Object-Oriented Programming -1

Introduction to C++

Goals Object Oriented Programming

- Programming: Not only the result is important

Goals Object Oriented Programming

- Programming: Not only the result is important
- Is the code understandable?
- Is the code extensible?

Goals Object Oriented Programming

- Programming: Not only the result is important
- Is the code understandable?
- Is the code extensible?
- Can you change parts easily?

Goals Object Oriented Programming

- Programming: Not only the result is important
- Is the code understandable?
- Is the code extensible?
- Can you change parts easily?
- Is the program fast?

Goals Object Oriented Programming

- Programming: Not only the result is important
- Is the code understandable?
- Is the code extensible?
- Can you change parts easily?
- Is the program fast?
- Can you get to your goal fast?

Object-Oriented Programming

- Model software through **objects and their interaction**

Object-Oriented Programming

- Model software through **objects and their interaction**

- **Object =** self-contained units that combines attributes (data) and methods (behavior)
- Object **attributes / elements / members** = data
- Object **methods** = functions that operate on object's internal data

Object-Oriented Programming

- Model software through **objects and their interaction**

- **Object =** self-contained units that combines attributes (data) and methods (behavior)
- Object **attributes / elements** = data
- Object **methods** = functions that operate on object’s internal data

Object-Oriented Programming

- Model software through **objects and their interaction**

- **Object =** self-contained units that combines attributes (data) and methods (behavior)
- Object **attributes / elements** = data
- Object **methods** = functions that operate on object’s internal data
- Access to methods and attributes can be public or private to objects

Encapsulation

- Hide internal details and protects the integrity of an object's state.

Encapsulation Example – Bank Account

- A bank account hides its internal database from customers.
- On an ATM you can deposit or withdraw money (public)
  but you cannot change the data / balance (private) directly.

Encapsulation Example – Bank Account

- A bank account hides its internal database from customers.
- On an ATM you can deposit or withdraw money (public)
  but you cannot change the database directly.

- Encapsulation **provides controlled access**

Encapsulation Example – Bank Account

- A bank account hides its internal database from customers.
- On an ATM you can deposit or withdraw money (public)
  but you cannot change the database directly.

- Encapsulation **provides controlled access** and **protects data**

Abstraction

- Exposing only the essential features (the public interface)
- Simplify complexity (by hiding (private) unnecessary details)

Abstraction

- Exposing only the essential features (the public interface)
- Simplify complexity (by hiding unnecessary details)

- It helps focus on what an object does rather than how it does it.

Classes and Objects

- Class is the blueprint of an object
- An object is generated by instantiating a class

Inheritance

- Inheritance allows a new class to **reuse** and **extend** an existing class.

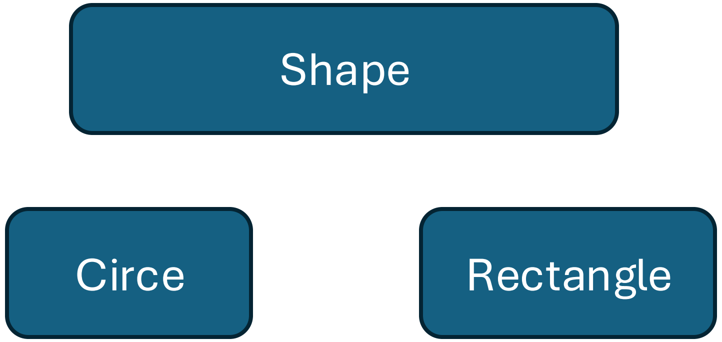
Inheritance

- Inheritance allows a new class to **reuse** and **extend** an existing class.
- It promotes code reuse and establishes relationships between classes.

