Object Oriented Programming II

Raul P. Pelaez

October 31, 2024

Contents

1 Introduction

In this lesson, we'll dive deeper into the special methods that C++ provides for managing the lifecycle of objects: constructors and destructors. We'll also explore other special methods and operators that allow us to customize the behavior of our classes.

Info

Lifecycle of an object

- Construction
- Usage
- Destruction

Constructors and destructors are special methods that handle the creation and cleanup of objects, respectively.

2 Constructors

Constructors are special methods called when an object is created. They initialize the object's state and allocate any necessary resources.
The main purposes of constructors are:

- To initialize the object's data members
- To allocate any resources the object needs
- To ensure the object starts in a valid state

Without a constructor, all data members of an object would be uninitialized, leading to potential undefined behavior, bugs and performance penalties.
When an object is created, the constructor is called automatically. If no default constructor is defined, the compiler provides a default constructor that initializes the object's data members to default values. This means that in a construct like this

Source code

```
class MyClass {
int data;
};
int main() {
  // Calls the default constructor
  MyClass obj; // data is allocated and default-initialized here
  obj.data = 42; // data is now set to 42
}
```

members are initialized twice.

Advice

Default-initialization might be ok for something like an `int` but there are situations where one really does not want to initialize the members twice. For instance, imagine a member in a class in a graphical application that represents a window. We would not like to open some default window, then close it and then open it again with the desired properties.
Another problem is if one of the members does not have a default constructor itself, or when a member is defined as const.

Info

Initialization is a really complex topic in C++, with many subtleties, hidden defaults and syntaxes. For instance, all of these are valid ways to initialize a variable in C++:

Source code

```
struct A{
  // Setting default values here is a way to initialize the members
  int a = 0;
  int b = 1;
  float c = 2.0;
};
int main(){
  A brace_initializer{1,2,3.0}; // from C++11
  A designated_initializer{.a=1,.b=2,.c=3.0}; // from C++20
  A default_initializer{};
  A parenthesis(1,2,3.0);
  A copy_initializer = brace_initializer;
  A copy_op(brace_initializer);
  A moved = std::move(brace_initializer);
}
```

2.1 Basic types of Constructors

C++ provides several types of constructors to handle different scenarios:

2.1.1 Parameterized Constructor

A parameterized constructor takes one or more arguments and is used to initialize an object with specific values.

Source code

```
class Rectangle {
private:
  double width;
  double height;
public:
  // Constructor with two parameters
  Rectangle(double w, double h) : width(w), height(h) {}
  // Constructor with one parameter (creates a square)
  Rectangle(double side) : width(side), height(side) {}
};
void foo(){
  Rectangle r1(2.0, 3.0); // Calls the first constructor
  Rectangle r2(4.0); // Calls the second constructor
}
```

Similarly to regular functions in C++, constructors can be overloaded, meaning that you can have multiple constructors with different signatures.

2.1.2 Default Constructor

A default constructor is one that can be called with no arguments. It's either defined by the programmer or provided by the compiler if no other constructors are defined.

Source code

```
class MyClass {
  std::string a_member;
public:
  MyClass() {
    // Initialize members here
    a_member = "default";
  }
};
MyClass obj; // Calls the default constructor
```

Advanced

The compiler will write a default constructor for you only when no other constructor is defined. You can overcome this by explicitly defining a default constructor or using `=default` to request the compiler to generate it.

Source code

```
class MyClass {
  int a;
public:
  MyClass(int a): a(a) {}
  //The default constructor must be defined explicitly if another is
  ↪  defined
  MyClass() = default; // Request the compiler to generate the default
  ↪  constructor
};
```

Advanced

`explicit` constructors
If a constructor can be called with a single argument, it can be used as an implicit conversion operator. This can lead to unexpected behavior and bugs. To prevent this, use the `explicit` keyword to make the constructor explicit.

Source code

```
class MyClass {
  int data;
public:
  explicit MyClass(int d) : data(d) {}
};
// This line would be ok if the constructor is not explicit
MyClass a = 42; // Error: no implicit conversion allowed
MyClass b(42); // Ok, explicitly-invoked constructor
```

2.1.3 Copy Constructor

A copy constructor creates a new object as a copy of an existing object. It's called when an object is passed by value, returned by value, or explicitly copied.

