Object-Oriented Programming

Amanda Araujo

Abstract

A compilation of the basic concepts of Object-Oriented Programming (OOP) and implementations with the C++ programming language (`.gpp`). The *4dummies* guide I needed to learn this new computing paradigm and language, while keeping my sanity. We ought to be our own best collaborator, and I think the future me would appreciate it. It's an ongoing and continuous process*.

Contents

*Codes available on GitHub in the repository: poo-cpp.

1 Introduction

1.1 Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **objects**, which can contain *data* and *code*: data in the form of fields (called *attributes* or properties), and code in the form of procedures (known as *methods*). In OOP, computer programs are designed by making them out of objects that interact with one another.

Figure 1: Let's take a leap into this new paradigm of programming.

1980s - The rise of Object-Oriented Programming*

"In the 1980s, object-oriented programming (OOP) gained prominence with the introduction of languages like Smalltalk and C++. OOP introduced the concept of 'objects'—data structures that combine both data and methods. This shift in programming approach improved code modularity, reusability, and maintenance, setting the stage for the development of more complex and scalable software systems."

1.2 Procedural *vs.* Object-Oriented Programming Languages

So far, what we've seen coming from a background of programming in FORTRAN, C, and focusing on tasks and functions in Python, resume to a way of thinking and designing code to what it is called *Procedural Programming*. It consists of a programming model derived from *structured programming* (programming without the use of `goto`), based on the concept of calling procedures, also known as routines, subroutines

*Ref. Coding milestones - Nomical Connecting worlds

or functions. Now the focus is shifted from actions, tasks, *functions* to agents, *objects*.

A key advantage of OOP over procedural programming is that OOP provides a clear structure and allows reusability with less code. The principal differences between them can be seen in (Table 1).

Procedural	*Object-Oriented*
Focus on *processes*	Focus on *objects*
Program divided into small parts called *functions*	Program divided into small parts called *objects*
Top-down approach	*Bottom-up approach*
Function over data	Data over function
Adding new data and functions is not easy	Adding new data and functions is easy
No overloading	Overloading is supported
No access specifier	Has access specifier
No concept of data hiding and inheritance	Concept of data hiding and inheritance used
No proper way of hiding data, *less secure*	Provides data hiding, *more secure*
Designing medium-sized programs	Designing large and complex programs
Concept of procedure abstraction	Concept of data abstraction
Code reusability absent	Code reusability present
FORTRAN, C, BASIC, Pascal	C++, Python, Java, C#, Dart

Table 1: Procedural programming *vs.* Object-Oriented programming.

1.3 Brief History of Programming Languages

Generations in programming languages are related to the evolution of both hardware and software progressing exponentially over the past half-century, thus allowing different levels of conceptual development and capabilities to be accomplished. In Computer Science history, there are five reconized generation languages (GLs):

- First-Generation Programming Languages (1GL)

  The beginning of it all: Machine language 01s

Low-level, no compiler, no assembler

Very fast, inputted directly to the computer

- Second-Generation Programming Languages (2GL)

  Assembly language (1947/1949)

  Alphabetical letters, huge leap from binary

  Low-level, no compiler, assembler to translate it into computer language

- Third-Generation Programming Languages (3GL)

  1^{st} look at high-level: FORTRAN (1957), BASIC (1964), Pascal (1970), C (1972)

  Syntax resembles human languages

  Introduction of variables, constants, loops, flow constrains

  Compiler (1^{st} 1951) to convert to machine language

  Enabled the development of complex software systems

  * "Advanced 3GL": Python, Java, Ruby, C++, C#

- Fourth-Generation Programming Languages (4GL)

  SQL, MATLAB, Octave

  Human language form of thinking and conceptualize

  Many of them based on C code

  Database related, report generation, and data manipulation

- Fifth-Generation Programming Languages (5GL)

  OPS5, Mercury, ICAD, ProLog

  Constrain-based languages (instead of procedural)

  Problem-solving using constrains rather than using an algorithm written by a programmer

  Use of logic and declarative programming paradigms

  Pushed forward the development of AI and complex problem-solving systems

1.4 The DRY Principle

The *DRY principle* stands for: "Don't Repeat Yourself", also known as "duplication is evil", and is a cornerstone of good practices in programming, specially in Software Development. The principle was coined by Andy Hunt and Dave Thomas in their book *The Pragmatic Programmer* (1999) and its powerful idea states:

Every piece of knowledge must have a single, unambiguous, and authoritative representation within a system.

