CHAPTER 2

Fundamentals of C++ for Robotics Programming

In the last chapter, we went through detailed procedures to install Ubuntu on VirtualBox and on a real PC. We also practiced important shell commands that we are going to use while building a robot. The next important requirement for working with a robot is to learn a few programming languages. By using these languages, we can program the robot for different application. Some of the popular programming languages used for creating robotic applications are C++ and Python. This doesn't mean that we won't use other languages. Programming languages like Java and C# are also used in robotics, but the most common languages are C++ and Python.

This chapter discusses some fundamental concepts of C++ and its compilation process. These concepts will definitely help you when you start working with ROS. The fundamentals include mainly object-oriented programming (OOP) concepts and compiling code using Make and CMake tools. This chapter assumes that you have some fundamental understanding of C programming languages. So let's get started with C++ fundamental.

L. Joseph, *Robot Operating System for Absolute Beginners*,
https://doi.org/10.1007/978-1-4842-3405-1_2

Getting Started with C++

We can define C++ as a superset of the C programming language, or we can say "C with classes." The C++ programming language project, initially called C with Classes, was started in 1979 by computer programmer Bjarne Stroustrup. His main work was adding object-oriented programming into the C language by maintaining its portability without sacrificing speed or low-level functionality. Like C, C++ is a compiled language. It needs a compiler to convert the source code into executable code.

Timeline: The C++ Language

In 1983, the C with Classes project changed to C++. The ++ operator is used for incrementing a variable, so C++ means it is the C language with new features. In 1990, Borland's Turbo C++ compiler released as a commercial product. In 1998, C++ standards were published as C++ ISO/IEC 14882:1992 or C++98. In 2005, the C++ standards committee released a report of new features added to the latest C++ standard. In 2011, the new C++ standards were completed. The Boost libraries (`www.boost.org`) made a considerable impact on the new standards. Boost C++ Libraries is a set of libraries for the C++ programming that provides support for tasks and structures, such as linear algebra, multithreading, image processing, regular expressions, and unit testing.

C/C++ in Ubuntu Linux

Ubuntu Linux comes with an in-built C/C++ compiler called GCC/G++. GCC stands for GNU Compiler Collection. It includes compilers for C, C++, Objective-C, Fortran, Ada, and Go, as well as libraries for these languages. GCC was written for the GNU Project (`www.gnu.org/gnu/thegnuproject.html`) by Richard Stallman.

Introduction to GCC and G++ Compilers

Let's start with GCC/G++ compilers. The latest Ubuntu Linux comes with preinstalled C and C++ compilers. The C compiler in Linux is GCC, and the C++ compiler is G++, the `gcc` and `g++` are shell commands of this compilers. You can type this command in the terminal to see what happens (see Figure 2-1).

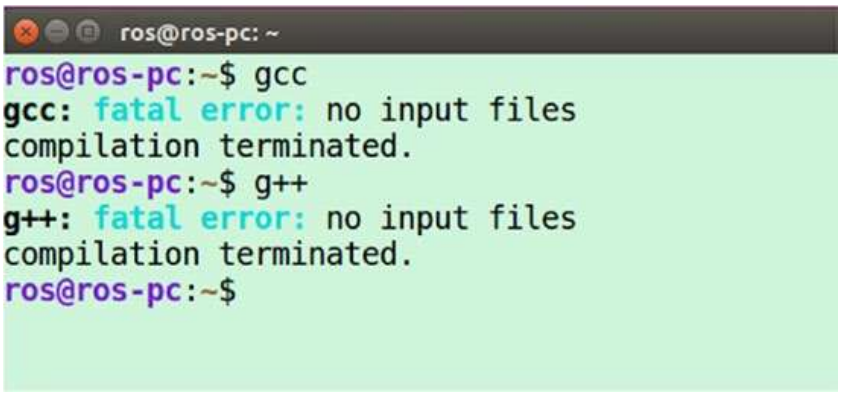

Figure 2-1. *Testing gcc and g++ commands in the terminal*

If you are not getting the message shown in Figure 2-1, then you confirm that these compilers are not preinstalled in your system. No worries! You can install these compilers using `apt-get` command.

Installing C/C++ Compiler

First, you may need to update the list of Ubuntu packages from the repository with the following command.

```
$ sudo apt-get update
```

Now install the packages for getting the compilers.

```
$ sudo apt-get install build-essential manpages-dev
```

The `build-essential` package is associated with numerous packages for developing software in Ubuntu Linux.

Verifying Installation

After installing the preceding package, you can verify whether the installation is correct by using the following commands.

```
$ whereis gcc
$ whereis g++
```

These commands locate the path of the gcc/g++ command and the manual page of the same command.

The following commands print the GCC compiler that we are going to use and display the path of the command.

```
$ which gcc
$ which g++
```

The following commands print the current version of GCC that we are going to use.

```
$ gcc   version
$ g++   version
```

Figure 2-2 shows the output of the preceding commands.

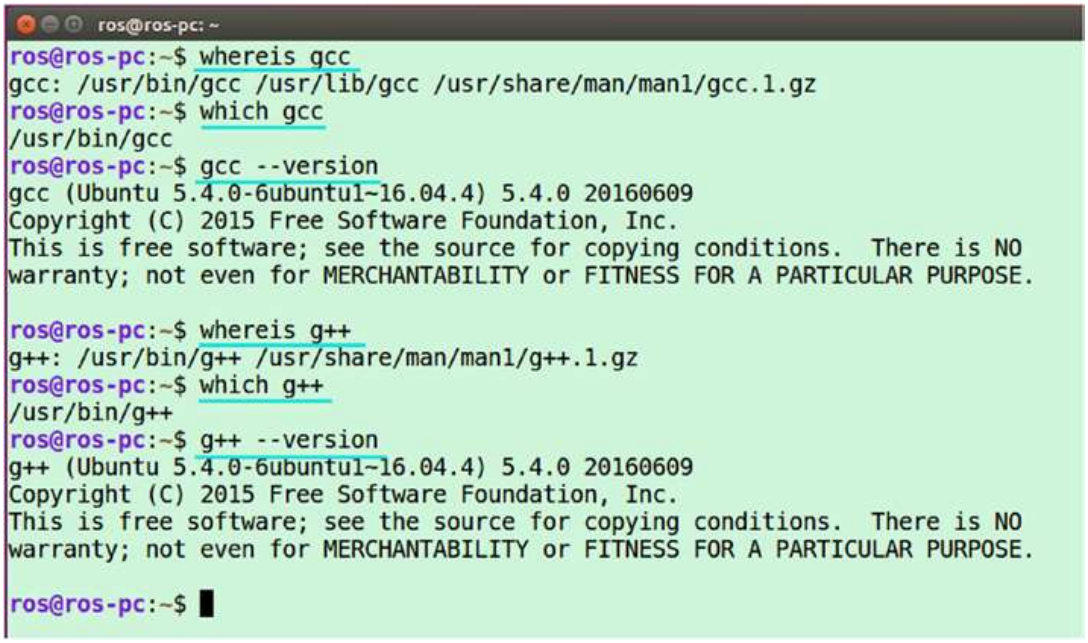

Figure 2-2. *Testing gcc and g++ commands in the terminal*

Introduction to GNU Project Debugger (GDB)

Let's have a look at debugger tools for C/C++. So, what is a debugger? A debugger is a program that runs and controls another program, examining each line of code to detect problems or bugs.