Source code

```
#include <iostream>
struct MyClass {
  int data;
  // Copy constructor
  MyClass(const MyClass& other) : data(other.data) {}
  MyClass(int d): data(d){
  }
};
void foo(){
  MyClass obj1(10);
  MyClass obj2(obj1); // Calls the copy constructor
  //MyClass obj2 = obj1; // Also calls the copy constructor
  std::cout << obj1.data << std::endl; // 10
  std::cout << obj2.data << std::endl; // 10
}
```

Advanced

=delete constructors
Sometimes you want to prevent certain operations from being performed on your class. You can do this by using the =delete specifier.

Source code

```
class MyClass {
  int data;
public:
  // Prevent copying
  MyClass(const MyClass& other) = delete;
};
MyClass obj1;
MyClass obj2(obj1);
// Error: copy constructor is deleted
```

2.1.4 Move Constructor

A move constructor creates a new object by "stealing" the resources of an existing object. It's called when an object is returned by value from a function or explicitly moved.

Source code

```
struct MyClass {
  int data;
  // Move constructor
  // The funny && is called an rvalue reference
  MyClass(MyClass&& other) : data(other.data) {
    other.data = 0; // Steal the resources
  }
  MyClass(int d): data(d){}
};
MyClass foo() {
  MyClass obj(10);
  return obj;
}
int main(){
  MyClass obj2(foo()); // Calls the move constructor
  return 0;
}
```

Advanced

std::move
The std::move function is used to cast an lvalue to an rvalue reference. This is useful when you want to move an object instead of copying it.

Source code

```
MyClass obj1;
MyClass obj2(std::move(obj1)); // Calls the move constructor
```

2.2 Key new concepts

Info

Constructors
Constructors are special methods called when an object is created.
They initialize the object's state and allocate resources.
There are many types of constructors: default, parameterized, copy and move.

3 Destructors

A destructor is a special method called when an object is destroyed (like when it goes out of scope). The main purposes of destructors are:

- To release resources acquired by the object during its lifetime.
- To perform any necessary cleanup operations.
- To ensure proper handling of object destruction.

Destructors are declared with a tilde (~) before the class name. They have no return type and take no arguments.
Example

Source code

```cpp
class FileHandler {
  FILE* file;// Note: in C++ we normally use std::fstream instead.
public:
  FileHandler(const char* filename) {
    // Acquaire the resource
    file = fopen(filename, "r");
  }
  ~FileHandler() {
    // Release the resource
    if (file) {
      fclose(file);
    }
  }
};
void read_file(const char* filename) {
  FileHandler handler(filename); // The constructor is called here
} // handler goes out of scope, destructor is called
```

Advice

RAII: Resource Acquisition Is Initialization
Perhaps the most powerful idiom in C++ is RAII. The idea is that resources should be acquired in constructors and released in destructors. This way, the lifetime of the resource is tied to the lifetime of the object, ensuring proper cleanup and avoiding resource leaks.
This is the reason why we can just create an `std::vector` without worrying about memory leaks, its destructor makes sure to release the memory.

3.1 Key new concepts

Info

Destructors
Destructors are called when an object is destroyed, like when it goes out of scope.
They clean up resources and perform finalization.
Destructors are declared with a tilde (˜) before the class name.

Info

RAII
Whenever a class manages a resource (a file, a pointer), it should make sure to release it in the destructor.

4 Operators

C++ allows operator overloading, which lets you define how operators work with objects of your class. We can overload any (and only) the following operators:

- Assignment operator (=)
- Stream insertion and extraction operators (<< and >>)
- Comparison operators (==, !=, <, >, <=, >=)
- Arithmetic operators (+, -, *, /, %)
- The +=, -=, *=, /=, %= operators are.
- The access operator `[]` and the parenthesis operator `()`.

When overloading operators, you can define them as member functions or as friend functions. Member functions are called on the left-hand operand, while friend functions are not.
Example

Source code

```cpp
#include <iostream>
class Complex {
  double real, imag;
public:
  Complex(double real, double imag) : real(real), imag(imag) {}
  Complex operator+(const Complex& other) const {
    return Complex(real + other.real, imag + other.imag);
  }
  bool operator==(const Complex& other) const {
    return real == other.real && imag == other.imag;
  }
  friend std::ostream& operator<<(std::ostream& os, const Complex& c) {
    return os << c.real << " + " << c.imag << "i";
  }
};
Complex a(1, 2), b(3, 4);
Complex c = a + b;
std::cout << c << std::endl; // Outputs: 4 + 6i
```

Advanced

Operators are actually just regular functions with a special syntax. For instance, the + operator is just a function called `operator+`:

Source code

```cpp
Complex a{1,2}, b{3,4};
Complex c = a + b;
Complex d = a.operator+(b);
```

4.1 Other essential operations

4.1.1 Copy Assignment Operator

A copy assignment operator assigns the value of one object to another.

