Object Oriented Programming

Samuel Navarro

May 14, 2019

Contents

1 Intro to OOP

In the Guideline, when you don't have logic in your classes, it's recommended to use public variables but when you have logic in your classes, it's recommended to stablish your variables as private and use setter or getter functions (public ones) to access them.

We have three access modifiers:

1. public: Everybody can access it.
2. private: Only members of the class can access it.
3. protected.

2 Encapsulation and Abstraction

2.1 Encapsulation

- **Encapsulation** means that we bundle related properties together in a single class and sometimes we protect those properties from being modified accidentally or in an unauthorized manner.
- **Abstraction** means that users of our class only need to be familiar with the we provide.

In one of the examples:

```
1 class someClass{
2         private:
3                 std::string name;
4         public:
5                 std::string getName() const;
6 };
```

Listing 1: const

We can mark as `const` only the getter functions.

One thing to note is that the C++ core guidelines specify that we need to avoid trivial getters and setters so instead we just make the variables public.

2.1.1 C-style strings

A **C-style string** is simply an array of characters that uses a null terminator. A **null terminator** is a special character `'\0'` used to indicate the end of the string. More generically, A c-style string is called a **null-terminated string.**

Note: To see more about the correct way of declaration of pointers see this by Stroustrup. (TL;DR the answer is about style and emphasis).

In the code below, we set the brand and convert this from string to c-style array.

The notion of encapsulation as a way of passing data and logic bundled together in a single object is at the very core of OOP.

One thing to notice is that the options that were incorrect were:

- A requirement that data and logic be packaged separately in distinct objects.
- The restriction that logic within a particular object can only operate on data stored within that same object.

```
class Car{
        private:
                char* brand;
        public:
                void setBrand(std::string brandName);
                std::string GetBrand();
};

void Car::SetBrand(std::string brandName){
        Car::brand = new char[brandName.length() + 1];

        strcpy(Car::brand, brandName.c_str());
}

std::string Car::GetBrand(){
        std::string result = "Brand name: ";
        result += Car::brand;
        return result;
}
```

Listing 2: C-Style string

But this ones were similar to what Bjarne said in the video. **Update: Bjarne just states the data has been Encapsulated inside the object that provides the interface.**

Making class attributes private and assigning them with a setter function allows you to Invoke logic that checks whether the input data are valid before setting attributes. Setter functions can be written to run any series of checks on the inputs before assigning attribute values, or return an error to the user.

Again, I need to check why this two are incorrect:

- Ensure that only class member functions have access to private class attributes.
- Prevent users from changing non-public class attributes.

2.2 Abstraction

Bjarne: *"Is just getting away from the hardware and building from other things. Is like if you stack a lot of pre-built things and built another one."*

You don't care about all the variables and the way the function handle the variables, you just know that you have a `SetDate` and a `GetDate` function. (Basically abstraction is just separating the logic from the call).

The user is able to interact with the `Date` class through the `GetDate()` function. But the user does not need to know how `Date` is implemented. For example, the user does not know, or need to know, that this object internally contains three `int` variables. The user can just call the `GetDate()` method to get data. If the designer of this class ever decides to change how the data is stored internally – using a vector of `ints` instead of three separate `ints`, for example – the user of the `Date` class will not need to know.

All that abstraction means is that you can modify all the logic and the implementation in your code and the users don't experience an impact in there usage by your changes in the code.

Abstraction is used to show only relevant information to the user and hide any irrelevant details. In this example, you'll get more practice with how to implement abstraction in your code.

The `privateMethod.cpp` file is a example to show that we can also hide irrelevant **methods** from the users by making them private.

2.2.1 Static Attribute

With abstraction we can use the concept of **static members** in C++ classes. Class members can be declared using the class specifier `static` in the class attributes list. Only one copy of the static attribute is shared by all instances of **a** class in a program. When you declare an object of a class having a static member, the static member is not part of the class object.

You can create as many instances of a class as you want but there's only one copy of the `static` attribute. The only thing to consider is the fact that if you have another class with the same `static` attribute, this attribute is only shared with the instances of this class. That is, static attributes exist beyond a particular instance of a class, but do not extend into conflict with other static attributes defined within other classes.