The Ubuntu Linux comes with a debugger called GNU Debugger, which is also called GDB (www.gnu.org/software/gdb/). It is one of the popular C and C++ program debuggers for the Linux system.

Installing GDB in Ubuntu Linux

Here is the command to install GDB in Ubuntu. It's already installed on the latest version of Ubuntu. If you are using other versions, you can use the following command to install it.

```
$ sudo apt-get install gdb
```

Verifying Installation

To check whether GDB is installed properly on your PC, use the following command. Once you type **gdb** in your terminal, the message in Figure 2-3 is shown.

```
$ gdb
```

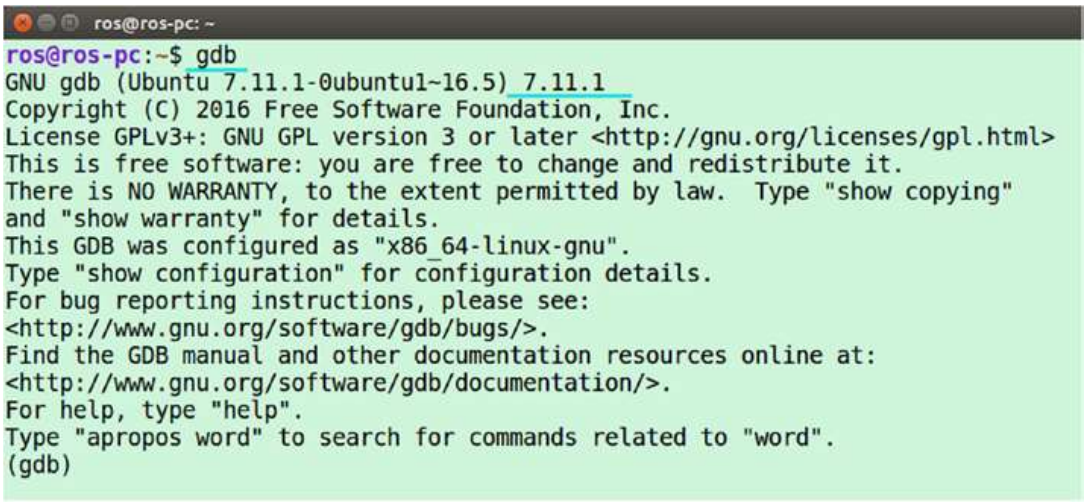

Figure 2-3. *Testing the gdb command*

You can verify the gdb version by using the following command.

```
$ gdb version
```

The version also shows when you enter the gdb command.

In the next section, we are going to write our first C++ code in Ubuntu. We will compile it and debug it to find bugs in the code.

Writing Your First Code

Let's start writing the first program in Ubuntu Linux. To write the code, you can use a text editor in Ubuntu. You can choose either the gedit or nano terminal text editor. gedit is a popular GUI text editor in Ubuntu. We already worked with nano in the first chapter, so now let's check out gedit.

In Ubuntu, search for gedit (see Figure 2-4) and select from the search results.

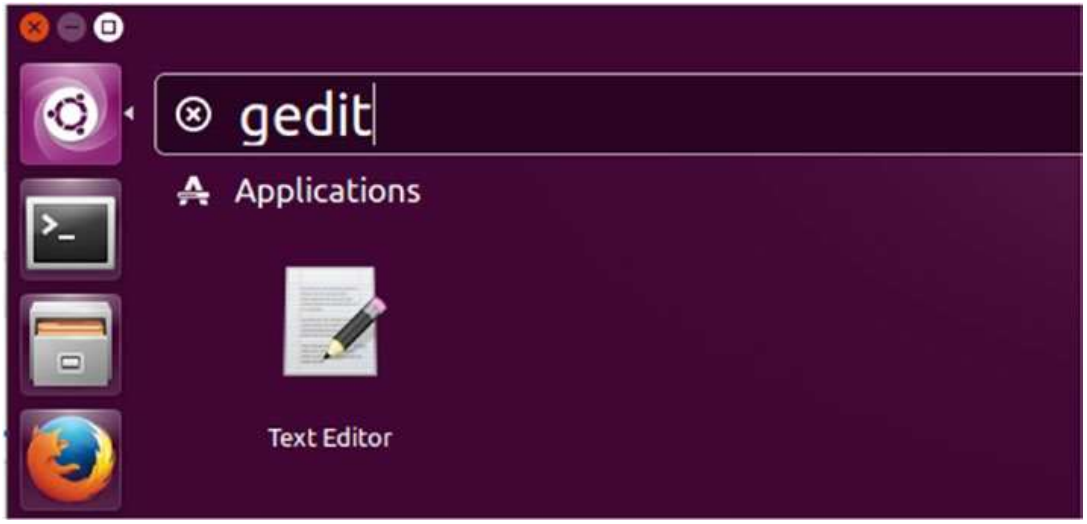

Figure 2-4. *Searching for the gedit text editor in Ubuntu search*

Once you click the text editor, you see the window shown in Figure 2-5.

Figure 2-5. *The gedit text editor*

This editor is very similar to Notepad or WordPad in Windows. You can write your first C++ code in this text editor.

Figure 2-6 shows the first C++ code that we are going to compile in the Linux.