Source code

```cpp
struct MyClass {
  int data;
  // Copy assignment operator
  MyClass& operator=(const MyClass& other) {
    // Copies information
    data = other.data;
    // Returns itself
    return *this;
  }
};
int main(){
  MyClass obj1(10);
  MyClass obj2;
  obj2 = obj1; // Calls the copy assignment operator
  std::cout << obj2.data << std::endl; // 10
  std::cout << obj1.data << std::endl; // 10
  return 0;
}
```

4.1.2 Move Assignment Operator

A move assignment operator assigns the value of one object to another by "stealing" the resources.

Source code

```cpp
struct MyClass {
  int data;
  MyClass(int a = 0): data(a) {}
  // Move assignment constructor
  MyClass (MyClass&& other): data(other.data) {
    other.data = 0;
  }
  // Move assignment operator
  MyClass& operator=(MyClass&& other) noexcept {
    if (this != &other) {
      data = other.data;
      other.data = 0;
    }
    return *this;
  }
};
MyClass foo() {
    MyClass obj(10);
    return obj; // Move constructor will be called here
}
int main() {
  MyClass obj1;
  obj1 = foo();  // This will call the move assignment operator
  return 0;
}
```

The rules of the language force us to define the move constructor too here. However, if you run this code while printing inside each method, you will notice the move constructor is not called. This is because the compiler is allowed to optimize the move constructor away using something called the Return Value Optimization (RVO).

4.2 Key new concepts

Info

Operators
Operators can be overloaded to define how they work with objects of your class.
You can overload the assignment, stream insertion/extraction, comparison, and arithmetic operators.
There are special operators for copy and move assignment, when the object is assigned to another via the = operator.

Info

The essential operations
When dealing with initialization/destruction, we can think of these essential operations:

- Destructor
- Copy constructor
- Move constructor
- Copy assignment operator
- Move assignment operator

5 The Rules of "0", and "all"

If a class needs any of the essential operations, it probably needs all. This adds up to two rules of thumb:

- Rule of zero: If you do not need to, don't define any of the essential operations.
- Rule of all: If you define any of the essential operations, define them all.

The Rules are guidelines for managing resources in C++ classes.
There is a great discussion about this concept in cppreference.

For instance, this class does not need any of the essential operations, the compiler will generate them for us.

Source code

```
struct Club {
  std::string name;
  std::vector<Member> members;
};
```

Advice

These rules are usually called the "Rule of 0, 3, 5" but the creator of C++, Bjarne Stroustrup, argues that only the rule of 0 and "all" is a better concept. In his opinion, the distinction between 3 and 5 stems from people not agreeing on how to count the operations.

6 Exercises

Each milestone is intended to reflect at least one of the "Key new concepts" laid out in the notes for today. Please read the notes if the milestone's description or intent is not immediately clear to you. Consider the advanced milestones as optional as far as submission is concerned, but please always read them anyway even if you decide you do not have time to go through them.

6.1 The rule of zero

Goal

Lets create a class called `Person` that does not define any of the special member functions (constructor, destructor, copy constructor, copy assignment operator, move constructor, move assignment operator), relying on the compiler to generate them for us.

With the implementation of this class, this code should compile and run:

Source code

```
int main(){
  Person p1{"Alice", 30, {"reading", "hiking"}};
  Person p2 = p1;  // Compiler-generated copy constructor
  p2.name = "Bob";
  p2.hobbies.push_back("swimming");
}
```

Learning goal

Realize that most of the time the compiler-generated special member functions are enough for our classes.