2.2.2 Static Method

In `main` function we haven't instantiated the class Abstraction.

Why? Because the method is static. So the scope is far greater than if it was just declared as `void PrintCharAsNumber(char c)`.

In the context of OOP, the concept of abstraction refers to: the notion of hiding unnecessary detail from the user.

Abstraction of unnecessary detail allows you to provide powerful functionality with much simpler looking code.

Static members are not bound to a class instance in the sense that only one instance of a static member can exist across multiple instances of the class in which it is defined.

A `static` member provide a way of tracking multiple instances of a class with a single member.

```cpp
#include <iostream>

class  Abstraction {
private:
    int number;
    char character;
public:
    void static PrintCharAsNumber(char c);
};


void Abstraction::PrintCharAsNumber(char c) {
    int result = c;
    std::cout << result << "\n";
}

int main() {
    char c = 'X';
    Abstraction::PrintCharAsNumber(c);
    // OUTPUT: 88
}
```

Listing 3: caption name

Static methods are not bound to a class instance in the sense that they can be invoked without actually instantiating the class.

Bjarne on Thinking about Classes Conceptually: basically you look at your application domain. A programmer just can't be a programmer expert. What the problem is that you have to solve?. So you have to understand the world of your users, of your customers, etc. You listen to how they talk and what are the concepts they deal with.

Allow your customers to work at the level they would like to work at using their vocabulary.

What we do is we lift the language from the machine up to the humans. We set a language that fits for humans.

3 Inheritance and Polymorphism

Inheritance is one way that classes can relate to each other.

Polymorphism is a related concept that allows an interface to work with several different types.

Example: We may have a `cut` function that can work with the plant,

fruit or apple class. In fact, the `cut` function may also accept another totally different types of objects, like paper.

3.1 Inheritance

Bjarne on inheritance: You build on top of something else. You start more general and then more and more specialized.

Inherited Access Modifiers

But it's important to note that declaring if the derived class will be `public` or `private` or `protected` is not obligatory. You can just inherit the class without stating the privacy level and you will get the same attributes as the base class.

3.2 Polymorphism

Polymorphism is just a fancy word that means "taking many forms".

In C++ it can be achieved in two ways:

- **Compile-time polymorphism:** This is when you overload a function. You write two (or more) functions that have the same name. This will of course generate compiler errors if you write the same function declaration and implementation. However, it works just fine if you overload different parts of the functin, such as defining each function with a different configuration of input arguments. It decides what function to use based on the parameters you use. Check the `polymorphism_overloading.cpp` file. The big difference is just in the declaration of `PrintDate`.

- **Function Overriding**: It occurs when a multiple definitions of a function have the same signature (same name and same arguments). Each time that function is called, one definition of the function will override the others, although which version gets to override may change based on context. When a method is defined in both a base class and a derived class, the definition in the derived class can override the definition in the base class.

There's one important concept called **Runtime Polymorphism** and it refers to the fact that virtual functions are binded at runtime. The **Function Overriding** could be like this:

Runtime Polymorphism is accomplished by performing function overriding. This is performed using class hierarchies. Any derived class can override function members that belong to the base class.

We need to notice that because the `PrintVirtual` function is marked as `virtual`, when you call the function with `pointer->PrintVirtual();` the derived class gets called.

```
// This example demonstrates the privacy levels
// between parent and child classes

class ParentClass{

public:
     int var1;
protected:
     int var2;
private:
     int var3;
};

class ChildClass_1 : public ParentClass{

   // var1 is public for this class
   // var2 is protected for this class
   // var3 cannot be accessed from ChildClass_1
};

class ChildClass_2 : protected ParentClass{

   // var1 is protected for this class
   // var2 is protected for this class
   // var3 cannot be accessed from ChildClass_2
};

class ChildClass_3 : private ParentClass{

   // var1 is private for this class
   // var2 is private for this class
   // var3 cannot be accessed from ChildClass_3
};
```

Listing 4: Access Modifiers

That is, **Virtual Methods** are declared (and possibly defined) in a base class, and are meant to be overridden by derived classes.