Inheritance Example – Shape

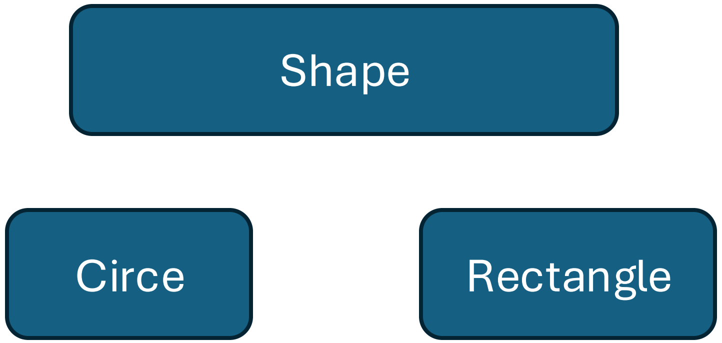
- A **Circle, Rectangle inherit from Shape** elements and methods
  - Shape: location, area, …

- Each type can reuse & customize
  - **Circle**: radius
  - **Rectangle**: length, width



Inheritance Example – Shape

- A **Circle, Rectangle inherit from Shape** elements and methods
  - Shape: location, area, …
- Each type can reuse & customize
  - Circle: radius
  - Rectangle: length, width
- Different types of inheritance (later)



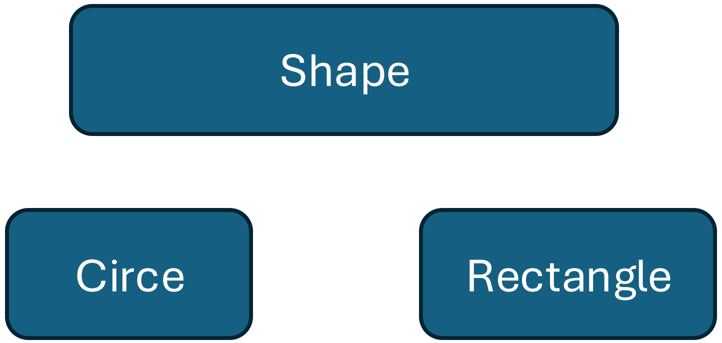
Polymorphism

- Polymorphism allows objects of different classes to be treated as objects of a common superclass.
- Polymorphism allows a single action to be performed in different ways

Polymorphism Example – Shape

- A Circle, Rectangle inherit from Shape elements and methods
  - Shape: location, **area**, **calculate_area()**

- Each type can reuse & customize
  - **Circle**: radius, **calculate_area()**
  - **Rectangle**: length, width, **calculate_area()**



- Different types of inheritance (later)

Core Principles of OOP

1. Encapsulation
2. Abstraction

3. Inheritance
4. Polymorphism

Today

- Objects: User-defined Datatypes
  - From C to C++ to C++ Classes

User-defined Datatypes - enum in C

```c
// definition:
enum _Result_ {INCORRECT, CORRECT};

// use:
enum _Result_ answer = CORRECT;
```

User-defined Datatypes - enum in C

```c
// definition:
enum _Result_ {INCORRECT, CORRECT};

// use:
enum _Result_ answer = CORRECT;
```

- Why do we need enumerations?

User-defined Datatypes - enum in C

```c
// definition:
enum _Result_ {INCORRECT, CORRECT};

// use:
enum _Result_ answer = CORRECT;
```

- Why do we need enumerations?
  - Constants with meaningful names
  - For example, useful in if-cases and switch/case

User-defined Datatypes - enum in C

```c
// definition:
enum _Result_ {INCORRECT, CORRECT};

// use:
enum _Result_ answer = CORRECT;
```

- implicit type casting
- underlying datatype: int

User-defined Datatypes - enum in C

```c
// definition:
enum _Result_ {INCORRECT, CORRECT};

// use:
enum _Result_ answer = CORRECT;
```

Better use dedicated Type

```c
// use:
enum _Result_ answer
```

```c
// use:
Result answer
```

User-defined Datatypes - enum in C

```c
// definition:
enum _Result_ {INCORRECT, CORRECT};

// use:
enum _Result_ answer = CORRECT;
```

