Object-oriented programming in C++

Classes, objects and polymorphism

C. Marcotte, A. Reinarz, A. Tuft, T. Weinzierl L. Morgenstern[1],
September 2023

[1] laura.morgenstern@durham.ac.uk

Motivation for OOP

The major cause of the **software crisis** *is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were* **no machines**, *programming was* **no problem** *at all; when we had a few weak computers, programming became a mild problem, and now we have* **gigantic computers**, *programming has become an equally* **gigantic problem**.

Edsger Dijkstra

Motivation for OOP

- Model systems more naturally by
  - Describing objects based on their attributes and actions
  - Describing relations between these objects
- Improving maintainability by
  - Enforcing encapsulation - via member access control
  - Enhancing code reusability - via composition and inheritance

Object-based programming vs. object-oriented programming

Object-based programming

Write program in terms of objects and operations acting on these objects. In contrast to object-oriented languages, object-based languages do not support polymorphism.

Object-oriented programming

Object-oriented programming is a programming style that focuses on the design, implementation, and use of class hierarchies. [Stroustrup, p. 11]

Definitions

Class

A class is a type. A class definition introduces a new type. [N4885, p. 253]

Object

An object is some memory that holds a value of some type. [Stroustrup, p. 40]
C++ programs create, destroy, refer to, access, and manipulate objects. An object occupies a region of storage in its period of construction, throughout its lifetime, and in its period of destruction. In C++ an object has a size, storage duration, lifetime, type, value and, optionally, a name. [N4885, p. 58]

Components of a class

- Class is custom data type that consists of member variables and methods

→ Encapsulates state and behaviour

- Member variables:
  - Capture the state of an object
  - Each object created has a copy of each member variable but not (necessarily) its value (except for `static` member variables)
- Member functions (methods):
  - Member functions can only be called on objects of their corresponding `class` type
  - Member functions can be seen as function pointers that are valid class-wide

Class definition

```
class ClassName {
    public:
        ClassName(parameters);
        ReturnType method1(parameters);
        ReturnType method2(parameters);
        ReturnType method3(parameters);
    private:
        Type member_variable1;
        Type member_variable2;
};
```

Access specifier

An access specifier specifies the access rules for members following it until the end of the class or until another access specifier is encountered. [N4485, p. 290]

public

A public member can be named anywhere without access restriction.

private

A private member can be named only by members and friends of the class in which it is declared. [N4885, p. 288]

Class definition, instantiation and usage

Example

Definition:

```
class ComplexNumber {
    public:
        ComplexNumber(double real, double imag) :
            real_(real), imag_(imag) {}
        double abs() { /* ... */ }
        double conj() { /* ... */ }
    private:
        double real_;
        double imag_;
};
```

Instantiation and usage:

```
ComplexNumber a(1, 2);
a.abs();
//alternative way in C++11 and onwards "uniform initialization"
ComplexNumber b{1, 2};
```

`struct` vs. `class` in C++

- Both are used to create custom types
- Both can have member variables and member functions

→ Identical, with the only difference being that
default accessibility in `structs` is `public`,
whereas in `classes` default accessibility is `private`

→ Coding convention: `structs` are typically used to aggregate data (and data only), while classes are used to bundle data and associated operations

→ Side note: the C++ standard uses the term *class* type as superordinate for `structs`, `unions` and `classes` at times [N4885, p. 255]

Special member functions

Special member functions ensure that custom types can be used like built-in types, and are required for resource management.

Constructor

Constructor is a special member function of a class that is used to initialize objects of its class type.

- Constructor does not have a name but is identified by the class name
- Invoked when initialization takes place and can only be called once in lifetime of an object
- Constructors that may be called without any argument are default constructors (compiler-generated)
- Constructors that take another object of the same type as the argument are copy constructors and move constructors

Example:

```
class X
 {
     public:
        X(int a, int b) : a_(a), b_(b)
        {}

     private:
         int a_;
         int b_;
 };
```

- Member variables are initialized in the order of their declaration, i.e. **not** the order in which they occur in the member initializer list
- Good practice: member initializer list should have the same order as declarations to avoid confusion

