8.2 — Classes and class members

BY ALEX ON AUGUST 30TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 27TH, 2018

While C++ provides a number of fundamental data types (e.g. char, int, long, float, double, etc…) that are often sufficient for solving relatively simple problems, it can be difficult to solve complex problems using just these types. One of C++'s more useful features is the ability to define your own data types that better correspond to the problem being solved. You have already seen how **enumerated types** and **structs** can be used to create your own custom data types.

Here is an example of a struct used to hold a date:

```
struct DateStruct
{
    int year;
    int month;
    int day;
};
```

Enumerated types and data-only structs (structs that only contain variables) represent the traditional non-object-oriented programming world, as they can only hold data. In C++11, we can create and initialize this struct as follows:

```
DateStruct today { 2020, 10, 14 }; // use uniform initialization
```

Now, if we want to print the date to the screen (something we probably want to do a lot), it makes sense to write a function to do this. Here's a full program:

```
#include <iostream>

struct DateStruct
{
    int year;
    int month;
    int day;
};

void print(DateStruct &date)
{
    std::cout << date.year << "/" << date.month << "/" << date.day;
}

int main()
{
    DateStruct today { 2020, 10, 14 }; // use uniform initialization

    today.day = 16; // use member selection operator to select a member of the struct
    print(today);

    return 0;
}
```

This program prints:

```
2020/10/16
```

Classes

In the world of object-oriented programming, we often want our types to not only hold data, but provide functions that work with the data as well. In C++, this is typically done via the **class** keyword. Using the class keyword defines a new user-defined type called a class.

In C++, classes are very much like data-only structs, except that classes provide much more power and flexibility. In fact, the following struct and class are effectively identical:

```cpp
struct DateStruct
{
    int year;
    int month;
    int day;
};

class DateClass
{
public:
    int m_year;
    int m_month;
    int m_day;
};
```

Note that the only significant difference is the *public:* keyword in the class. We will discuss the function of this keyword in the next lesson.

Just like a struct declaration, a class declaration does not declare any memory. It only defines what the class looks like.

Warning: Just like with structs, one of the easiest mistakes to make in C++ is to forget the semicolon at the end of a class declaration. This will cause a compiler error on the *next* line of code. Modern compilers like Visual Studio 2010 will give you an indication that you may have forgotten a semicolon, but older or less sophisticated compilers may not, which can make the actual error hard to find.

Just like with a struct, to use a class, a variable of that class type must be declared:

```cpp
DateClass today { 2020, 10, 14 }; // declare a variable of class DateClass
```

In C++, when we define a variable of a class, we call it **instantiating** the class. The variable itself is called an **instance**, of the class. A variable of a class type is also called an **object**. Just like how defining a variable of a built-in type (e.g. int x) allocates memory for that variable, instantiating an object (e.g. DateClass today) allocates memory for that object.

Member Functions

In addition to holding data, classes can also contain functions! Functions defined inside of a class are called **member functions** (or sometimes **methods**). Member functions can be defined inside or outside of the class definition. We'll define them inside the class for now (for simplicity), and show how to define them outside the class later.

Here is our Date class with a member function to print the date:

```cpp
class DateClass
{
public:
    int m_year;
    int m_month;
    int m_day;

    void print() // defines a member function named print()
    {
        std::cout << m_year << "/" << m_month << "/" << m_day;
    }
};
```

Just like members of a struct, members (variables and functions) of a class are accessed using the member selector operator (.):

```cpp
#include <iostream>

class DateClass
{
public:
    int m_year;
    int m_month;
    int m_day;

    void print()
    {
        std::cout << m_year << "/" << m_month << "/" << m_day;
    }
};

int main()
{
    DateClass today { 2020, 10, 14 };

    today.m_day = 16; // use member selection operator to select a member variable of the class
    today.print(); // use member selection operator to call a member function of the class

    return 0;
}
```

This prints:

```
2020/10/16
```

Note how similar this program is to the struct version we wrote above.

However, there are a few differences. In the DateStruct version of print() from the example above, we needed to pass the struct itself to the print() function as the first parameter. Otherwise, print() wouldn't know what DateStruct we wanted to use. We then had to reference this parameter inside the function explicitly.

Member functions work slightly differently: All member function calls must be associated with an object of the class. When we call "today.print()", we're telling the compiler to call the print() member function, associated with the today object.

Now let's take a look at the definition of the print member function again:

```cpp
    void print() // defines a member function named print()
    {
        std::cout << m_year << "/" << m_month << "/" << m_day;
    }
```

What do m_year, m_month, and m_day actually refer to? They refer to the associated object (as determined by the caller).

So when we call "today.print()", the compiler interprets `m_day` as `today.m_day`, `m_month` as `today.m_month`, and `m_year` as `today.m_year`. If we called "tomorrow.print()", `m_day` would refer to `tomorrow.m_day` instead.

In this way, the associated object is essentially implicitly passed to the member function. For this reason, it is often called **the implicit object**.

We'll talk more about how the implicit object passing works in detail in a later lesson in this chapter.

The key point is that with non-member functions, we have to pass data to the function to work with. With member functions, we can assume we always have an implicit object of the class to work with!