- with typedef

```c
// definition:
typedef enum _Result_ {INCORRECT, CORRECT} Result;

// use:
Result answer = CORRECT;
```

User-defined Datatypes - enum in C++

- no typedef necessary

```
// definition:
enum Result {INCORRECT, CORRECT};

// use: (no enum needed despite missing typedef above)
Result answer = CORRECT;
```

Problem - enum in C++

- All symbols defined in global scope

```cpp
// definition:
enum Result {INCORRECT, CORRECT};

// use: (no enum needed despite missing typedef above)
Result answer = CORRECT;
```

Problem 1 - enum in C++

- All symbols defined in global scope

```
enum Assignment_1 {
    Incorrect,
    Correct
};

enum Assignment_2 {
    Incorrect,    // ❌ Error: redefinition of 'Incorrect'
    Correct       // ❌ Error: redefinition of 'Correct'
};

int main() {
    Assignment_1 a1 = Correct;
    Assignment_2 a2 = Incorrect;
    std::cout << a1 << " " << a2 << std::endl;
    return 0;
}
```

Since C++11:
Scoped enumerations with enum class

- The scope needs to be indicated

```
// definition:
enum class MenuCancelOk {CANCEL, OK};

// use: (note the scope)
MenuCancelOk button_clicked = MenuCancelOk::OK;
```

Problem 2 - enum in C++

```cpp
// definition:
enum Result {INCORRECT, CORRECT};
```

- Implicit type cast to int

Problem 2 - enum in C++

```
// definition:
enum Result {INCORRECT, CORRECT};
```

- Implicit type cast to int

Problem 2 - Examples

Assigning Invalid Values

```
enum Direction { NORTH, SOUTH, EAST, WEST };
Direction d = (Direction)42;  // Allowed
```

Problem 2 - Examples

Assigning Invalid Values

enum Direction { NORTH, SOUTH, EAST, WEST };

Direction d = (Direction)42; // Allowed

std::cout << d << std::endl; // Prints 42 -> nonsense

- If subsequent code assumes 0–3 are valid values
  -> program may misbehaves or crashs

Problem 2 - Examples

Ambiguity

- void process(int x) { std::cout << "int"; }
- void process(Color c) { std::cout << "Color"; }

- Might call int version, depending on compiler

Problem 2 - enum in C++

```cpp
// definition:
enum Result {INCORRECT, CORRECT};
```

- Implicit type cast to int

- The **compiler doesn't protect us** from doing meaningless or dangerous operations
- Can easily causes error which are often hard to find

enum class

- no <u>implicit</u> type casting

```
// definitions:
enum class MenuCancelOk {CANCEL, OK};
```

enum class

- no <u>implicit</u> type casting

```
// definitions:
enum class MenuCancelOk {CANCEL, OK};

// warning: format ‘%d’ expects argument of type ‘int’
printf("User clicked on button %d.", MenuCancelOk::OK);
```

struct

- we know this from C
- struct is a Plain-Old-Data Class (POD)

PODs

- Construct **new datatype by bundling multiple elements**
  - Elements are called member (variable) or attribute

PODs

- Construct **new datatype by bundling multiple elements**
- Different to arrays => Datatypes of elements can be different ()

PODs: Example

- Pet
  - Name (C-String (=char-Array))
  - Species (C-String (=char-Array))
  - Age (unsigned)

struct in C

```
// definition:
struct _Pet_ { // style guide requires underscores
  char name_[100]; // style guide requires trailing underscore
  char animal_[50];
  unsigned age_;
};

// make instance:
struct _Pet_ pet = {"Babe", "pig", 0};  // struct necessary in C!
```

```c
// definition:
struct _Pet_ { // style guide requires underscores
  char name_[100]; // style guide requires trailing underscore
  char animal_[50];
  unsigned age_;
};

// make instance:
struct _Pet_ pet = {"Babe", "pig", 0};  // struct necessary in C!
```

