# Objects II

**T2** Chapter **11**

## Object-Oriented Programming

Let's remind ourselves from the previous sections that the struct and class keywords are used to define custom types so that we may use them the same way that we use an int or a float. We also came across the suggestion that in C++, object-oriented programming is all about the creation of the right objects for the problem we are trying to solve. For example, if we are trying to keep track of a bank account, you will probably need to define types like Account and Transaction, while for a program that analyzes baseball scores, it may have types such as Player and Team.

Unlike the procedural paradigm, where functions are the fundamental units of behavior, in the object-oriented paradigm, objects are the fundamental units of behavior. Not only do objects store data but this data is also closely associated with behavior. The way to understand behavior in objects is to think of them as functions, but more correctly, these functions are called **methods** or **member functions**, just as a class or struct has **member variables**.

Note that in the procedural paradigm, data is separate from functions - data is given to functions to operate on. In the object paradigm, on the contrary, there is no clear distinction between data and functions or methods. It is more correct to say that objects contain methods and that these methods operate on the objects as a whole, and thus changing the state of the object.

As a consequence of this close association of behavior with data, objects interact with other objects only through their methods or behavior **encapsulated** inside them while never directly exposing the data they carry as their state. One way to think about objects is to think of what does an object do, or in other words, what operations does an object perform. This way of an object exposing functions or behavior but not its data is referred to as **data hiding**. In its most strict form, all data is hidden away behind methods and only modified or accessed through the methods that an object exposes, i.e., its **public** methods.

## class

Data hiding and encapsulation is conveniently achieved through the class keyword in C++. A class is the twin of a struct, in that it can be used everywhere a struct can be used. However, C++ makes an interesting distinction between the two. Specifically, it considers a struct to be, what is called a **plain-old-datatype** since it is an entity left in the language to preserve compatibility with the C programming language. This resulted in a convention where if all that is required is to store some data together, then a struct is used but if there is a need to define methods and thus behavior, then a class is used. The one important difference between a struct and a class is that by default, members and methods in the former are directly accessible, as we saw in the Customer object earlier, i.e., the members and methods are **public**. In contrast, in a class, members and methods are not directly accessible, and hence **private** by default. public and private are keywords used to specify the kind of access we would like to give other objects of the members and methods in the defining class or struct. The same Customer struct above can be defined using a class as follows,

#include <string\_view>
  
#include <cstdint>
  
  
class Customer
  
{
  
 std::string\_view name;
  
 uint8\_t age;
  
 float credit\_score;
  
  
 public:
  
 Customer(std::string\_view name, uint8\_t age, float cs) :
  
 name{name},
  
 age{age},
  
 credit\_score{cs}
  
 {}
  
};
  
  
auto main() -> int
  
{
  
 Customer ravi {"Ravi", 45, 75.3f};
  
 ravi.name; // error: member is inaccessible
  
}

## Methods

Objects have behavior and this behavior is encapsulated in their public methods. Consider the example below,

#include <string\_view>
  
#include <cstdio>
  
  
class Box
  
{
  
 float l, w, h;
  
  
 public:
  
 Box(float l, float w, float h) : l{l}, w{w}, h{h} {}
  
  
 public:
  
 auto volume() const -> float
  
 {
  
 return l\*w\*h;
  
 }
  
};
  
  
auto main() -> int
  
{
  
 const Box b {2.5f, 1.2f, 4.5f};
  
 const auto v = b.volume();
  
 printf("volume: %f\n", v); // will print: 13.5
  
}

Here, we define a Box class which contains three float members as its data or state. Aside from a constructor, it defines a method to compute its volume. In this simple class, we can see that that the behavior of the Box object b is to compute its volume, which we print and display in the main function.

Note the const suffix in the header of the volume method. Specifying this is necessary if we want to invoke this method on a const object like we do when we instantiate the Box object b in the main function. It is also a promise to the compiler that invoking this method is not going to modify the state of the object.

### Exercises

1. Define a class named Student that contains data associated with their name, id, age and sex. (E)
2. Define a method that puts a pretty-printed version of a Student object into a character array given as its argument. (M)
3. Define an array of Student objects. Write a program that requests the user to enter details of a few students and use these details to construct the Student objects in the array. Then sort them according to their age, and display each student's details using the pretty-print method defined previously. (H)

### Homework

**T2** Chapter **11** - Inheritance and Polymorphism