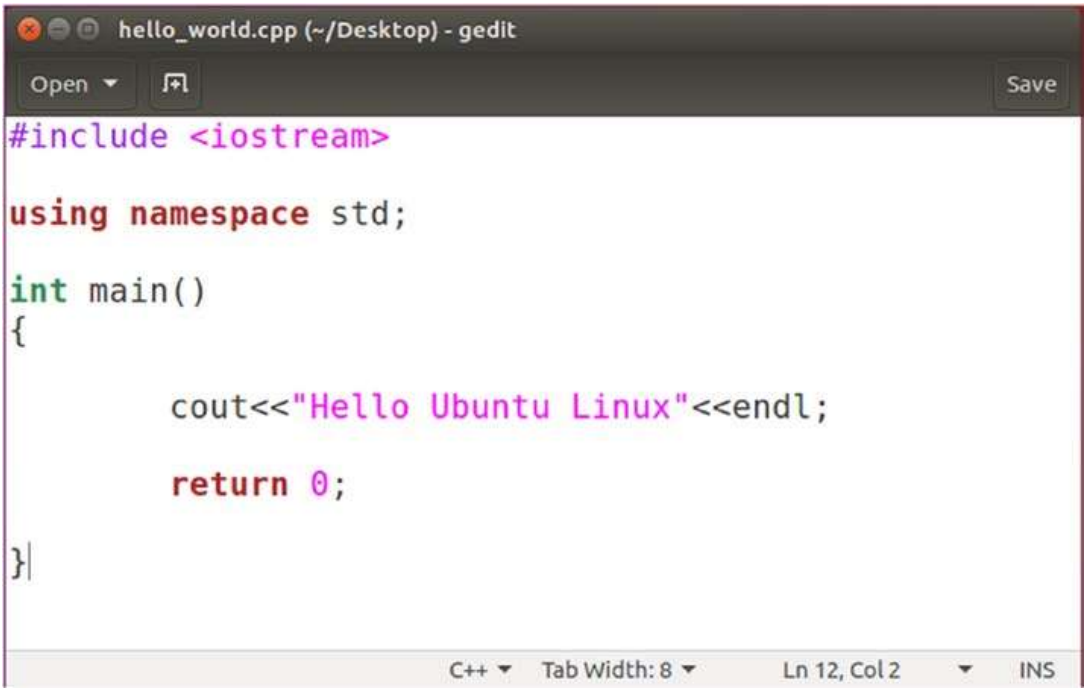

Figure 2-6. *The gedit text editor*

Write the code in the text editor and save it as `hello_world.cpp`.

Explaining Code

The `hello_world.cpp` code is going to print the message, "Hello Ubuntu Linux". `#include <iostream>` is a C++ header file for input/output functions, such as taking input from a keyboard or printing a message. In this program, we are only using the print function to print messages, so `iostream` will be enough. The next line is `using namespace std`.

The namespace (www.geeksforgeeks.org/namespace-in-c/) is a special feature in C++ to group a set of entities. The `std` namespace is used in the `iostream` library. When we `using namespace std`, we can access the functions or other entities included in the `std` namespace, such as functions like `cout` and `cin`. If we are not using this line of code, you have to mention `std::` for accessing functions inside that namespace; for example, `std::cout` is a function to print a message.

After discussing the header file and other lines code, we can discuss what is included in the main function. We are using `cout<<"Hello Ubuntu Linux"<<endl` to print that message. The `endl` adds a new line after printing the message. After printing the message, the function returns 0 and exits the program.

Compiling Your Code

After saving your code, the next step is to compile the code. The following procedure will help you to compile the code.

You can take a new terminal and switch the terminal path to the folder where the code is saved. In this case, we have saved the code to `/home//Desktop` folder. To change the terminal path to the `Desktop` folder, you have to use the `'cd'` command like shown below.

```
$ cd Desktop
```

If you have saved your code in the `home` directory, you don't need to run this command.

After switching to the `Desktop` folder, type **`ls`** to list the files in it (see Figure 2-7).

```
$ ls
```

Figure 2-7. *Listing the files in the Desktop folder*

If your code is in the folder, you can do the compilation by using the following command.

```
$ g++ hello_world.cpp
```

The G++ compiler checks the code, and if there is no error, it creates an executable named `a.out`. You can execute this file by using the following command (see Figure 2-8).

```
$ ./a.out
```

It shows the output as

```
Hello Ubuntu Linux
```

Figure 2-8. *Running the output executable*

Congratulations! You have successfully compiled and executed your first C++ code. Now let's check some of the g++ options. This will be useful in the upcoming sections.

If you want to create an executable with a particular name, you can use the following command.

```
$ g++ hello_world.cpp -o hello_world
```

The `-o` argument points out the output executable name. So, the preceding command creates an executable named `hello_world`. You can execute it by using the following command.

```
$./hello_world
```

The output of the preceding commands is shown in Figure 2-9.

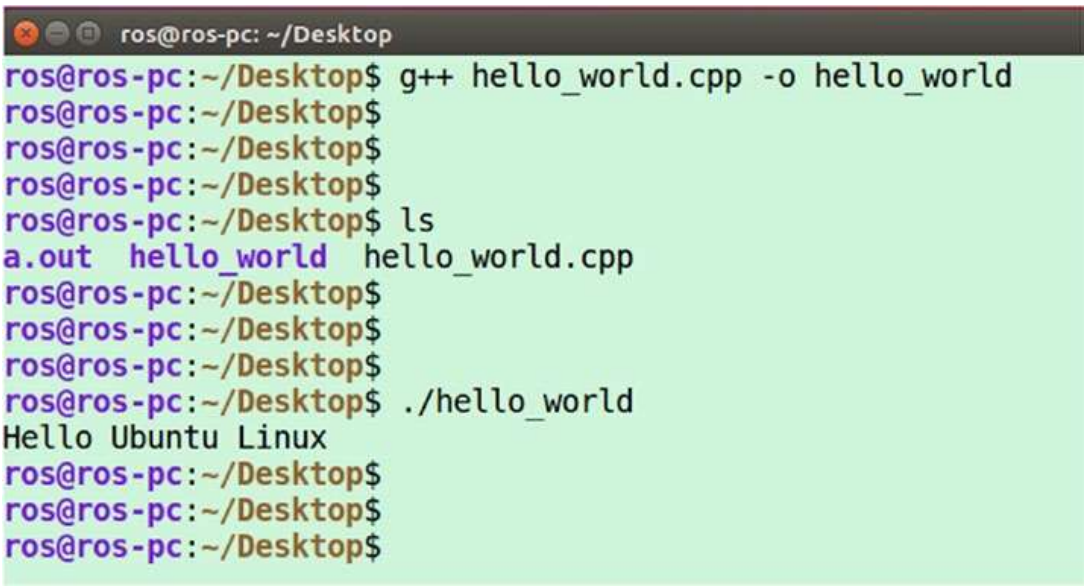

Figure 2-9. *Running the hello_world output executable*

Debugging Your Code

Using the debugger tool, we can go through each line of code and inspect the values of each variable. Figure 2-10 shows C++ code to compute the sum of two variables. Let's save this code as `sum.cpp`.