Figure 2: "Don't repeat yourself" (DRY) principle.

In practical terms, it means no physical copies of code all over the place! It aims to reduce repetition of information which is likely to change (and it *will*! We are humans and code need to be flexible), replacing it with abstractions[†] that are less likely to change. One of the purposes of the DRY principle is to centralize reusable code. In the context of OOP, we avoid repetition by defining *Classes* and instantiating *objects*.

Keep it *DRY*, don't be *WET*: "write everything twice", "write every time", "we enjoy typing" or "waste everyone's time"!

1.5 Programming Language: C++

Here are the notes of a first contact with Object-Oriented Programming (OOP) through the lens of the C++ programming language. As the previous background relies on C programming, and so many times codes implemented in C/C++ seems interchangeable, it sounds worthy to highlight differences between C and C++.

Starting with the common heard allegations that "C++ is a superset of C" and that C++ is "C with class", nowadays, it is no longer nearly true as it was when originally created (Figure 3). Both languages evolved and developed specific aspects that make them differ. It is not bad nor false to consider:

C++ is mostly a superset of C C adding Object-Oriented Programming, Exception Handling, Templating, and a more extensive standard library.

Aspects of C++ that differ from C:

- C++ supports polymorphism, encapsulation, and inheritance *i.e.* is an OOP language;
- C++ supports both procedural and object-oriented programming paradigms (hybrid language);

[†]In computer science, information hiding is the principle of segregation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed.

- Data and functions encapsulated together in form of an object in C++ against separated in C;
- Functions can be used inside a structure in C++;
- Namespace is used in C++ avoiding name collisions;
- Virtual and friend functions are supported by C++;
- Exception handling is supported by C++;
- Strict type checking in done in C++. So many programs that run well in C compiler will result in many warnings and errors under C++ compiler;
- Named initializers must match the data layout of the struct in C++, instead of may appearing out of order in C;
- ...

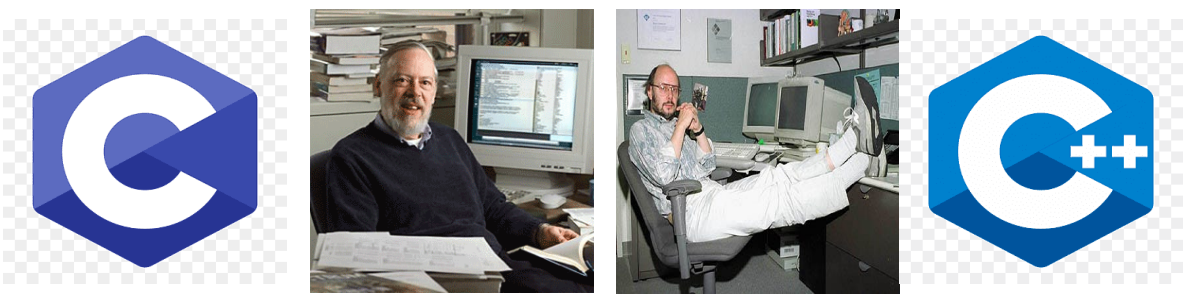


Figure 3: Dennis Richie, developer of C (1969 - 1973) and Bjarne Stroustrup, developer of C++ (1979) (*from left to right*).

Practical changes:

Extension	`.c` $\longrightarrow$ `.cpp`
Header	`<stdio.h>` $\longrightarrow$ `<iostream.h>`
Input	`scanf()` $\longrightarrow$ `cin`
Output	`printf()` $\longrightarrow$ `cout`

Regarding dynamic memory, allowed by both, the functions `malloc()` and `free()` are forgotten as C++ provides new memory allocation operator and delete operator for memory de-allocation. Also, less common file extensions in C++ are given by `.c++`, `.cc`, `.cxx`.

Personal comment: Thanks gods of C++, no more dozens of variable specifications just to get input/output (I/O)!

2 Objects

The most fundamental concept in Object-Oriented Programming (as it couldn't be different) is the concept of **Object**. Here is a wide variety of definitions[‡] for *Object*:

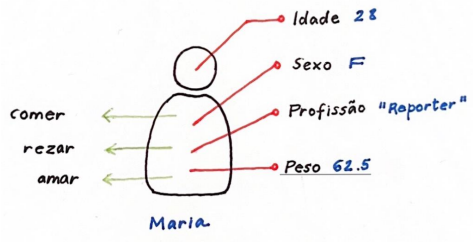
An object is a data structure or abstract data type containing fields (state variables containing data) and methods (subroutines or procedures defining the object's behavior in code).