- using typedef

```c
// definition:
typedef struct _Pet_ {
  char name_[100];
  char animal_[50];
  unsigned age_;
} Pet;

// make instance:
Pet pet = {"Babe", "pig", 0};  // no struct necessary
```

struct in C++

- no typedef necessary

```cpp
// definition:
struct Pet {
  char name_[100];
  char animal_[50];
  unsigned age_;
};

// make instance:
Pet pet = {"Babe", "pig", 0};

// make another instance:
Pet g_cat = {"Ketchup", "cat", 6};
```

Alternative to struct

- **struct Pet**

```cpp
struct Pet {
  void setAge(unsigned age) {
    age_ = age;
  }
  unsigned getAge() {
    return age_;
  }

  private:
    unsigned age_;
};
```

Alternative to struct => class

- **struct Pet**
- **class Pet**

```cpp
struct Pet {
  void setAge(unsigned age) {
    age_ = age;
  }
  unsigned getAge() {
    return age_;
  }

  private:
    unsigned age_;
};
```

```cpp
class Pet {
  unsigned age_;

  public:
    void setAge(unsigned age) {
      age_ = age;
    }
    unsigned getAge() {
      return age_;
    }
};
```

Equivalent

Alternative to struct => class

- **struct Pet**
- Everything is public by default

```cpp
struct Pet {
  void setAge(unsigned age) {
    age_ = age;
  }
  unsigned getAge() {
    return age_;
  }

  private:
    unsigned age_;
};
```

- **class Pet**
- Everything is private by default

```cpp
class Pet {
  unsigned age_;

  public:
    void setAge(unsigned age) {
      age_ = age;
    }
    unsigned getAge() {
      return age_;
    }
};
```

Equivalent

Encapsulation

- Access to private members is only possible over public interface
  - With access specifier **public**
- Member functions can access all members of their own class

```
struct Pet {
  private:
    unsigned age_;
  public: // from here on, everything is public
    void setAge(unsigned age) {
      age_ = age;
    }
};

int main() {
  Pet pet{};
  pet.setAge(0); // OK
  return 0;

}
```

Setter

- We call setAge a **setter**

```cpp
struct Pet {
  private:
    unsigned age_;
  public: // from here on, everything is public
    void setAge(unsigned age) {
      age_ = age;
    }
};

int main() {
  Pet pet{};
  pet.setAge(0); // OK
  return 0;
}
```

Setter

- We call setAge a **setter**
- For writing to private attributes
- Usually 1 parameter, no return value, Name: setAttributeName

```
struct Pet {
  private:
    unsigned age_;
  public: // from here on, everything is public
    void setAge(unsigned age) {
      age_ = age;
    }
};

int main() {
  Pet pet{};
  pet.setAge(0); // OK
  return 0;
}
```

Setter

- Setters can do more
- e.g. logging or validating

```cpp
struct Pet {
  private:
    unsigned age_;
  public:
    void setAge(unsigned age) {
      if (age > 1000)
        printf("A stone is not a pet.");
      else
        age_ = age;
    }
};
```

Setter

- Setters can do more
- e.g. logging or validating

```
struct Pet {
  private:
    unsigned age_;
  public:
    void setAge(unsigned age) {
      if (age > 1000)
        printf("A stone is not a pet.");
      else
        age_ = age;
    }
};
```

```
int main() {
  Pet pet{};
  pet.setAge(0);
  pet.setAge(2000);
  printf("Your pet's age: %u", pet.age_); // POLL
  return 0;
}
```

Getter

- How to access attributes from outside? => **Getter**

```
struct Pet {
  private:
    unsigned age_;
  public:
    void setAge(unsigned age) {
      if (age > 1000)
        printf("A stone is not a pet.");
      else
        age_ = age;
    }
};
```

Getter

- How to access attributes from outside? => **Getter**
- Usually 0 parameters, attribute as return value, Name: getAttrName

```
struct Pet {
  private:
    unsigned age_;
  public:
    void setAge(unsigned age) {
      if (age > 1000)
        printf("A stone is not a pet.");
      else
        age_ = age;
    }
};
```

How do I generate an object?