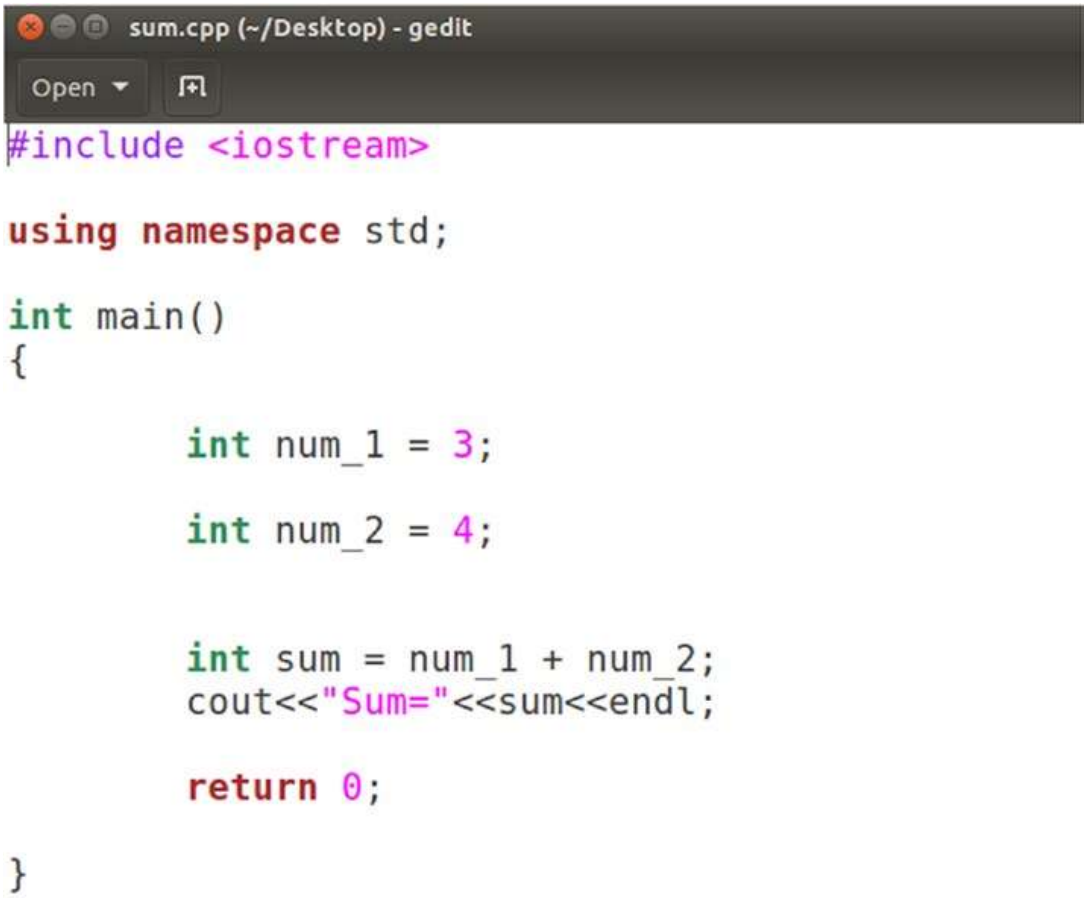

Figure 2-10. *C++ code for summing two numbers*

To debug/inspect each line of code, you have to compile the `sum.cpp` using g++ with the `-g` option. This builds the code with debugging symbols and enables it to work with GDB.

The following command helps to compile the code with debug symbols.

```
$ g++ -g sum.cpp -o sum
```

After compiling, you can execute it by running the following command.

```
$. /sum
```

For debugging, use GDB. The output of the preceding set of commands is shown in Figure 2-11.

Figure 2-11. *Compiling sum.cpp*

After creating the executable, you can debug the executable by using following the command.

```
$ gdb sum
```

`sum` is the name of the executable. After entering the command, you have to use the GDB commands to proceed with debugging. The following are important GDB commands that you need to remember.

- `b line_numer`: Creates a break point in the given line number. While debugging, the debugger stops at this break point.
- `n`: Executes the next line of code.
- `r`: Runs the program until the break point.
- `p variable_name`: Prints the value of a variable.
- `q`: Exits the debugger.

Let's try these commands. The output of each command is shown in Figure 2-12.

