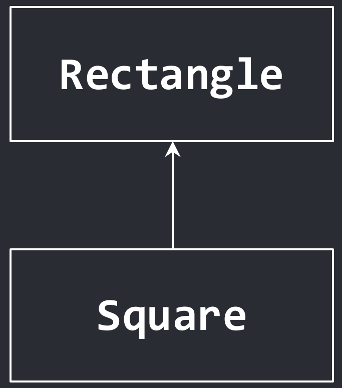
Inheritance

Inheritance is overrated

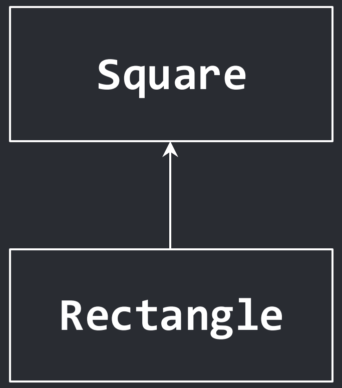
In some cases it's tricky

Sean Parent, 2013: Inheritance Is The Base Class of Evil

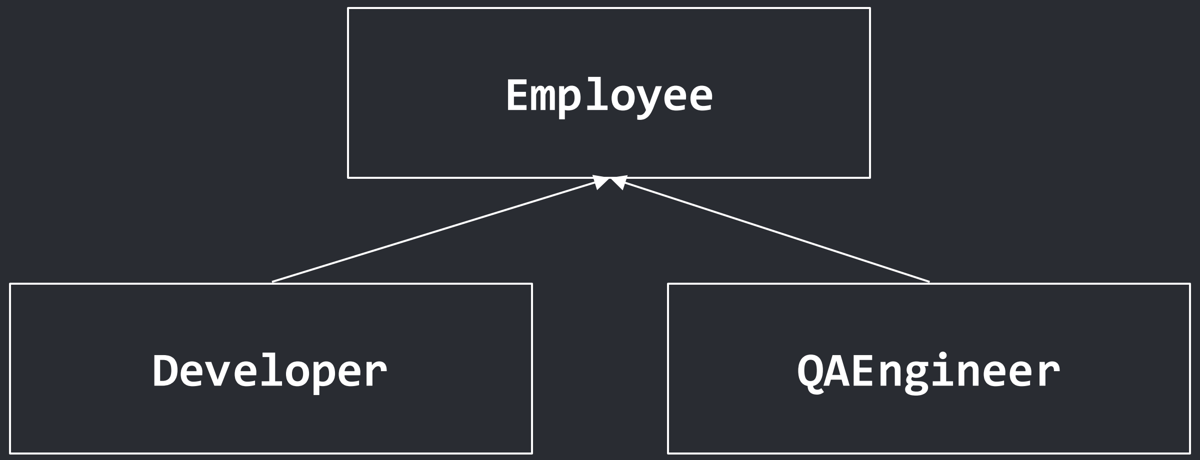
Inheritance and Liskov Substitution



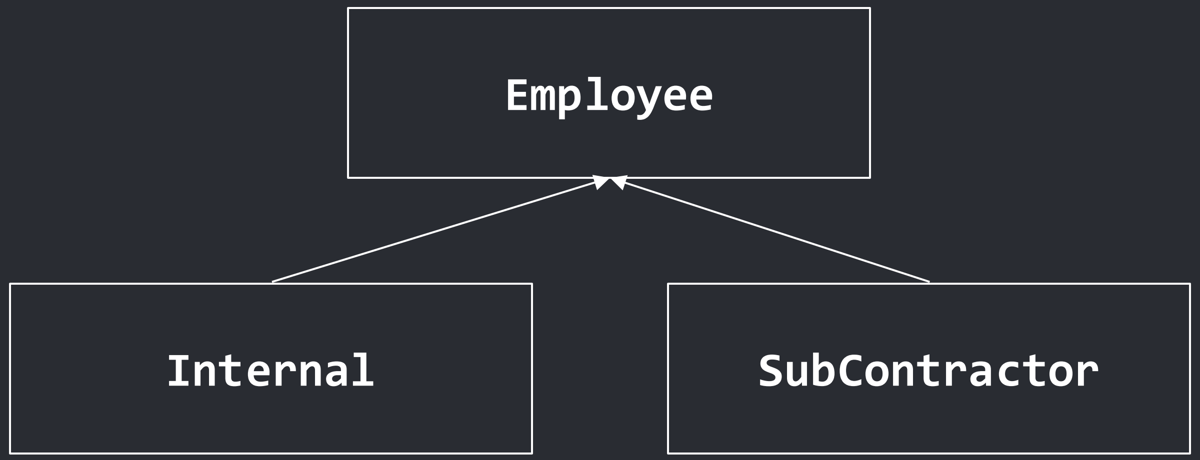
OR



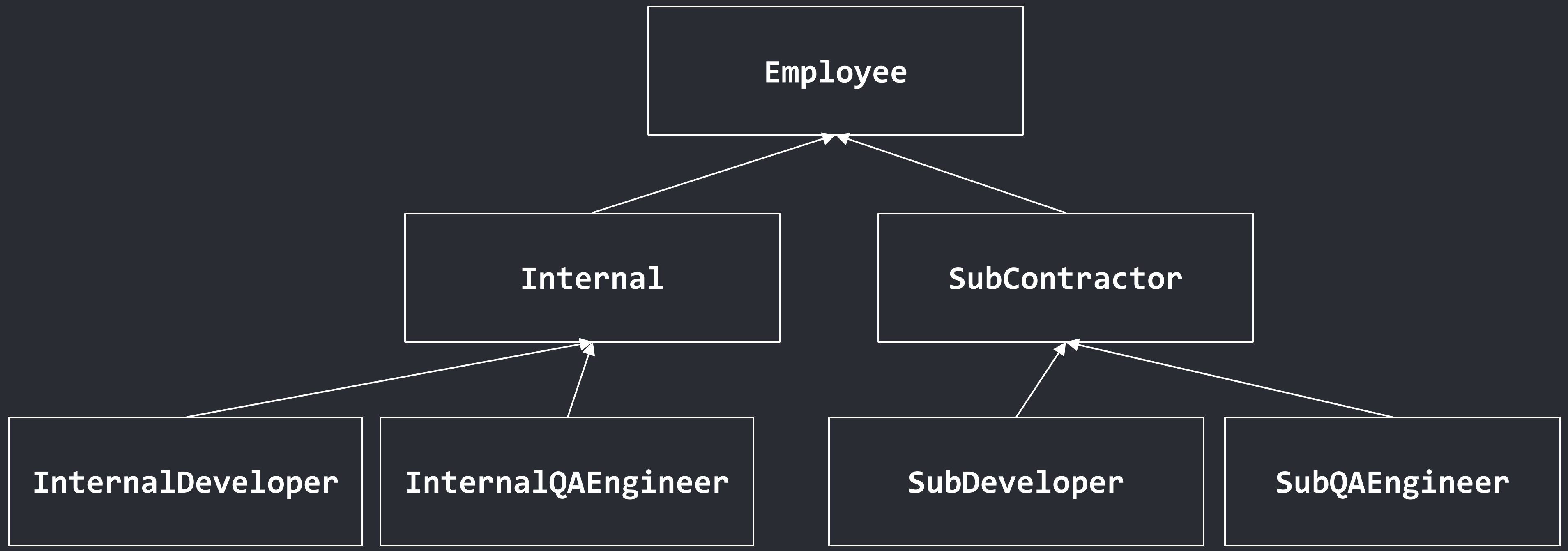
Employee Inheritance (?)



