Modern C for Absolute Beginners

A Friendly Introduction to the C Programming Language

Slobodan Dmitrović

Apress®

Modern C for Absolute Beginners

A Friendly Introduction to the C Programming Language

Slobodan Dmitrović

Apress®

Modern C for Absolute Beginners: A Friendly Introduction to the C Programming Language

Slobodan Dmitrović
Belgrade, Serbia

ISBN-13 (pbk): 978-1-4842-6642-7 ISBN-13 (electronic): 978-1-4842-6643-4
https://doi.org/10.1007/978-1-4842-6643-4

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Kyler Nixon on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484266427. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

For Sanja and Katarina

Table of Contents

About the Author

Slobodan Dmitrović is a software consultant, trainer, and entrepreneur. He is the founder and CEO of "Clear Programming Paradigm," an LLC that provides outsourcing and training services. Slobodan's ability to summarize complex topics and provide insightful training made him a sought-after consultant for automotive, fintech, and other industries. He has a strong interest in C, C++, software architecture, training, and R&D. Slobodan can be reached at `www.cppandfriends.com`.

About the Technical Reviewer

German Gonzalez-Morris is a software architect/engineer working with C/C++, Java, and different application containers, in particular, with WebLogic Server. He has developed different applications including JEE/Spring/Python. His areas also include OOP, Java/JEE, Python, design patterns, algorithms, Spring Core/MVC/Security, and microservices. German has worked in performance messaging, Restful API, and transactional systems. For more, see www.linkedin.com/in/german-gonzalez-morris.

Acknowledgments

I would like to thank my friends and fellow peers who have supported me in writing my second book.

I am forever indebted to Peter Dunne, Glenn Dufke, Bruce McGee, Tim Crouse, Jens Fudge, Rainer Grimm, and Rob Machin for all their work, help, and support.

I am grateful to the outstanding professionals at Apress who have supported me during the entire writing process.

I am thankful to all of the amazing software developers, architects, and entrepreneurs I met and collaborated with throughout the years.

PART I

The C Programming Language

CHAPTER 1

Introduction

Dear reader, congratulations on choosing to learn the C programming language, and thank you for picking up this book. My name is Slobodan Dmitrović, and I will try to introduce you to a wonderful world of C programming to the best of my abilities. This book is divided into four parts. In Part 1, we cover the C language basics. Part 2 explains the C standard library, and Part 3 introduces us to modern C standards. The final part explains the dos and don'ts in modern C. Let us get started!

1.1 What Is C?

C is a programming language, a general-purpose, procedural, compiled programming language. C language was created by Dennis Ritchie in the late 1960s and early 1970s. The C program is a collection of C source code spread across one or more source and header files. Source files by convention have the *.c* extension, and header files have the *.h* extension. Source and header files are plain text files that contain some C code.

1.2 What Is C Used For?

C is often used for so-called *systems programming,* which is operating systems programming, application programming, and embedded systems programming, to name a few. A large portion of Linux and Windows operating systems was programmed using C. C is often used as a replacement for an assembly language. C language constructs efficiently translate to the hardware itself. Whenever we want to get *down to the metal*, we can opt for C.

S. Dmitrović, *Modern C for Absolute Beginners*, https://doi.org/10.1007/978-1-4842-6643-4_1

1.3 C Compilers

To compile and run a C program, we need a C compiler. A compiler compiles a C program and turns the source code into an object file. The linker then links the object files together and produces an executable file or a library, depending on our intention. For the most part, we say we *compile* the program and assume the compilation process results in an executable file that we can run. At the time of writing, some of the more popular C compilers are:

- gcc – as part of the GCC toolchain
- Clang – as part of the LLVM toolchain
- Visual C/C++ compiler – as part of the Visual Studio IDE
- MinGW – a Windows port of the GCC

1.3.1 Installing Compilers

Here we describe how to install C compilers on Linux and Windows and how to compile and run our programs.

1.3.1.1 On Linux

To install a GCC compiler on Linux, open a terminal window and type:

```
sudo apt install build-essential
```

This command installs a GCC toolchain, which we can use to compile, debug, and run our C programs. Using a text editor of our choice, let us create a file with the following code:

```
#include <stdio.h>

int main(void)
{
      printf("Hello World!\n");
}
```

Let us save the file as a *source.c*. To compile this program using GCC, we type:

```
gcc source.c
```

This will produce an executable file with a default name of *a.out*. To run this file, type the following in a console window:

```
./a.out
```

Running this program should output the `Hello World!` string in our console window.

Note For now, let us take the source code inside the *source.c* file for granted. The example is for demonstration purposes. We will get into detailed code explanation and analysis in later chapters.

To install a clang compiler on our Linux system, type:

```
sudo apt install clang
```

This command installs another compiler called *Clang,* which we can also use to compile our programs. To compile our previous program using a clang compiler, we type:

```
clang source.c
```

Same as before, the compiler compiles the source file and produces an executable file with the default name of *a.out*. To run this file, we type:

```
./a.out
```

The compiler choice is a matter of preference. Just substitute `gcc` with `clang` and vice versa. To compile with warnings enabled, type:

```
gcc -Wall source.c
```

Warnings are not errors. They are messages indicating that something in our program might lead to errors. We want to eliminate or minimize the warnings as well.