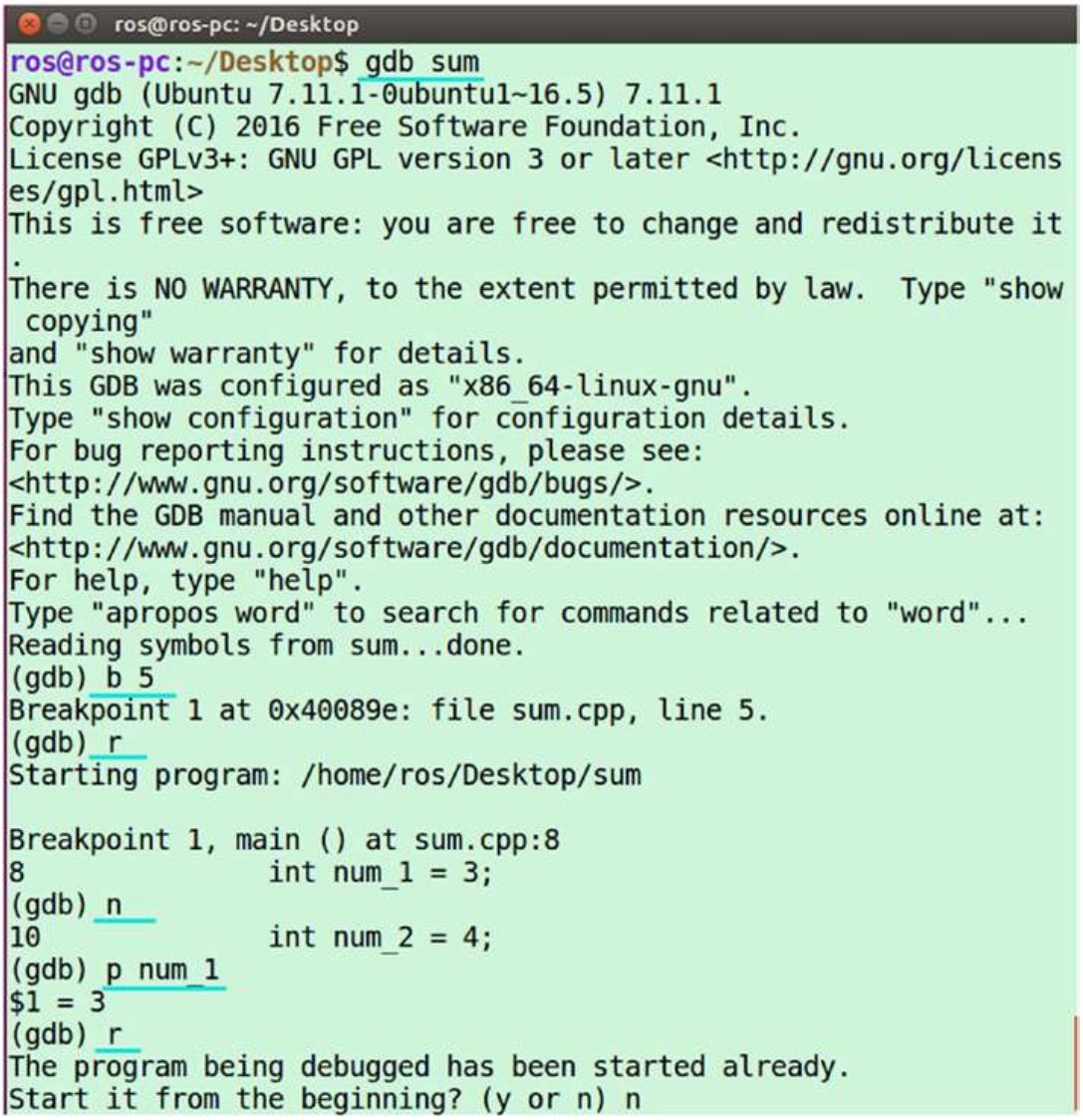

Figure 2-12. *Debugging sum application*

Now that you've learned the basics of compiling and debugging, let's start learning the basics of OPP concepts in C++. The following section discusses some of the important concepts that are required knowledge in the upcoming chapters.

Learning OOP Concepts from Examples

If you already know C structures, then learning about OOP concepts will not take much time. In C structures, we can group different data types—such as integer, float, and string—into a single, user-defined data type. Similar to structures, C++ has an enhanced version of structs that has a provision to define functions. This enhanced struct version is called the C++ class. Each instance of the C++ Class is called an *object*. An object is simply a copy of the actual class. There are several properties associated with objects, which are called *object-oriented programming concepts*. The main OOP concepts are explained with C++ code next.

The Differences Between Classes and Structs

Before going through the OOP concepts, let's look at the basic differences between a struct and a class. Listing 2-1 helps differentiate them.

Listing 2-1. Example Code to Demonstrate C++ Class and Struct

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

using namespace std;

struct Robot_Struct

{
        int id;
        int no_wheels;
        string robot_name;

};

class Robot_Class
{
```

```
public:
        int id;
        int no_wheels;

        string robot_name;

        void move_robot();

        void stop_robot();

};

void Robot_Class::move_robot()
{

        cout<<"Moving Robot"<<endl;

}


void Robot_Class::stop_robot()
{

        cout<<"Stopping Robot"<<endl;
}

int main()

{
        Robot_Struct robot_1;
        Robot_Class robot_2;

        robot_1.id = 2;
        robot_1.robot_name = "Mobile robot";

        robot_2.id = 3;
        robot_2.robot_name = "Humanoid robot";

        cout<<"ID="<<robot_1.id<<"\t"<<"Robot       Name"<<robot_1.
        robot_name<<endl;
```

```
        cout<<"ID="<<robot_2.id<<"\t"<<"Robot Name"<<robot_2.
        robot_name<<endl;

        robot_2.move_robot();
        robot_2.stop_robot();

        return 0;

}
```

This code defines a struct and a class. The struct name is `Robot_Struct` and the class name is `Robot_Class`.

Figure 2-13 shows how to define a structure. It defines a struct with variables such as id, name, and the number of wheels.

```
struct Robot_Struct

{
        int id;
        int no_wheels;
        string robot_name;


};
```

Figure 2-13. *Defining a structure in C++*

As you know, a struct has a name, and the declaration of all the variables are inside it. Let's check the definition of a class (see Figure 2-14).

```
class Robot_Class
{


public:
        int id;
        int no_wheels;

        string robot_name;

        void move_robot();

        void stop_robot();


};
```

Figure 2-14. *Defining a Class in C++*

So, what is the difference between the two? A struct can only define a different variables, but a class can define different variables and declare functions too. The class shown in Figure 2-14 declares two functions along with the variables. So where is the definition of each function? We can either define the function inside the class or outside the class. The standard practice is to keep the definition external to the class definition to keep the class definition short.

Figure 2-15 shows the definitions of functions mentioned inside the class.

```
void Robot_Class::move_robot()
{

        cout<<"Moving Robot"<<endl;

}

void Robot_Class::stop_robot()
{

        cout<<"Stopping Robot"<<endl;
}
```

Figure 2-15. *External definition of function inside the class*

In the function definition, the first term is the return data type, followed by the class name, and then the function name followed by `::`, which states that the function is inside the class. Inside the function definition, we can add our code. This particular code prints a message.

You have seen the function definition inside a class. The next step is to learn how to read/write to variables and functions.

C++ Classes and Objects

This section explains how to read/write to structs and classes. Figure 2-16 shows lines of code that do the job.

```
Robot_Struct robot_1;
Robot_Class robot_2;



robot_1.id = 2;
robot_1.robot_name = "Mobile robot";


robot_2.id = 3;
robot_2.robot_name = "Humanoid robot";
```

Figure 2-16. *Creating struct and class instances*

Similar to the struct instance, we can create an instance of a class and that is called an object.

Let's look at `Robot_Class robot_2;` here, `robot_2` is an object and `robot_1` is an instance of the structure. Using this instance or object, we can access each variable and function. We can use the `.` operator to access each variable. The struct and class variables are accessed by using the `.` operator. If you use struct or class pointers, you have to use the `->` operator to access each variable. Listing 2-2 is an example.

Listing 2-2. Creating a C++ Object and Accessing Object by Reference

```
Robot_Class *robot_2;
robot_2 = new Robot_Class;
robot_2->id - 2;
robot_2->name = "Humanoid Robot";
```

The `new` operator allocates memory for the C++ object. We can access the functions inside the class and print all values by using the `.` operator. Figure 2-17 shows how to do that.

```
cout<<"ID="<<robot_1.id<<"\t"<<"Robot Name"<<robot_1.robot_name<<endl;

cout<<"ID="<<robot_2.id<<"\t"<<"Robot Name"<<robot_2.robot_name<<endl;

robot_2.move_robot();
robot_2.stop_robot();
```

Figure 2-17. *Printing values and calling functions*

We can save the code as `class_struct.cpp`, and compile it by using the following command.

```
$ g++ class_struct.cpp -o class_struct
$. /class_struct
```

Figure 2-18 shows the output of the code.