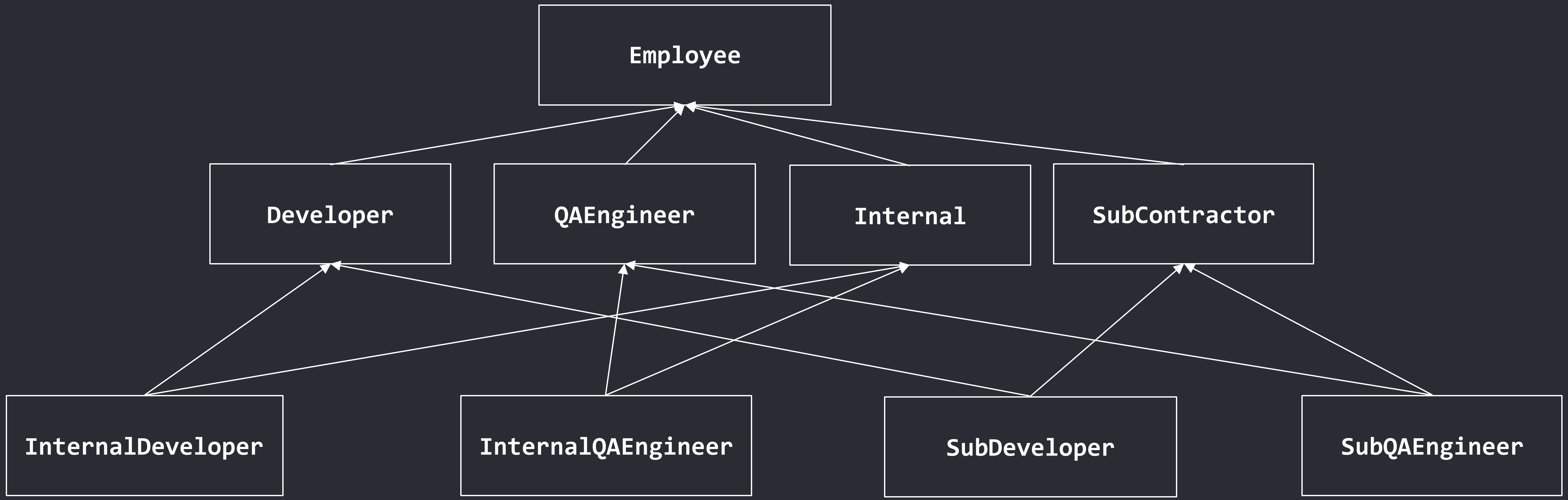
Employee Inheritance (?)



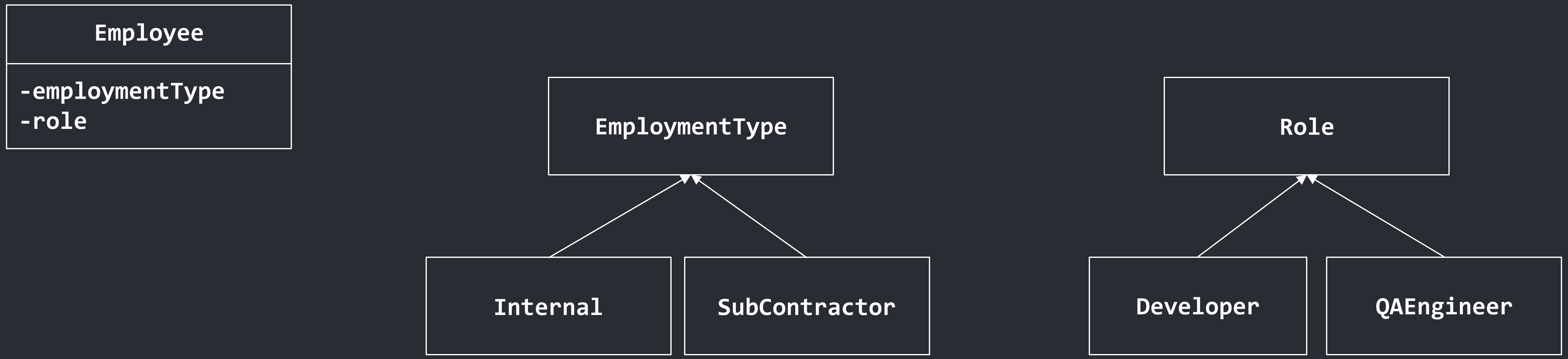
Employee Inheritance (?)



Employee Inheritance (?)



Employee Inheritance (!)