In software development, an object is an entity that has state, behavior, and identity. An object can model some part of reality or can be an invention of the design process whose collaborations with other such objects serve as the mechanisms that provide some higher-level behavior. Put another way, an object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages).

In object-oriented programming (OOP), objects are the basic entities that actually exists in the memory. Each object is based on a blueprint of attributes and behaviours (variables and functions) defined as Class.

Example of an object



- Identity: Maria (object's name)
- State: `Idade`, `Sexo`, `Profissão`, `Peso` (age, sex, profession, weight) - Contains information about Maria
- Behavior: `comer`, `rezar`, `amar` (eat, pray, love)

3 Classes & Objects

The role of a **class** in C++ is to define a *template* or blueprint for creating **objects**. A class contemplates a family of objects that share a same group of characteristics (*attributes*) and behave/act in a certain manner (*methods*).

To create a new class, we use the command `class`:

```
class MyClass;
```

[‡]Ref. OOP - Objects Wikipedia, Object (computer science) Wikipedia, The Java Tutorials Oracle, What are Objects in Programming? Geeksforgeeks

```cpp
#include <iostream>

// Creation of a Class
class Treco {
    // Body
};
```

Regarding position in the C++ code, the classes are defined above the `main` function to keep the code legible and well-organized (good practices). By common agreement, classes are indicated with upper case.

The creation of an object in OOP is often called *instantiation*[§]. A specific object is an **instance** of the class. To create an object, we use the indication of the class of the object, followed by the object name:

```
MyClass myObject;
```

```cpp
// Creation of an object
int main(){
    Treco marreco; // Object of class Treco: instantiation

    // Fill the object
    marreco.tam = 10;
    marreco.name = "Marre";

    return 0;
}
```

Objects are created inside the `main` function, and following good practices, objects are indicated with lower case.

It is possible to create as many objects of a class as wanted. And a scenario where we need dozens, hundreds, or even more objects is where we can start to visualize and appreciate the power, economy, and sanity provided by the OOP paradigm. Instead of copying the block of code describing the object characteristics and actions n times the amount of required objects, just one declaration of the Class is needed, followed by the instantiation of the specifics objects in question. It's the DRY principle in action!

3.1 Attributes

Attributes are characteristics of a Class, or the characteristics that a Class can have within the program. Attributes are basically *variables* that belong to the class. An attribute is a "class member". A Class can have multiple attributes, each one presented as a variable declaration, with a certain type of variable and a name associated to it:

[§]**Instantiation**, most simply understood to mean "exemplification," or "the ideal example or representation," is also a more complicated philosophical idea. The "instantiation principle" is essentially the concept that a characteristic or property can't exist unless you can point to a real-world instance of it. Ref.https://www.vocabulary.com/dictionary/instantiation

```
type attribute;
```

Classes in C++ allow for attributes of different `type`. There is no problem in mixing types and using them multiple times, a Class can have inside it `char`, `int`, `double`, `bool`, `string` and even `struct` attributes.

The attributes access of an object in C++ is done using the dot (`.`) operator with the object name:

```
myObject.name;
```

```cpp
#include <iostream>
using namespace std;

// Class definition
class Animal {
    public: // Access specifier
        char zone;        // Attribute (type char)
        int size;         // Attribute (type int)
        double weight;    // Attribute (type double)
        bool carnivorous; // Attribute (type bool)
        string sp;        // Attribute (type string)
};

int main() {
    Animal tiger; // Class instantiation

    tiger.zone = 'a';
    tiger.size = 10;
    tiger.weight = 120.5;
    tiger.carnivorous = true;
    tiger.sp = "Bengala Tiger";

    cout << "Zone: " << tiger.zone << '\n';
    cout << "Size: " << tiger.size << '\n';
    cout << "Weight: " << tiger.weight << '\n';
    cout << "Carnivorous: " << tiger.carnivorous << '\n';
    cout << "Species: " << tiger.sp << '\n';
    return 0;
}
```

3.2 Methods

Methods are basically *functions* that belong to the class in C++. They are related to the actions and behavior of the objects of a particular class. A method is a "class member". A method is defined in C++ in one of the following ways: inside or outside class definition. The subsequent access of the method is done by creating an object of the class and using the dot (`.`) operator (just like accessing attributes).