```
ros@ros-pc:~$ ./class_struct
ID=2    Robot NameMobile robot
ID=3    Robot NameHumanoid robot
Moving Robot
Stopping Robot
ros@ros-pc:~$
```

Figure 2-18. *Output of the program*

For further reference, go to www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm.

Class Access Modifier

Inside the class, you may have seen a keyword called `public:`. It is called an *access modifier*. Figure 2-19 is a code snippet of the access modifier used in Listing 2-1.

```
class Robot_Class
{


public:
        int id;
        int no_wheels;
```

Figure 2-19. *Public access keyword usage*

This feature is also called *data hiding*. By setting the access modifier, we can limit the usage of functions defined inside it. There are three types of access modifiers in a class.

- `public`: A public member can access from anywhere outside the class within a program. We can directly access the public variable without even writing functions.
- `private`: Variables or functions cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.
- `protected`: Access is very similar to private members, but the difference is the child class can access the members. The concepts of child class/ derived class are discussed in the upcoming section.

Access modifiers help you group variables, which you can keep visible or hidden in the class.

C++ Inheritance

Inheritance is another important concept in OOP. If you have two or more classes, and you want to have the functions inside those classes in a new class, you can use the inheritance property. By using the inheritance property, you can reuse the function inside the existing classes in a new class. The new class that is going to inherit an existing class is called a *derived class*. The existing class is called a *base class*.

A class can be inherited through public, protected, or private inheritance. The following explains each type of inheritance.

- *Public inheritance*: When we derive a class from a public base class, the public members of the base class become public members of the derived class, and protected members of the base class become protected members of the derived class. The private members of the base class can never be accessed in the derived class. It can access through calls to the public and protected members of the base class.

- *Protected inheritance*: When we inherit using the protected base class, the public and protected members of the base class become protected members of the derived class.
- *Private inheritance*: When deriving from a private base class, public and protected members of the base class become private members of the derived class.

Listing 2-3 gives a simple example of public inheritance.

Listing 2-3. Example of C++ Public Inheritance

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

using namespace std;

class Robot_Class
{

public:
       int id;
       int no_wheels;

       string robot_name;

       void move_robot();

       void stop_robot();

};

class Robot_Class_Derived: public Robot_Class
{

public:

       void turn_left();
```

```
        void turn_right();
};
void Robot_Class::move_robot()
{
        cout<<"Moving Robot"<<endl;
}
void Robot_Class::stop_robot()
{
        cout<<"Stopping Robot"<<endl;
}
void Robot_Class_Derived::turn_left()
{
        cout<<"Robot Turn left"<<endl;
}
void Robot_Class_Derived::turn_right()
{
        cout<<"Robot Turn Right"<<endl;
}
int main()
{
        Robot_Class_Derived robot;
        robot.id = 2;
        robot.robot_name = "Mobile robot";
        cout<<"Robot ID="<<robot.id<<endl;
        cout<<"Robot Name="<<robot.robot_name<<endl;
```

```
        robot.move_robot();
        robot.stop_robot();

        robot.turn_left();
        robot.turn_right();

        return 0;

}
```

So in this example we are creating a new class called `Robot_Class_Derived`, which is derived from a base class called `Robot_Class`. The public inheritance is done using a public keyword followed by the base class name (see Figure 2-20). There should be a `:` after the derived class name, followed by a public keyword and a base class name.

```
class Robot_Class_Derived: public Robot_Class
{

public:

        void turn_left();

        void turn_right();


};
```

Figure 2-20. *Code snippet of public inheritance*

If you chose public inheritance, you can access the public and protected variables and functions of the base class; in this case, `Robot_Class`.

We are using the same class that we used in the first example. The definition of each function in the derived class is given in Figure 2-21.

```
void Robot_Class_Derived::turn_left()
{

        cout<<"Robot Turn left"<<endl;

}


void Robot_Class_Derived::turn_right()
{

        cout<<"Robot Turn Right"<<endl;

}
```

Figure 2-21. *Function definition inside a derived class*

Now let's look at how to access the functions inside the derived class (see Figure 2-22).

```
Robot_Class_Derived robot;


robot.id = 2;
robot.robot_name = "Mobile robot";


cout<<"Robot ID="<<robot.id<<endl;
cout<<"Robot Name="<<robot.robot_name<<endl;


robot.move_robot();
robot.stop_robot();


robot.turn_left();
robot.turn_right();
```

Figure 2-22. *Accessing the derived class object*

Here we are creating an object of `'Robo_Class_Derived'` called `'robot'`. If you go through the code, you can understand that we didn't declare `id` and `robot_name` variables in the `Robo_Class_Derived`, but it was defined in the `Robo_Class`. Using inheritance property, we can access the variable of `Robo_Class` inside its derived class.

Let's look at the output of the code. We can save this code as `class_inherit.cpp` and compile it by using the following command.

```
$ g++ class_inherit.cpp -o class_inherit
./class_inherit
```

This gives you the output shown in Figure 2-23, without showing any errors. This means that the public inheritance is working fine.

```
ros@ros-pc:~$ ./class_struct
Robot ID=2
Robot Name=Mobile robot
Moving Robot
Stopping Robot
Robot Turn left
Robot Turn Right
ros@ros-pc:~$
```

Figure 2-23. *Output of a derived class program*

If you look at the output, we are getting all the messages from functions, defined in the base class and the derived class. We can also access the base class variables and set the values.

We have covered some important OOP concepts. To explore more concepts, refer to `www.tutorialspoint.com/cplusplus`.

C++ Files and Streams

Let's discuss file operation in C++ and how to read/write data to a file. We have already discussed the `iostream` header for doing file operations. We need another standard C++ library called `fstream`. The following three data types are inside `fstream`.

- `ofstream`: Stands for *output file stream*. It is used to create a file and to write data into it.
- `ifstream`: Represents an input file stream. It is used to read data from files.
- `fstream`: Has both read and write capabilities.

Listing 2-4 demonstrates writing and reading a file using C++ functions.