State Pattern

1

Encapsulate Behavior

Behavior is based on object's state

2

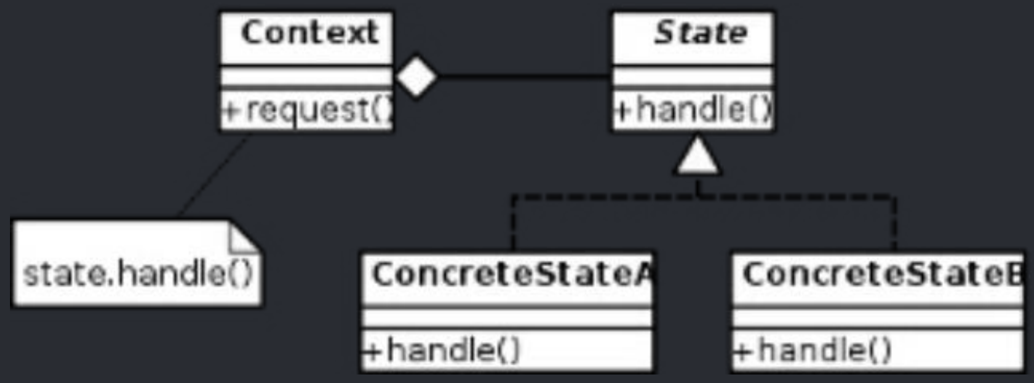
State Hierarchy

Allowing combination of behaviors per characteristic, with specific State hierarchy per each.

3

Decouple State

Separate state management from the object's core structure.



https://en.wikipedia.org/wiki/State_pattern

State Pattern

Advantages

Allowing objects to dynamically change state.

Allowing objects to have more than one state.

Strategy Pattern

1. ## Encapsulate Behavior

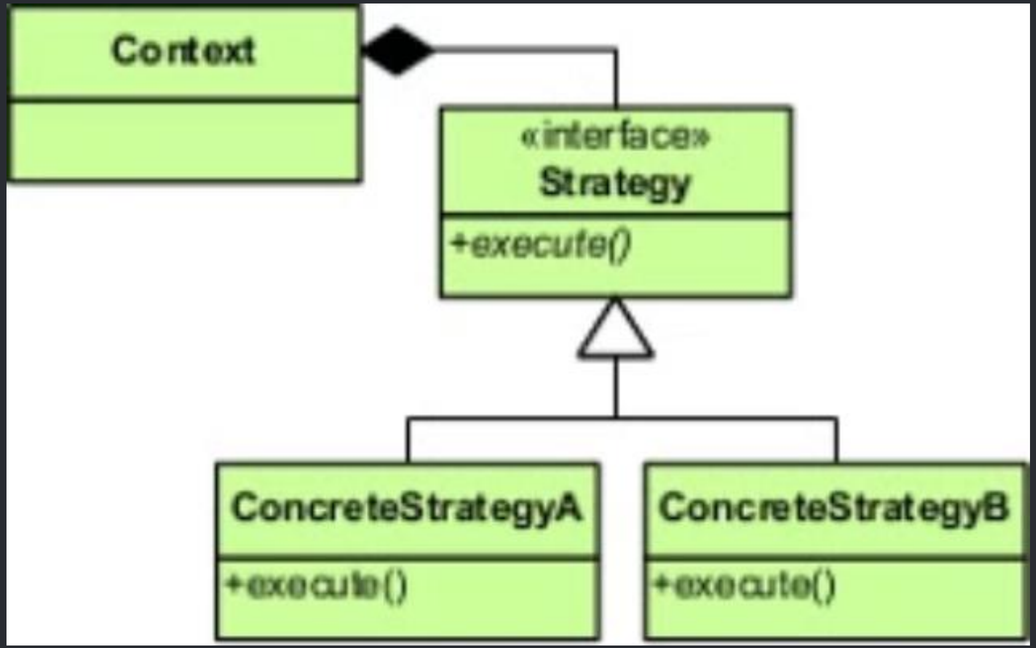
Select algorithm (strategy) to be used at runtime

2. ## Algorithms Family

Defines a family of possible algorithms for same problem.