Using the "m_" prefix for member variables helps distinguish member variables from function parameters or local variables inside member functions. This is useful for several reasons. First, when we see an assignment to a variable with the "m_" prefix, we know that we are changing the state of the class. Second, unlike function parameters or local variables, which are declared within the function, member variables are declared in the class definition. Consequently, if we want to know how a variable with the "m_" prefix is declared, we know that we should look in the class definition instead of within the function.

By convention, class names should begin with an upper-case letter.

Rule: Name your classes starting with a capital letter.

Here's another example of a class:

```cpp
#include <iostream>
#include <string>

class Employee
{
```

```
public:
    std::string m_name;
    int m_id;
    double m_wage;

    // Print employee information to the screen
    void print()
    {
        std::cout << "Name: " << m_name <<
                "  Id: " << m_id <<
                "  Wage: $" << m_wage << '\n';
    }
};

int main()
{
    // Declare two employees
    Employee alex { "Alex", 1, 25.00 };
    Employee joe { "Joe", 2, 22.25 };

    // Print out the employee information
    alex.print();
    joe.print();

    return 0;
}
```

This produces the output:

```
Name: Alex  Id: 1  Wage: $25
Name: Joe  Id: 2  Wage: $22.25
```

Unlike normal functions, the order in which member functions are defined doesn't matter!

A note about structs in C++

In C, structs can only hold data, and do not have associated member functions. In C++, after designing classes (using the class keyword), Bjarne Stroustrup spent some amount of time considering whether structs (which were inherited from C) should be granted the same capabilities. Upon consideration, he determined that they should, in part to have a unified ruleset for both. So although we wrote the above programs using the class keyword, we could have used the struct keyword instead.

Many developers (including myself) feel this was the incorrect decision to be made, as it can lead to dangerous assumptions: For example, it's fair to assume a class will clean up after itself (e.g. a class that allocates memory will deallocate it before being destroyed), but it's not safe to assume a struct will. Consequently, we recommend using the struct keyword for data-only structures, and the class keyword for defining objects that require both data and functions to be bundled together.

Rule: Use the struct keyword for data-only structures. Use the class keyword for objects that have both data and functions.

You have already been using classes without knowing it

It turns out that the C++ standard library is full of classes that have been created for your benefit. std::string, std::vector, and std::array are all class types! So when you create an object of any of these types, you're instantiating a class object. And when you call invoke a function using these objects, you're calling a member function.

```
#include <string>
#include <array>
#include <vector>
#include <iostream>

int main()
{
    std::string s { "Hello, world!" }; // instantiate a string class object
    std::array<int, 3> a { 1, 2, 3 }; // instantiate an array class object
    std::vector<double> v { 1.1, 2.2, 3.3 }; // instantiate a vector class object

    std::cout << "length: " << s.length() << '\n'; // call a member function

    return 0;
}
```

Conclusion

The class keyword lets us create a custom type in C++ that can contain both member variables and member functions. Classes form the basis for Object-oriented programming, and we'll spend the rest of this chapter and many of the future chapters exploring all they have to offer!

Quiz time

1a) Create a class called IntPair that holds two integers. This class should have two member variables to hold the integers. You should also create two member functions: one named "set" that will let you assign values to the integers, and one named "print" that will print the values of the variables.

The following main function should execute:

```
int main()
{
    IntPair p1;
    p1.set(1, 1); // set p1 values to (1, 1)

    IntPair p2{ 2, 2 }; // initialize p2 values to (2, 2)

    p1.print();
    p2.print();

    return 0;
}
```

and produce the output:

```
Pair(1, 1)
Pair(2, 2)
```

Hide Solution

```
#include <iostream>

class IntPair
{
public:
    int m_first;
    int m_second;

    void set(int first, int second)
    {
        m_first = first;
        m_second = second;
    }
    void print()
    {
        std::cout << "Pair(" << m_first << ", " << m_second << ")\n";
    }
};

int main()
{
    IntPair p1;
    p1.set(1, 1);

    IntPair p2{ 2, 2 };

    p1.print();
    p2.print();

    return 0;
}
```

(h/t to reader Pashka2107 for this quiz idea)

1b) Why should we use a class for IntPair instead of a struct?

Hide Solution

This object contains both member data and member functions, so we should use a class. We should not use structs for objects that have member functions.

8.3 -- Public vs private access specifiers

Index

8.1 -- Welcome to object-oriented programming

Share this:

133 comments to 8.2 — Classes and class members

« Older Comments 1 2

erad
July 8, 2018 at 3:48 pm · Reply

While I know that you haven't yet dealt with constructors (btw, I do have some prior knowledge), how is it that Line 26 (in Quiz 1a Solution) is possible since no constructors were declared/defined for class IntPair? Normally, if no constructors are defined for a class, the compiler implicitly defines a "default" constructor for it; however, in your solution, object p2 has arguments as it was initialized with two integers values!

Why wouldn't this line generate a compiler error?

nascardriver
July 9, 2018 at 4:58 am · Reply

Hi erad!

You can use aggregate initalization to initialize classes just like structs.
Even though it's possible, I wouldn't do it in practice. A proper constructor is way easier to get behind.

References:
* Lesson 4.7 - Structs
* Aggregate initialization: https://en.cppreference.com/w/cpp/language/aggregate_initialization