Example Method defined inside and outside class definition

```
    class Car {
    public: // Access specifier
        // Attributes
        int cc;      // Car engine (cc)
        int tank;    // Capacity (L)
        string name; // Model name
        double price_tank; // Amount to fuel the tank ($)

        // Methods
        // Method defined inside class
        void fuelCar(double price_fuel) {
            price_tank = tank * price_fuel;
        }
        // Method defined outside class
        void moveCar(); // Method declaration
};

// Method definition outside the class
void Car::moveCar() {
    cout << "vrum vrum" << "\n";
    cout << name << " on the highway " << cc << " cc" << "\n";
}
```

3.3 Constructor

Special method called (automatically) when an object of a class is created.

- Always same name as the class;
- Always public;
- Don't have return value!

A constructor is defined in C++ using the class name followed by parentheses:

```
class MyClass {
    public:
      MyClass() {
       cout << "Hello World!";
       }
};
int main() {
    MyClass myObj;
    return 0;
}
```

```
    class Toy { // The class
        public: // Access specifier
            Toy() { // Constructor
                cout << "Toy Story" << endl;
            }
    };
```

It is possible to add parameters to the Constructor (just like regular functions):

```
    class Car { // The class
        public: // Access specifier
            string brand; // Attribute
            string model; // Attribute
            int year;     // Attribute

            // Constructor with parameters
            Car(string x, string y, int z) {
                brand = x;
                model = y;
                year = z;
            }
    };
```

Adding parameters to the Constructor makes it specially useful in terms of setting the initial values for the object's attributes, which can be performed right along with the object instantiation:

```
    int main() {
    // Create Car objects + call constructor w/ different values
    Car car1("Fiat Uno", "Mille", 1983);
    Car car2("Ford", "Mustang", 1969);
    //Car uno; // Error: default constructor is missing

    // Print the objects and its attributes
    cout << car1.brand << " " << car1.model << " " << car1.year << endl;
    cout << car2.brand << " " << car2.model << " " << car2.year << endl;
    return 0;
}
```

Example Definition of a constructor outside of a class

→ Use of the :: operator with the class name

```
class Food { // The class
    public: // Access specifier
        string name;   // Attribute
        bool tasty;    // Attribute
        int stars;     // Attribute
        string status; // Attribute
        // Constructor declaration
        Food(string in_name, bool in_tasty, int in_stars);

```

```
        // Methods
        void avaliation() {
            cout << "Food " << name << " avaliation:" << endl;
            cin >> stars;
        }
};

// Constructor definition outside the class
Food::Food(string in_name, bool in_tasty, int in_stars) {
    name = in_name;
    tasty = in_tasty;
    stars = in_stars;
    status = "Not rated yet";
}
```

3.4 Specifier

The specifier element of a Class indicates the *access* each component - attribute or method - has. It's the idea of having credentials in real life to access something. Allows hiding members from the outside world. In C++, there are three *access specifiers*:

- `Public`: accessible from outside the class;
- `Private`: cannot be accessed (or viewed) from outside the class;
- `Protected`: cannot be accessed from outside the class, but can be accessed in inherited classes.

Note that: Private members are not lost. It is possible to access private members from inside the class, *ie.* using a public method inside the same class.

OBS: It is considered good practice to declare class attributes as private (as often as possible). This will reduce the possibility of messing up the code. It is related to the property of C++ of providing data hiding, thus more secure code.