3. ## Decouple State

Separate algorithm defenition from the object's core structure.



https://en.wikipedia.org/wiki/Strategy_pattern

Strategy Pattern

Advantages

Can be used to pick the matching/ best algorithm according to defined rules.

Algorithm selection is encapsulated and can be cached

Pet Inheritance (?)

Inheritance Model

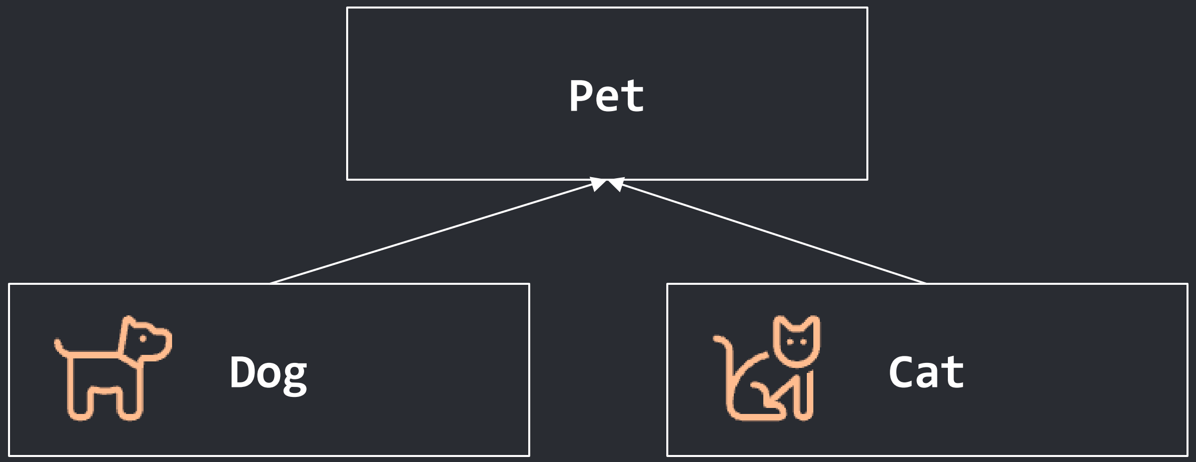
A traditional inheritance model directly inherits `Dog` and `Cat` from `Pet`.

This approach might seem simple but can become problematic when adding new pet types.