In the class Shape 6 If we delegate with instruction `= 0` we are notifying compiler that this (base) class doesn't have virtual method implementation but every other derived class is required to implement this method. Function overriding occurs when a derived class function calls its own definition of a

```cpp
class Base_Class {
    public:
        virtual void PrintVirtual ()
        { std::cout << "This is a message from the base class!!" << "\n"; }

        void Print ()
        { std::cout << "This displays the base class." << "\n"; }
};

class Derived_Class : public Base_Class {
  public:
      //print () is already virtual function in derived class, we could also declar
      //void print () explicitly
      void PrintVirtual ()
      { std::cout << "This is a message from the derived class!!" << "\n"; }

      void Print ()
      { std::cout << "This displays the derived class." << "\n"; }
};


int main()
{
    Base_Class *pointer;
    Derived_Class der;
    pointer = &der;
    //virtual function, binded at runtime (Runtime polymorphism)
    pointer->PrintVirtual();  // OUTPUT: print derived class

    // Non-virtual function, binded at compile time
    pointer->Print(); // OUTPUT: print base class
}
```

Listing 5: Polymorphism Overriding

method, instead of the base class's implementation.

3.3 Generic Programming

Generic code is the term used for code that is independent to types. It is mandatory to put the `template<>` tag just above your function to specify and mark which implementation is generic.

Bjarne: you don't want to organize your classes and have hierarchy

```
class Shape {
        public:
                Shape() {}
                virtual double Area() const = 0;
                virtual double PerimeterLength() const = 0;
};
```

Listing 6: Virtual Functions

across them. What you want to say is: *this algorithm works* for anything that are can be used for you algorithm.

Hierarchy and inheritance disappears but the ability to operate on anything that has the right interface stays.

3.3.1 Templates

With templates, the idea is to pass a data type as a parameter so that you don't need to write the same function code for operating on different data types.

Rather than writing and maintaining the multiple function declarations, each accepting slightly different arguments, you can write one function and *pass the argument types as parameters*. At compile time, the compiler then expands the code using the types that are passed as parameters.

Example:

```
template <typename Type>
        Type Sum(Type a, Type b){
                return a+b;
        }

int main(){
        std::cout << Sum<double>(20.0, 13.7) << std::endl;
}
```

Listing 7: Template

With the code for Templates 10, the compiler add the following internally:

Bjarne on Templates: is something you can parametrize with types or values.

Templates: manipulate things at compile time. But they have lousy interfaces. You cannot say what are the requirements of the elements of a

```
double Sum(Type a, Type b){
        return a+b;
}
```

Listing 8: caption name

vector.

C++20 will have something called `concepts` that allows you to precisely specify template requirements on its elements.

3.4 Constructor Syntax

Initialization lists are used for a number of reasons.

1. The compiler can optimize initialization faster than if the members were initialized from within the constructor. The code directly assigns to the class attributes.

2. The second reason is a bit of a technical paradox. If you have a **constant** class attribute, you cann only initialize it using an initialization list. Otherwise, you would violate the `const` keyword simply by initializing the member in the constructor!

3. Attributes defined as references must use initialization lists.

Initialization lists are particularly helpful for user-defined class members. IF you don't have access to the default constructor fo your attributes, you have a perfect use case for initialization lists. (Check `initialization_list.cpp`)

4 Project

5 Questions

I don't understand why you need to apply the scope resolution operator on attributes in the definition of methods. Example:

I believe the answer is in the class_hierarchy.cpp code. The answer is that you don't want to mess up with the attributes of the other classes or you can't even change them.

In the `friend_class.cpp` I don't understand the need to first declare the `Rectangle` class, then implement the `Square` class and then implemente `Rectangle`. I've tried to implement `Rectangle` after `Square` and it works fine.

Another one:

```
void Abstraction::ProcessAttributes() {
    number *= 6;
    character++;
}

// vs


void Abstraction::ProcessAttributes(){
        Abstraction::number *= 6;
        Abstraction::character++;
}
```

Listing 9: Question Scope Resolution

```
template <typename T1, typename T2>
void multiply(T1 num1, T2 &container){
        for( auto& element : container){
                element *= num1;
        }
}
```

Listing 10: Template

I don't understand why we need to pass by reference the container and it's element in order to print the value after multiplication.

Check code in `constructor_syntax.cpp` when we apply `+=` to the referenced variable.