Contents

[Getting Classy with C++ 1](#_Toc57060584)

[Class Members 1](#_Toc57060585)

[Creating Objects 2](#_Toc57060586)

[Access Control: Public and Private 3](#_Toc57060587)

[Constructors 4](#_Toc57060588)

[Destructors 5](#_Toc57060589)

## Getting Classy with C++

So far, we’ve used several data types, including int, double, std::string, and bool. When we work with int or std::string, we can create variables with certain properties and methods specific to those types. For example:

int age = 33;  
age++; // age is now 34

But what happens when you want to create a “type” for something else? You can make your own! Bjarne Stroustrup developed C++ because he wanted to add a feature known as “classes” to the C language. A C++ *class* is a user-defined type.

The class serves as a blueprint for *objects*, which are instances of the class (just like age is an instance of int). An object gets characteristics and behaviors from its class.

We can create an empty C++ class like this in a header file:

class City {  
   
}; // <-- notice this semicolon!

**Fun fact:** C++’s original name was “C with Classes.”

## Class Members

An empty class is pretty useless. Classes are designed to bring together related information and functionality — time to add stuff inside!

Components of a class are called *class members*. Just like you can get a string’s length using .length(), you can access class members using the dot operator (object.class\_member).

There are two types of class members:

* *Attributes*, also known as *member data*, consist of information about an instance of the class.
* *Methods*, also known as *member functions*, are functions that you can use with an instance of the class. We use a . before method names to distinguish them from regular functions.

We *encapsulate* — or enclose for simpler user access — attributes and methods in a class like this:

class City {  
   
  // attribute  
  int population;  
   
// we'll explain 'public' later  
public:  
  // method  
  void add\_resident() {  
    population++;  
  }  
   
};

Unless we have a mostly empty class, it’s common to split function declarations from definitions. We declare methods inside the class (in a header), then define the methods outside the class (in a **.cpp** file of the same name).

How can we define methods outside a class? We can do this using ClassName:: before the method name to indicate its class like this:

int City::get\_population() {  
  return population;  
}

Unlike with regular functions, we need to include the header file in the **.cpp** file where we define the methods.

**Note:** We must declare a method inside the class if we want to define it outside.

## Creating Objects

Now that you have a class, let’s create some objects! To refresh your memory, an object is an instance of a class, which encapsulates data and functionality pertaining to that data.

To create (or instantiate) an object, we can do this:

City accra;

We can give the object’s attributes values like this (note that these must be attributes you defined in the class):

accra.population = 2270000;

Later, we can access this information using the method we added to the City class (if it’s in a public part of the class):

accra.get\_population();

## Access Control: Public and Private

Let’s circle back to that public keyword. What was that about?

By default, everything in a class is private, meaning class members are limited to the scope of the class. This makes it easier to keep data from mistakenly being altered, and abstracts away all the nitty gritty details. If you try to access a private class member, you’ll get an error:

error: 'population' is a private member of 'City'

But sometimes you need access to class members, and for that, there is public. You can use it to make everything below accessible outside the class:

class City {  
   
  int population;   
   
public: // stuff below is public  
  void add\_resident() {   
    population++;  
  }  
   
};

There is also a private access modifier for when you want something below public to be private to the class:

class City {  
   
  int population;   
   
public:  
  void add\_resident() {   
    population++;  
  }  
   
private: // this stuff is private  
  bool is\_capital;  
   
};

## Constructors

Is there a way to give an object some data right when it gets created? We’re so glad you asked!

A *constructor* is a special kind of method that lets you decide how the objects of a class get created. It has the same name as the class and no return type. Constructors really shine when you want to instantiate an object with specific attributes.

If we want to make sure each City is created with a name and a population, we can use parameters and a member initializer list to initialize attributes to values passed in:

// city.hpp  
#include "city.hpp"  
   
class City {  
   
  std::string name;  
  int population;  
   
public:  
  City(std::string new\_name, int new\_pop);  
   
};  
   
// city.cpp  
City::City(std::string new\_name, int new\_pop)  
  // members get initialized to values passed in   
  : name(new\_name), population(new\_pop) {}

You could also write the definition like this:

City::City(std::string new\_name, int new\_pop) {  
  name = new\_name;  
  population = new\_pop;  
}

However, a member initialization list allows you to directly initialize each member when it gets created.

To instantiate an object with attributes, you can do:

// inside main()  
City ankara("Ankara", 5445000);

Now we have a City called ankara with the following attributes:

* ankara.name set to "Ankara".
* ankara.population set to 5445000.

## Destructors

So far, you’ve learned how to create and use objects. But there’s another part of the object lifecycle we need to cover: how to destroy them! Muahahaha.

It’s actually far less nefarious than it sounds; object destruction is really about tidying up and preventing memory leaks. A *destructor* is a special method that handles object destruction. Like a constructor, it has the same name as the class and no return type, but is preceded by a ~ operator and takes no parameters:

City::~City() {  
   
  // any final cleanup  
   
}

Inside you add any housekeeping that needs to happen before the object is destroyed. You generally won’t need to call a destructor; the destructor will be called automatically in any of the following scenarios:

* The object moves out of scope.
* The object is explicitly deleted.
* When the program ends.

Review

You’ve learned the basics of classes and objects in C++:

* Classes are user-defined types.
* Objects are instances of classes.
* Class members are the data attributes and functions inside of a class.
* Creating a new object from a class is called instantiation.
* Class members can be designated as either private or public — they are private by default.
* You can create a constructor to instantiate objects in a particular way.
* A destructor allows you to execute any cleanup necessary before an object gets destroyed.