# Objects I

**T2** Chapter **11**

Working with the C++ programming language, so far, we have come across datatypes like int, float, char and pointers to these types like int\*, float\*. When we needed the ability to store several items of a particular type together, we created arrays like int x[N]. Though this might be sufficient for simple programs of the kind that we create during the course of the module, it is not enough if we want to create complex programs capable of solving real-world problems which can contain data of different kinds.

Take for example, a bank might store information about their customer related to their name, age, credit score etc. Here, the datatypes required to represent each of these variables is already familiar to us - name: const char\*, age: int, credit score: float. It also makes sense that this information needs to be together as a single unit, instead of separate variables so that we can pass this information together between functions for various kinds of processing. To enable us to express this in code, C++ provides us with the ability to define our own types so that we may use them the same way we use an int or a float in code. There are two keywords for this purpose - struct and class, which are identical but for some important differences, which we will look at later in the series of notes.

## struct

#include <string\_view>  
#include <cstdint>  
  
struct Customer  
{  
 std::string\_view name;  
 uint8\_t age;  
 float credit\_score;  
};

We will be working with the above snippet of code, iteratively modifying and adding to it as we learn more concepts related to the idea of an **object**, and what this has to do with defining types.

In the above snippet of code, we use the keyword struct to define a new datatype named Customer which contains the **members** named - name, age and credit\_score. Notice that the purpose of the struct is to provide us with the functionality to store several items together, just like an array, but unlike an array, it enables us to store several items of potentially different types. Also unlike an array, we cannot access the members using the **subscript** [] operator, but instead use the **dot** . operator as follows,

#include <string\_view>  
#include <cstdint>  
  
struct Customer  
{  
 std::string\_view name;  
 uint8\_t age;  
 float credit\_score;  
};  
  
auto main() -> int  
{  
 Customer ravi;  
 ravi.name = "Ravi";  
 ravi.age = 45;  
 ravi.credit\_score = 75.3f;  
}

Here, we **declare** a variable called ravi of type Customer, and **initialize** its members by accessing them individually using the . operator and **assigning** them values.

## Constructors

In C++ jargon, this process of declaring a variable and initializing it is called **construction**. More specifically, it is the construction of an **instance** of the **class** Customer, and such an instance is called an **object**. In other words, an instance of a class is called an object.

Object-oriented programming, in its C++ incarnation, is all about the construction of objects.

Alan Kay, the coiner of the term and the inventor of the Smalltalk programming language, however, meant to emphasize passing messages to objects, which can be understood as applying a function on a given object.

Since construction is a common process, C++ provides a more integrated way of performing this, as follows,

#include <string\_view>  
#include <cstdint>  
  
struct Customer  
{  
 std::string\_view name;  
 uint8\_t age;  
 float credit\_score;  
   
 // Constructor  
 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};  
}

Here, notice the function with the signature Customer(std::string\_view, uint8\_t, float) and how the members of the struct are assigned the values passed in through the arguments to this function. This function has a slightly different syntax to functions that we have come across so far, and it does not specify a return type. Otherwise, it works identically to functions. This function exists inside the struct to serve the purpose of constructing a Customer object, and hence this has a special name called a **constructor**.

As can be seen inside the main function, the Customer object ravi is constructed by invoking the constructor we just defined. This functionality enables us to conveniently create several such objects with different values, and notice that we are using the Customer class just like an int or float type. For all practical purposes, it is sufficient to think of a class as synonymous to a type.

### Exercises

1. Create an array of Customer and construct each item in the array with different values, using the constructor we just defined above. (E)
2. Extend the Customer class to also store the median salary and modify the constructor appropriately to also take a salary argument. (E)
3. Assume that the credit score of a customer is correlated with their age and median salary by the equation where . Modify the previously defined constructor to assign a credit score, the result of this equation. (M)
4. A class can have multiple constructors defined to take different set of arguments to provide the user with flexibility to create an object in various ways. Define a new constructor for the Customer class that uses const char\* instead of std::string\_view and copies the contents of the argument into its member. (H)

## Homework

**T2** Chapter **11**