Listing 2-4. Example C++ Code to Read/Write from a File

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{

        ofstream out_file;
        string data = "Robot_ID=0";
        cout<<"Write data:"<<data<<endl;
        out_file.open("Config.txt");
        out_file <<data<<endl;
        out_file.close();

        ifstream in_file;
        in_file.open("Config.txt");
        in_file >> data;
```

```
        cout<<"Read data:"<<data<<endl;
        in_file.close();

        return 0;
}
```

We have to include the `fstream` header to get the read/write data type in C++. We have created an `ofstream` class object, and in that object, there is a function called `open ()` to open a file. After opening the file, we can write to it by using the `<<` operator. After writing the data, we close the file for a reading operation. For reading, we are using the `ifstream` class object in C++ and opening the file with the `open("file_name")` function inside the `ifstream` class. After opening the file, we can read from the file by using the `>>` operator. After reading, it is printed on the terminal. The file name that we are going to write is `Config.txt` and the data is a robot parameter. Figure 2-24 shows the output if we compile the code and run it.

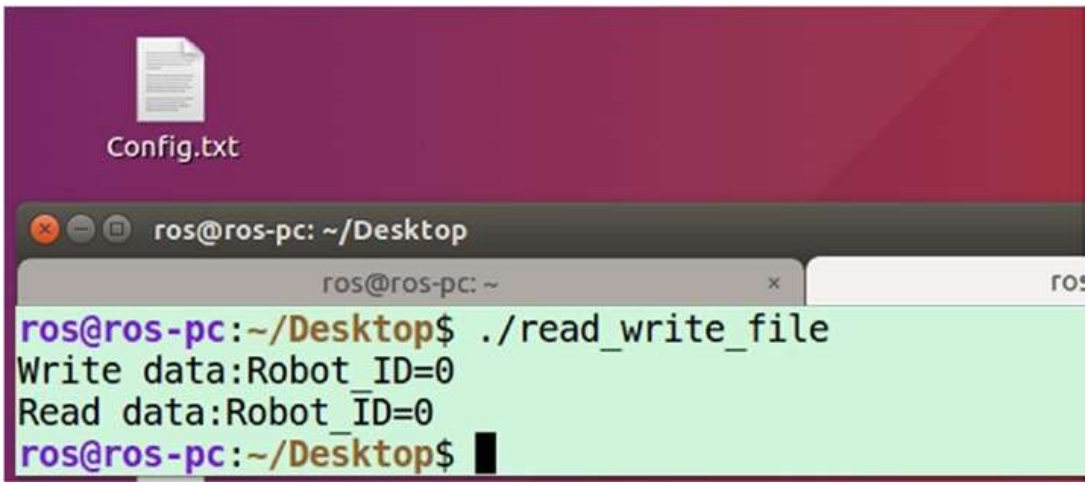

Figure 2-24. *File read/write program*

You can see that `Config.txt` has been created in the `Desktop` folder.

For further information, visit www.tutorialspoint.com/cplusplus/cpp_files_streams.htm.

Namespaces in C++

The namespace concept was mentioned earlier with the `Hello World` code. In this section, you learn how to create, where to use, and how to access a namespace. Listing 2-5 provides an example of creating and using two namespaces.

Listing 2-5. Example Code for C++ Namespaces

```
#include <iostream>

using namespace std;
namespace robot {
	void process(void)
	{
		cout<<"Processing by Robot"<<endl;
	}
}

namespace machine {
	void process(void)
	{
		cout<<"Processing by Machine"<<endl;
	}
}
int main()
{
	robot::process();
	machine::process();
}
```

To create a namespace, use the `namespace` keyword followed by name of the namespace. In Listing 2-5, we are defining two namespaces. If you go through the code, you see that the same function is defined inside each namespace. The namespaces are used to group a set of functions or classes that perform a unique action. We can access the members inside the namespace using the name of the namespace followed by `::` and the function name. In this code, we are calling two functions inside the namespace, called `robot` and `machine`.

Figure 2-25 shows the output of the code in Listing 2-5. The code is saved as `namespace.cpp`.

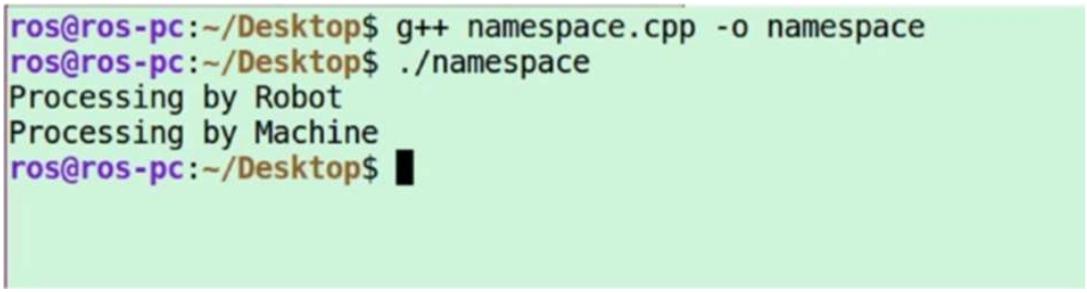

Figure 2-25. *Output of the namespace code*

For additional reference, visit www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm.

C++ Exception Handling

Exception handling in C++ is a new method for handling circumstances in which there is an unexpected output in response to user input. The exception can happen during runtime. Listing 2-6 is an example of the C++ exception handling feature.

Listing 2-6. Example of C++ Exception Handling

```
#include <iostream>
using namespace std;
int main()
{
	try
	{
		int no_1 = 1;
		int no_2 = 0;

		if(no_2 == 0)
		{
			throw no_1;
		}

	}

	catch(int e)

	{
		cout<<"Exception found:"<<e<<endl;
	}

}
```

To handle an exception, we mainly use three keywords.

- `try`: Inside the `try` block, we can write our code, which may raise an exception.
- `catch`: If the `try` block raises an exception, the `catch` block catches the exception. We can decide what to do with that exception.

- `throw`: We can throw an exception from the `try` block when the problem starts to show. If the `throw` statement is executed, it raises an exception and is caught by the `catch` block.

Listing 2-7 shows the general structure.

Listing 2-7. General Structure for Exception Handling

```
try
{
      //Our code snippets
}
catch (Exception name)
{
      //Exception handling block
}
```

The code in Listing 2-6 is checking whether `num_2` is 0. If `num_2` is 0, an exception is raised by using the `throw` keyword with `num_1`, so the `catch` block can receive the `num_1` value for inspecting.

Figure 2-26 shows the output of Listing 2-6.

Figure 2-26. *Output of the exception code*

Inside the catch block, we print the exception value (i.e., the value of `num_1`, which is 1).

Exception handling is widely using for easily debugging a program.

For further reference, visit www.geeksforgeeks.org/exception-handling-c/.

C++ Standard Template Libraries

If you want to work with data structures such as list, stacks, arrays, and so forth, it is best to look at the Standard Template Library (STL). STL provides the implementation of various standard algorithms in computer science, such as sorting and searching, and data structures like vectors, lists, and queue. This is an advanced C++concept. It is a good idea to review the information at www.geeksforgeeks.org/the-c-standard-template-library-stl/.

Building a C++ Project

Now that you've learned some important OOP concepts, let's have a look at how to build a C++ project. Just imagine, you have hundreds or thousands of lines of source code, and you need to compile and link it. How do you do that? This section discusses that.

If you are working with more than one source code, it is a good idea to review and use the following tools to compile and build your project.

Creating a Linux Makefile

A Linux makefile is a tool to compile one or more sources in a single command and build the executable. Let's discuss a simple project to demonstrate the makefile capabilities.

We are going to write code for adding two numbers. For the addition, we first create a class. While working with the C++ classes, we write the declaration and definition of the class in the main source code. Another

approach is to declare and define the class in a header and .cpp file, and then include this header in the main code for getting that class. This approach is helpful in modularizing the entire project. So, our project has three files.

- main.cpp: The main code that we are going to build.
- add.h: The header file of the add class. It has a declaration of the class.
- add.cpp: This file has the entire definition of the add class.

It is a good idea to use the class name as the name of the header and .cpp file. Here we create the add class so that the name of the header is add.h and add.cpp.

Listing 2-8, Listing 2-9, and Listing 2-10 provide the code for each file.

Listing 2-8. add.h