6.2 The DynamicArray class

Goal

Lets write a class that tries to mimic some of the functionality of `std::vector`.
With the complete implementation, this code should succeed:

Source code

```cpp
#include "DynamicArray.h"
#include <iostream>
#include <random>
using namespace oop; // Lets assume the header is in a namespace called
↪  oop
DynamicArray createRandomArray(){
  static std::default_random_engine generator;
  static std::uniform_int_distribution<int> distribution(1,100);
  DynamicArray a(10);
  for(int i = 0; i < 10; i++){
    a[i] = distribution(generator);
  }
  return a;
}
int main(){
  DynamicArray a(10, 1); // 10 elements, all initialized to 1
  a[0] = 2; a[1] = 3;
  DynamicArray b = a; // Copy assignment operator
  DynamicArray c = createRandomArray(); // Move assignment operator
  DynamicArray d(b); // Copy constructor
  DynamicArray e(createRandomArray()); // Move constructor
  std::cout << a << std::endl; // Stream insertion operator
  DynamicArray f = a + b; // Addition operator, concatenates the arrays
  if(f.size() != a.size() + b.size()){
    std::cerr << "Error: size of f is not the sum of sizes of a and b" <<
    ↪  std::endl;
  }
  return 0; // Destructor is called on all objects: no memory leaks
}
```

The class should have just two private members: a pointer to the memory allocated to store the array, and an integer to store the size of the array.
The rule of zero cannot be applied to this class, as we need to store a pointer to the array inside the class. When copying this class, we do not want just the pointer to be copied, but the memory it points to. This means we need to define a copy constructor that will allocate a new memory block and copy the contents of the original array to the new one.
Because of the rule of all, this forces us to define all the other essential member functions: destructor, copy assignment operator, move constructor, and move assignment operator.

We have not yet learned about generic programming (templates), so we cannot make our class hold any type (like we do with vector when we write `std::vector<int>` or `std::vector<std::string>`). We will make our class hold integers.

Learning goal

Familiarize with the rule of all, i.e. a case where all the essential operations must be defined manually.

Milestone

Implement the class DynamicArray using `std::vector` as a member to hold the data.
Add the following member functions:

- A constructor that takes an integer and initializes the array with that size, all elements set to 0.
- A constructor that takes two integers, initializing the array with the first integer, all elements set to the second

hint

By relying on `std::vector` to do the heavy lifting, you can make use of the rule of zero and simply skip all special operations. All copies and move will work out of the box.

Learning goal

- Realize that even with classes with sophisticated functionality, the rule of zero can be applied by making use of the standard library.
- Have a quick, safe implementation of the class to test the rest of the functionality.

Milestone

Make the `[] operator` work on DynamicArray.

hint

Overload the [] operator to return a reference to the element at the given index, simply forwarding the call to the [] operator of the vector member.

Learning goal

Familiarize with operator overloading. Realize there is no difference, syntactically, between operators and any other function (besides funny naming).

Advanced milestone

Make the + `operator` work on DynamicArray by concatenating both arrays.

hint

Use the `insert` function of the vector member to concatenate the two arrays.

Advanced milestone

Make the << `operator` work on DynamicArray by streaming the contents of the array.

hint

Operator overloading must be implemented as part of the class of the left-hand side of the operation (in this case, `std::cout`) or as a free function that takes as arguments the left-hand side and the right-hand side of the operation (in this case, `std::ostream` and `DynamicArray`).
We do not have access to the implementation of `std::ostream`, so we must implement the operator as a free function. In order to do that the << `operator` must be a friend function of the class, so it can access its private members.
Even if it does not need to access any private members, people usually declare it as a friend function to be able to define it inside the class definition.

Milestone

At this point, the code in the goal description should compile and run successfully. If you decided not to work on the advanced milestones, you can just comment the lines involved in them.
Change the class so that you manually manage the memory for the array. Remove the `std::vector` member and replace it with a pointer to an integer and an integer storing the length.

hint

- You need to allocate the memory in the constructor (use the `new` operator) and delete it in the destructor (use the `delete` operator).
- This changes forces you to define all the special member functions, as the rule of zero cannot be applied anymore. You will need to define a destructor, a copy constructor, a copy assignment operator, a move constructor, and a move assignment operator.
- Keep the previous implementation of the class with the name "DynamicArrayVector" and create a new class called "DynamicArray" with the new implementation. This way you can use that to test that the new implementation works as expected.

Learning goal

- Realize that the rule of zero is not always applicable.
- Realize that sometimes the compiler-generated essential operations do not do what we need.
- Understand the importance of the rule of all in avoiding memory leaks.

Milestone

Write tests with GTest for the new implementation of DynamicArray. Either by comparing with the reference one that uses a vector or by writing new tests.

hint

Use your (or my) homework solutions from last week as a starting point.

Learning goal

Advance your familiarity with CMake and GTest.