The Constructor

- Special method - automatically called when the object is created
- Name of constructor = name of class
- no return value

```
struct Pet {
  private:
    unsigned age_;
  public:
    Pet() { age_ = 0; }
    unsigned getAge() {
      return age_;
    }
};

int main() {
  Pet pet{}; // constructor sets pet.age_ to 0.
  return 0;
}
```

The Constructor

- Can have multiple constructors
- Must have different parameters

```
struct Pet {
  private:
    unsigned age_;
  public:
    Pet() { age_ = 0; }
    Pet(unsigned age) { age_ = age; }
    unsigned getAge() {
      return age_;
    }
};

int main() {
  Pet pet{2}; // constructor sets pet.age_ to 2.
  return 0;
}
```

The Constructor

- Problem

```
struct Pet {
    unsigned age_;
    Pet(unsigned age) {
        bark();
        age_ = age;
    }
    void bark() { std::cout << age_; } /
};
```

- age_ is used before initialization (default value is random)

The Constructor

```cpp
struct Pet {
    std::string name_;
    Pet(std::string name) {
        name_ = name;
    }
};
```

- Problem

- Can be inefficient:
    1. Create empty string
    2. copy content to override empty string

    redundant !

The Constructor

- Better: use **initializer list**

```
struct Pet {
  unsigned age_;
  Pet() : age_{0} {}
  Pet(unsigned age) : age_{age} {}
};
```

The Constructor

- Better: use initializer list => avoids mentioned problems

```
struct Pet {
  unsigned age_;
  Pet() : age_{0} {}
  Pet(unsigned age) : age_{age} {}
};
```

- With multiple attributes:

```
Pet(char* name, unsigned age)
  : name_{name}, age_{age}
  {}
```

The Default-Constructor

- Compiler adds **Default-Constructor**, if you don't write one.
- The Default-Constructor has no parameters and initializes nothing.

```cpp
struct Pet {
private:
  unsigned age_;
};

int main() {
  Pet pet{};
  return 0;
}
```

The Default-Constructor

- Will not be generated if another constructor exists !

```cpp
struct Pet {
private:
   unsigned age_;
public:
   Pet(unsigned age) : age_{age} {}
};

int main() {
   Pet pet{}; // error: no matching function for call to 'Pet::Pet()'
   return 0;
}
```

The Default-Constructor

- We can add a Default-Constructor manually

```
struct Pet {
private:
   unsigned age_;
public:
   Pet() = default;  // similar to: Pet() {}
   Pet(unsigned age) : age_{age} {}
};

int main() {
   Pet pet{}; // OK
   return 0;
}
```

The Default-Constructor

- **Problem**: age_ is not initialized by default constructor

```cpp
struct Pet {
private:
    unsigned age_;
public:
    Pet() = default;  // similar to: Pet() {}
    Pet(unsigned age) : age_{age} {}
};

int main() {
    Pet pet{}; // OK
    return 0;
}
```

The Default-Constructor

- Possible solution: Custom constructor without parameters

```
struct Pet {
private:
   unsigned age_;
public:
   Pet() : age_{0} {}
   Pet(unsigned age) : age_{age} {}
};

int main() {
   Pet pet{}; // OK
   return 0;
}
```

The Default-Constructor

- Other possibility: Default values for attribute

```
struct Pet {
private:
   unsigned age_ = 0; // also possible: unsigned age_{0}
public:
   Pet() = default;  // similar to: Pet() {}
   Pet(unsigned age) : age_{age} {}
};

int main() {
   Pet pet{}; // OK
   return 0;
}
```