Example:

```
class X
 {
     public:
        X(int size, int* array)
        {
             size_ = size;
             array_ = new int[size_];
        }
     private:
         int size_;
         int* array_;
};
```

→ Good practice: prefer member initializer lists over initialization via assignment for performance

- **Copy constructor**: initializes object from another object of the same type during
- Initialization such as `T a(b);` or `T a = b;` where `b` is of type `T`
- Function argument passing: `f(a);` where `a` is of type `T` and `f` is `void f(T t);`
- Function return: `return a;` inside a function such as `T f()`

```
// constructor
ComplexNumber(double imag, double real) :
    imag_(imag), real_(real) {}
// copy constructor
ComplexNumber(Complex& other) :
    imag_(other.imag_), real_(other.real_) {}
```

→ Why can we access the private member variables of `other`?!
→ Access control in C++ works on a per-class basis, not a per-object basis.

Destructor

A destructor is a special member function that is called when the lifetime of an object ends. The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.

- Member variables are destructed in reverse order of their declaration
- Custom destructor is required when class manages resources with dynamic storage duration (keyword `new` or function `malloc()`)

Member functions

Special member functions: destructor definition

```
class X
 {
     public:
        X(int size, int* array)
        {
             size_ = size;
             array_ = new int[size_];
        }
        ~X()
        {
         delete[] array_;
        }
     private:
         int size_;
         int* array_;
 };
```

- **Constructor**: initializes an object
- **Destructor**: frees resources an object may have acquired
- **Copy constructor**: initializes object from another object of the same type
- **Copy assignment operator**: already initialized object is assigned new value from another existing object
- **Move constructor**: initializes object from another object of the same type by swapping resource pointers
- **Move assignment operator**: replace object with another object by swapping resource pointers

→ Can be compiler-generated (especially for trivial classes)

→ Great article on the *rule of zero*: `https://www.fluentcpp.com/2019/04/23/the-rule-of-zero-zero-constructor-zero-calorie/`

Member functions

`default`-generation of special member functions

```
class X
 {
 public:
    X() = default; //constructor

    X(X const& other) = default; // copy constructor
    X& operator=(X const& other) = default; // copy assignment op.

    X(X&& other) = default; // move constructor
    X& operator=(X&& other) = default; // move assignment operator

    ~X() = default; // destructor
 };
```

Member functions

`default`-generation of special member functions

Equivalently:

```
class X
 {

 };
```

A word of warning: the most vexing parse

(default-generated) constructor with empty parameter list

What's the output of the following code?

```
class X
{
  public:
      X(){ std:: cout << "Constructor of X" << std::endl;}
};

int main() {
     X x();
}
```

A word of warning: the most vexing parse

(default-generated) constructor with empty parameter list

What's the output of the following code? → Nothing!

```
class X
{
  public:
      X(){ std:: cout << "Constructor of X" << std::endl;}
};

int main() {
     X x();
}
```

→ Syntax of variable initialization and function declaration can be ambiguous

→ In case of ambiguity, the compiler always chooses function declaration

→ `warning: empty parentheses were disambiguated as a function declaration [-Wvexing-parse]`

- Getter: public member function that allows caller to get the value of a member variable
- Setter: public member function that allows caller to set the value of a member variable
- Used to control access to member variables
- Use with care: can be short of breaking encapsulation when over-used or implemented too broadly

Member functions

Example: setters and getters

```
class ComplexNumber {
    public:
        ComplexNumber(double real, double imag) :
            real_(real), imag_(imag) {}
        double abs() { /* ... */ }
        double conj() { /* ... */ }
        double real() {return real_;} // getter
        double imag() {return imag_;} // getter
        void real(double real) {real_ = real;} // setter
        void real(double ima) {imag_ = imag;} // setter
    private:
        double real_;
        double imag_;
};
```

Implicit object parameter

Implicit object parameter

The object on which a member function is called.

- When calling `myObject.do_something()` `myObject` is the implicit object parameter
- Allows the member function to access the object it was called on

this pointer

- `this` is an expression whose value is the address of the implicit object parameter
- Syntax: `this->member`
- Typical use cases:
  - Return pointer or reference to `this` from a member function
  - Resolution of name ambiguity between local variables and member variables

```
class A
{
  public:
      A(int N)
      {
          //equivalent to "a = new int[N]"
          this->a = new int[N];
      }
  private:
      int N;
      int* a;
};
```

Encapsulation

Encapsulation

Encapsulation refers to the grouping of data and methods that operate on this data to protect the data from erroneous usage.