4 Graphic Representation - Class Diagram UML

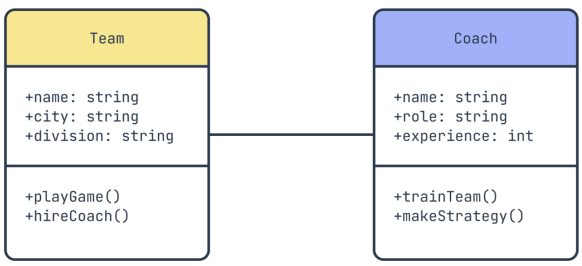


Figure 4: UML Class Diagram representation

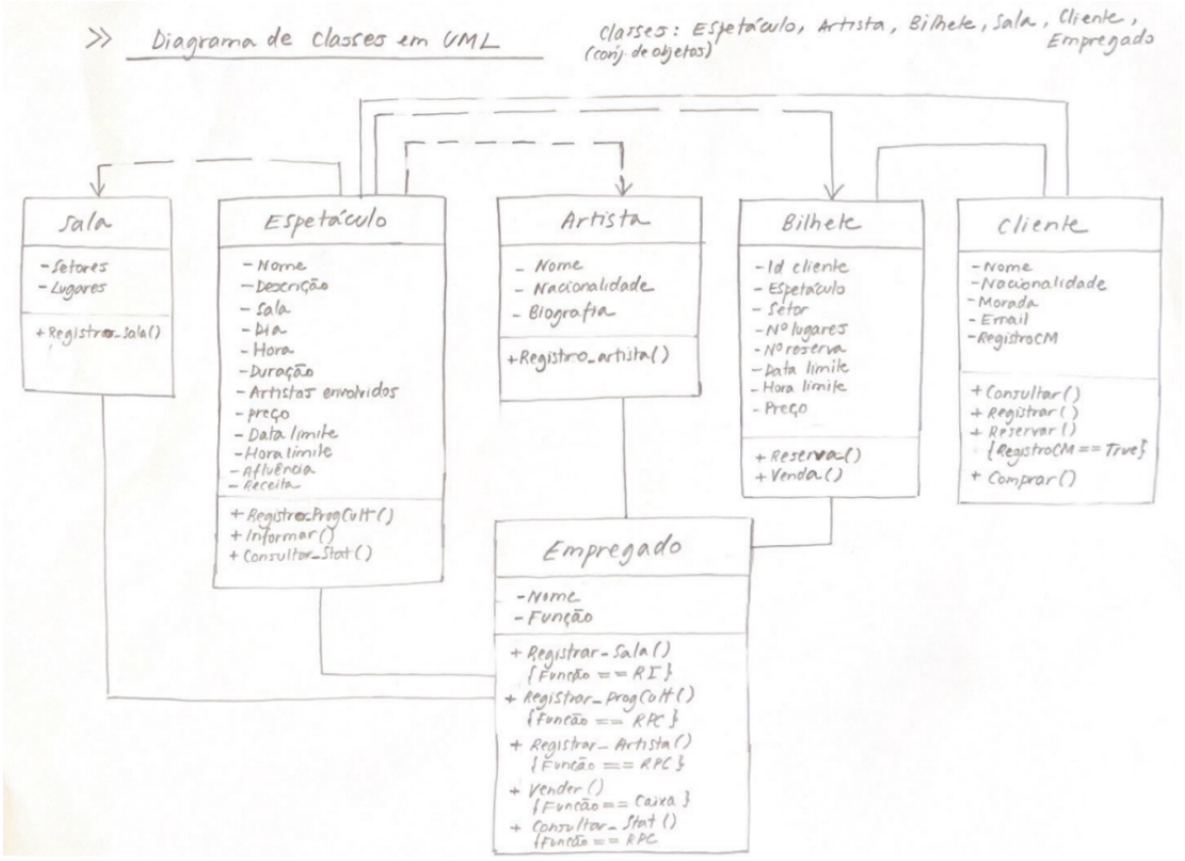


Figure 5: Example of a Class Diagram in UML - Casa da Música

5 Encapsulation

6 Inheritance

6.1 Class-based programming

6.2 Prototype-based programming

Referências

1. "C++ Classes", *W3Schools*, https://www.w3schools.com/cpp/cpp_oop.asp
2. "Object-oriented programming", *Wikipedia*, https://en.wikipedia.org/wiki/Object-oriented_programming#cite_note-2
3. "Coding milestones: a journey through the history of programming", *Nomical Connecting Worlds*, https://nomical.com/news/coding-milestones
4. "The Generations of Programming Languages — Computer Science History", ForrestKnight, *YouTube*, https://www.youtube.com/watch?v=ZFIeEV2HfPQ&ab_channel=ForrestKnight
5. "Programming languages generations", Steve Yonkeu, https://dev.to/yokwejuste/programming-languages-generations-2o6p
6. "OO in One Sentence: Keep It DRY, Shy, and Tell the Other Guy", Andy Hunt and Dave Thomas, *IEEE Software* (2004) https://media.pragprog.com/articles/may_04_oo1.pdf

Index