```
#include <iostream>
class add
{
public:
       int compute(int no_1,int no_2);

};
```

Listing 2-9. add.cpp

```
#include "add.h"

int add::compute(int a, int b)
{

        return(a+b);

}
```

Listing 2-10. main.cpp

```
#include "add.h"
using namespace std;
int main()
{
        add obj;
        int result = obj.compute(43,34);
        cout<<"The Result:="<<result<<endl;
        return 0;
}
```

In the `main.cpp` (see Listing 2-10), we include the `add.h` header file to access the `add` class. We create an object of the `add` class, pass two numbers to the compute function, and print the result.

We can compile and execute the code in Listing 2-10 using the following command.

```
$ g++ add.cpp main.cpp -o main
$ ./main
```

The g++ command is easy to use for compiling a single source code, but if we want to compile several source codes, the g++ command is inconvenient. A Linux makefile is one way to compile multiple source codes in a single command. Listing 2-10 shows how to write a makefile for compiling the code.

The code in Listing 2-11 needs to be saved as the makefile.

Listing 2-11. A Linux Makefile

```
CC = g++
CFLAGS = -c
SOURCES =  main.cpp add.cpp
OBJECTS = $(SOURCES:.cpp=.o)
EXECUTABLE = main
```

```
all: $(OBJECTS) $(EXECUTABLE)

$(EXECUTABLE) : $(OBJECTS)
            $(CC) $(OBJECTS) -o $@

.cpp.o: *.h
       $(CC) $(CFLAGS) $< -o $@

clean :
       -rm -f $(OBJECTS) $(EXECUTABLE)

.PHONY: all clean
```

After saving the code in Listing 2-11 as a makefile, you have to execute the following command to build it.

```
$ make
```

This builds the source code, as shown in Figure 2-27.

Figure 2-27. *Output of make command*

After building using the make command, you can execute the program by using the following command. The results are shown in Figure 2-28.

```
$. /main
```

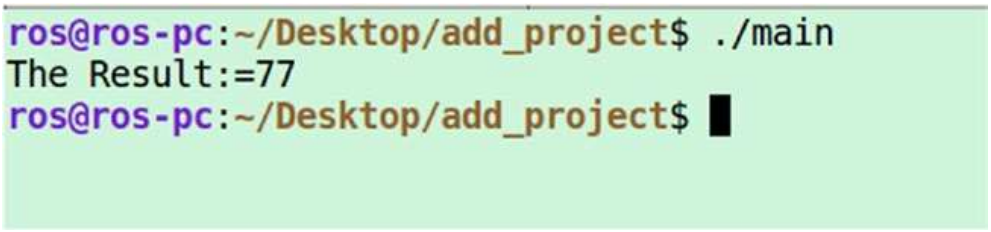

Figure 2-28. *Output of main code*

You can learn more about makefiles at www.bogotobogo.com/cplusplus/gnumake.php.

Creating a CMake File

CMake (cmake.org) is another approach to building a C++ project. CMake stands for *cross-platform makefile*. It is an open source tool to build, test, and package software across multiple OS platforms.

Install CMake by using the following command.

```
$ sudo apt-get install cmake
```

After installing, you can save Listing 2-12 as `CMakeLists.txt`.

Listing 2-12. The CMakeLists.txt File

```
cmake_minimum_required(VERSION 3.0)
set(CMAKE_BUILD_TYPE Release)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14")
project(main)
add_executable(
        main
        add.cpp
        main.cpp
)
```

The code is self-explanatory. It basically sets the C++ flags and creates an executable named `main` from the source code: `add.cpp` and `main.cpp`. The list of CMake commands is available at cmake.org/documentation/.

After saving the preceding commands as `CMakeLists.txt`, we have to create a folder for building the project. You can choose any name for the folder. Here, we use `build` for that folder.

```
$ mkdir build
```

After building the folder, switch to the `build` folder and open the terminal from the `build` folder.

Execute the following command from the `build` folder path.

```
$ cmake ..
```

This command parses `CMakeLists.txt` in the project path. The `cmake` command can convert `CMakeLists.txt` to a makefile, and we can build the makefile after that. Basically, it automates the process of making the Linux makefile.

If everything is successful after executing the `cmake ..` command, you should get the message shown in Figure 2-29.

```
ros@ros-pc:~/Desktop/add_project/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/ros/Desktop/add_project/build
```

Figure 2-29. *Output of cmake command*

After this, you can make the project by entering the make command (`$ make`).

If successful, you can execute the project executable (`$. /main`). Figure 2-30 shows the output of the `make` command and executable.

```
ros@ros-pc:~/Desktop/add_project/build$ make
Scanning dependencies of target main
[ 33%] Building CXX object CMakeFiles/main.dir/add.cpp.o
[ 66%] Building CXX object CMakeFiles/main.dir/main.cpp.o
[100%] Linking CXX executable main
[100%] Built target main
ros@ros-pc:~/Desktop/add_project/build$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  main  Makefile
ros@ros-pc:~/Desktop/add_project/build$ ./main
The Result:=77
```

Figure 2-30. *Output of the make command and executable*

Summary

This chapter discussed the fundamentals of the C++ programming language and how to program in the C++ language in Ubuntu Linux. Knowledge of C++ is a prerequisite for working with ROS. The chapter started by discussing the C++ compiler in Ubuntu and how to compile a C++ file using the compiler. After seeing a compilation, we covered Object Oriented Concepts in C++. We discussed the basic difference between C++ classes and structs in C, and important object-oriented programming concepts, such as access modifiers and inheritance. We also saw examples of these concepts. Then we covered file operations, namespaces, exception handling, and the Standard Template Library in C++. The end of the chapter covered how to compile C++ source code using Linux makefiles and `CMakeLists.txt` files.

In the next chapter, we see how to work with Python in Ubuntu Linux.