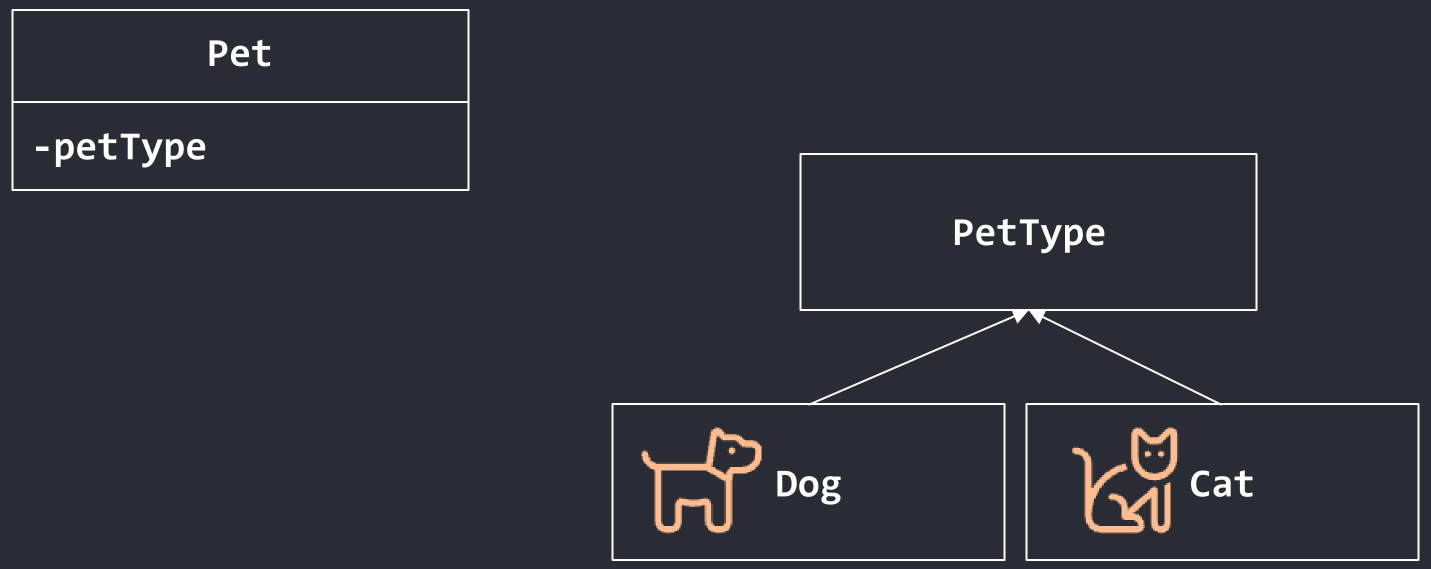
Potential Issues

Maintaining a large number of pet classes directly inherited from `Pet` can be complex.

If a new pet type needs to be added, modifications to the base class (`Pet`) might be required.



Pet Inheritance - Better with State



Issues with Inheritance

Runtime Type Changes

Changing the type of an object at runtime [QAEngineer becoming a Developer]

Inflation of Derived Classes

As the number of derived classes grows, the inheritance hierarchy can become unwieldy, requiring ways to reduce the total number of classes.

Solution: State/ Strategy Patterns

Exposing Internal Design

Forcing the user to be aware of the internal design details, such as which exact type to create, can make the code less flexible and harder to maintain.

Solution: Factory Method / Abstract Factory Patterns

Inheritance Design Principles

Make non-leaf classes abstract

[Scott Meyers]

making non-leaf classes abstract prevents them from being instantiated directly

Don't derive from concrete classes

[Herb Sutter]

don't derive from concrete classes

Make virtual function private

Inheritance Design Principles

Amir Kirsh:

same type represents all

User should work with a universal type, keep your inheritance for internal State/Strategy

stateless

Prefer to have **stateless** abstract classes (“pure interfaces”)

small and specific

Data manged by base class should be very small and very specific

Polymorphism vs. Templates

Polymorphism vs. Templates

Implement A generic 'Volume' function for any prism

1 Based on Polymorphism

2 Based on templets

Solutions

- Polymorphism
- Templates

Hierarchy for functionality?

Substitutes for Inheritance (or how to delay it)

1 **Avoiding inheritance**

Using: templates, composition, lambdas or just simple “duck type” with generic algorithms

list::iterator and vector::iterator do not (necessarily) share a base!

(as a side note => may use C++20 concepts to set expectations on type)

2 **Inheritance of smaller things**

Using State/Strategy

[Properties, Behavior, Policy]

3 **Hide your inheritance**

With a facade / Proxy of a one clear type

User should preferably work with one universal type

To Summarize

To Summarize

- **Object Oriented Programming is good**
  - This is why it’s so widely used
- Use with Care
  - Different problems may need different tools
  - Think of things that may change: additional future classes and usages

Complex Code

Classes that do more than one thing

Methods that do more than one thing

or Methods that don't use helper methods

Too much abstraction

An interface for the interface

Exposing internal design

Forcing the user to know too much, allowing abuse

Bad design

OO Low-Level Design Principles

Single Responsibility

A class should have a single, clearly defined purpose.

Break Down Complexity

Large, complex entities should be divided into smaller, more manageable classes.

Composition & Inheritance

Use composition when a class needs to use another class. Use inheritance when a class extends the functionality of another class.

Abstraction

Design your classes to be generic and reusable, focusing on interfaces rather than specific implementations.

Data Hiding

Protect your data members and member functions.
design decisions such as inheritance can also be hidden in a universal holder

Clear API

Provide a simple and well-defined interface for your classes.

Rule of Zero

Aim to make your classes resource-free, minimizing the need for explicit memory management.

Any questions before we conclude?

Thank you