- Encapsulation can be enforced in `C++` via
- Class types (unions, structs classes) $\rightarrow$ grouping
- Access specifiers such as `private` $\rightarrow$ information hiding
- Limitation of modifiability $\rightarrow$ `const` correctness

const type qualifier

The const qualifier defines that the type is constant and declares the initialized data object as immutable. The value of the object is set at initialization.

const-qualified member functions

When following a member function's parameter list, const specifies that the function does not modify the implicit object parameter.

- → Trying to modify a const qualified object leads to compilation error
- → Helps to enforce encapsulation by stopping developers from erroneously modifying variables
- → Good practice: const correctness - apply const whenever an object should not be modified

Encapsulation

const type qualifier

```
const int a = 5;
int const b = 6;
const int* e = &a;
int f = 9;
int * const g = &f;
int const * const h = &b;
const int& func1();
```

```
const int a = 5; // a is an int that is const
int const b = 6; // b is a constant int
const int* e = &a; // e is a ptr to an int that is const
int f = 9;
int * const g = &f; // g is a constant ptr to a (non-const) int
int const * const h = &b; // h is a constant ptr to a const int
const int& func1(); // returns a ref to an int that is const
```

- const applies to the thing left of it. If there is nothing on the left then it applies to the thing right of it.
- Bottom line: types are read from right to left
- Fun fact: there are no "const references" (since references are not objects)

Encapsulation

const-qualified member functions

```
class ComplexNumber {
    public:
        ComplexNumber(double real, double imag) :
            real_(real), imag_(imag) {}
        double abs() { /* ... */ }
        double conj() { /* ... */ }
        double real() const {return real_;} // getter
        double imag() const {return imag_;} // getter
        void real(double real) {real_ = real;} // setter
        void real(double ima) {imag_ = imag;} // setter
    private:
        double real_;
        double imag_;
};
```

`friend` functions

`friend` declaration

The `friend` declaration appears in a class body and grants a function or another class access to private and protected members of the class where the `friend` declaration appears. Friend functions are non-member functions.

- Function overloading: definition of multiple functions with the same name but different parameter types

```
void function1(int a);
void function1(float a);
void function1(double a);
```

Operator overloading

Operator

Operators are functions with special function names prefixed with `operator` as defined by the C++ standard.

- Arithmetic operators: `+, -, *, %, /`
- Increment and decrement: `++,--`
- Compound assignment operators: `+=, -=, *=, /=, ...`
- Logical operators, comparison operators, bitwise operators, ...

→ Operator overloading means to overload any of these operators for custom types

Binary operator

A binary operator is an operator that acts on two operands.

- Two options to overload binary operators:
  - As member function with a single argument (with lhs being the implicit object parameter)
  - As freestanding (potentially `friend`) function with two arguments
- Good practice:

→ overloaded operators for custom types should behave similar to operators for built-in types

→ for commutative operations, operator overloads are expected to preserve commutativity

Operator overloading

Overloading binary operators via `friend` function

```
class ComplexNumber {
    public:
        /* ... */
        friend ComplexNumber operator+
            (const ComplexNumber& rhs, const ComplexNumber& lhs);
    private:
        double real_;
        double imag_;
};
ComplexNumber operator+
    (ComplexNumber const& rhs, ComplexNumber const& lhs)
{
    ComplexNumber result;
    result.real_ = rhs.real_ + lhs.real_;
    result.imag_ = rhs.imag_ + lhs.imag_;
    return result;
}
```

Operator overloading

Overloading binary operators via member function

```
class ComplexNumber {
    public:
        /* ... */
        ComplexNumber operator+(ComplexNumber const& rhs) const
        {
            ComplexNumber result;
            result.real_ = real_ + rhs.real_ ;
            result.imag_ = imag_ + rhs.imag_;
            return result;
        }
    private:
        double real_;
        double imag_;
};
```

- **Copy assignment operator**: already initialized object is assigned a new value from another existing object
- Generally, assignment operators should return by reference, while arithmetic operators should return by value

```
// usage
ComplexNumber a(1,1), b(2,2);
a = b; // equivalent to a.operator=(b);

// copy assignment operator
ComplexNumber& ComplexNumber::operator=(ComplexNumber const& other)
{
    real_ = other.real_;
    imag_ = other.imag_;
    return *this; // Why is this required?
}
```

Operator overloading

Last but not least: operators that cannot be overloaded

- Operators `.` `(dot)` `::` `?:` `sizeof` can - syntactically - not be overloaded
- Overloading these would either break basic language rules or massively complicate programming
- More info:
  `https://isocpp.org/wiki/faq/operator-overloading#overload-dot`

Composition

Composition

Composition is the concept of building new data types by combining existing data types. Composition models *has-a relationships*.

- Specifically, `class` types can consist of member variables with other `class` types
- Typically used when you want the features of an existing class inside your new class but not its interface
- A book *has a* set of pages
- A `std::vector` *has a*n iterator

Polymorphism

Ad-hoc polymorphism

Write multiple implementations of a function, one for each type you wish to support.
→ Function and operator overloading

Subtype polymorphism

Relate data types by some substitutability. Write a function for a supertype instance such that all subtypes can use it.
→ Inheritance

Parametric polymorphism

Write a single implementation of a function that applies generically and identically to values of any type.

Inheritance

Inheritance

Inheritance is the concept of building new data types by deriving from existing data types. Inheritance models *is-a relationships*.

- `class` types can inherit of member variables and methods from other `class` types
- Typically used when you want to reuse the interface of an existing class inside your new class but not (all of) its implementation
- A planet *is a* celestial body; a sun *is a* celestial body

Inheritance

Example

```
class Base
{
    public:
        Base(){ std::cout << "Constructor of Base class" << "\n";}
        void function1()
        { std::cout << "function1" << "\n"; }
};

class Derived : public Base
{
    public:
        Derived(){ std::cout << "Constructor of Derived class" << "\n";}
};
```

What is the output of the following code?

```
int main()
{
    Base b;
    Derived d;
    b.function1();
    d.function1();
}
```

Inheritance

Example continued

```
int main()
{
    Base b;
    Derived d;
    b.function1();
    d.function1();
}
```

Output:

```
Constructor of Base class
Constructor of Base class
Constructor of Derived class
function1
function1
```

Access specifier

An access specifier specifies the access rules for members following it until the end of the class or until another access specifier is encountered. [N4485, p. 290]

protected

A member can be named only by members and friends of the class in which it is declared, *by classes derived from that class*, and by their friends. [N4885, p. 288]

Virtual functions

A virtual function is a member function whose behavior can be overridden in derived classes.

Runtime polymorphism

Overriding base class functions in derived class

```
   class Base
{
    public:
        Base(){ std::cout << "Constructor of Base class" << "\n";}
        virtual void function1()
        { std::cout << "function1 in Base" << "\n"; }
};

class Derived : public Base
{
    public:
        Derived(){ std::cout << "Constructor of Derived class" << "\n";}
        void function1(){ std::cout << "function1 in Derived" << "\n"; }
};
```

Runtime polymorphism

Abstract class

- Virtual operations can be made abstract by appedning =0 to the declaration
- A class with at least one abstract method is an abstract class.
- We cannot instantiate abstract classes.
- This way, we enforce subclasses to implement them.
- Class with solely abstract operations is called an interface.

```
class Base
{
    public:
        Base(){ std::cout << "Constructor of Base class" << "\n";}
        virtual void function1() = 0;
};
```

Multiple inheritance

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class D : public A, public B, public C { /* ... */ };
```

→ Combines attributes and methods from all Base classes into a new class type

→ Can model aspect-oriented programming and cross-cutting concerns (should mainly be applied for inheriting interfaces, though)

Liskov substitution principle

Objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

- Holds only in case of public inheritance (*is-a* relationships)
- Not applicable to interfaces since interfaces cannot be instantiated (i.e. there is no way to replace something that does not exist)

Summary Inheritance

- Inheritance represents is-a relations.
- It enables us to model the real world more accurately.
- It helps us to avoid code duplication.
- We can write functions for a whole set of different class types.