To produce a custom executable name, add the `-o` flag, followed by the custom executable name, so that our compilation string now looks like:

```
gcc -Wall source.c -o myexe
```

To run the executable file, we now type:

```
./myexe
```

The ISO C standard governs the C programming language. There are different versions of the C standard. We can target a specific C standard by adding the `-std=` flag, followed by a standard name such as `c99`, `c11`, or `c17`. To compile for a `c99` standard, for example, we would write:

```
gcc -std=c99 -Wall source.c
```

To compile for a C11 standard, we use:

```
gcc -std=c11 -Wall source.c
```

If we want to adhere to strict C standard rules, we add the `-pedantic` compilation flag. This flag issues warnings if our code does not comply with the strict C standard rules. Some of the use cases are:

```
gcc -std=c99 -Wall -pedantic source.c
gcc -std=c11 -Wall -pedantic source.c
gcc -std=c17 -Wall -pedantic source.c
gcc -std=c2x -Wall -pedantic source.c
```

To compile and run the program using a single statement, we type:

```
gcc source.c && ./a.out
```

This statement compiles the program and, if the compilation succeeds, executes the *a.out* file.

Let us put it now all together and use the following compilation strings in our future projects. If using gcc:

```
gcc -Wall -std=c11 -pedantic source.c && ./a.out
```

If using Clang:

```
clang -Wall -std=c11 -pedantic source.c && ./a.out
```

1.3.1.2 On Windows

On Windows, we can install *Visual Studio*. Choose the *Create a new project* option, make sure the *C++* option is selected, choose *Empty Project*, and click *Next*. Modify the project and solution names or leave the default values, and click *Create*. We have now created an empty Visual Studio project. In the *Solution Explorer* window, right-click on a project name and choose *Add – New Item…*. Ensure the *Visual C++* tab is selected, click on the *C++ File (.cpp)* option, modify the file name to *source.c*, and click *Add*. We can use a different file name, but the extension should be *.c*. Double-click on the *source.c* file, and paste our previous *Hello World* source code into it. Press *F5* to run the program. To compile for the C11 standard, use the `/std:c11` compiler switch. To compile for the C17 standard, use the `/std:c17` compiler switch.

Alternatively, install the MinGW (Minimalist GNU for Windows) and use the compiler in a console window, the same way we would on Linux.

So far, we have learned how to set up the programming environments on Linux and Windows and compile and run our C programs. We are now ready to start with the C theory and examples.

1.4 C Standards

The C programming language is a standardized language. There were different C standards throughout history. The first notable standard was the ANSI C, and now it is the ISO standard known as the ISO/IEC:9989 standard. Some of the C standards throughout the years:

- **ANSI C** Standard (referred to as ANSI C and C89)
- **C90** (official name: ISO/IEC 9899:1990; it is the ANSI C Standard adopted by ISO; the C89 and C90 are the same things)
- **C99** (ISO/IEC 9899:1999)
- **C11** (ISO/IEC 9899:2011)
- **C17** (ISO/IEC 9899:2018)
- The upcoming standard informally named **C2x**

Each of the standards introduces new features and changes to the language and the standard library. Everything starting with C11 is often referred to as the *modern C*. And modern C is what we will be teaching in this book. Let us get started!

CHAPTER 2

Our First Program

This section describes the main program entry point, how to work with comments, and how to write a simple "Hello World" program.

2.1 Function main()

Every C program that produces an executable file must have a starting point. This starting point is the function `main`. The function `main` is the function that gets called when we start our executable file. It is the program's main entry point. The signature of the function `main` is:

```
int main(void) {}
```

The function `main` is of type `int`, which stands for integer, followed by the reserved name `main`, followed by an empty list of parameters inside the parentheses `(void)`. The name `void` inside the parentheses means the function accepts no parameters. Following is the function body marked with braces `{}`. The opening brace `{` marks the beginning of a code block, and the closing brace `}` marks the end of the code block. We write our C code inside the code block marked by these braces. The code we write there executes when we start our executable file.

For readability reasons, we can put braces on new lines:

```
int main(void)
{

}
```


S. Dmitrović, *Modern C for Absolute Beginners*, https://doi.org/10.1007/978-1-4842-6643-4_2

We can keep the opening brace on the same line with the `main` function definition and have the ending brace on a new line:

```
int main(void) {

}
```

Note Braces placement position is a matter of conventions, preferences, and coding styles.

In early C standards, the function `main` was required to have a `return 0;` statement. This statement ends the program and returns control to the operating system. The return value of `0` means the program finished the execution as expected. It ended normally. If the `main` function returns any value other than `0`, it means the program ended unexpectedly. So, in previous standards, our blank program would look like:

```
int main(void)
{
      return 0;
}
```

Statements in C end with a semicolon (`;`). The `return 0;` statement within the `main` function *is no longer required* in modern C. We can omit that statement. When the program execution reaches the closing brace, the effect is the same as if we explicitly wrote the statement. In modern standards, we can simply write:

```
int main(void)
{

}
```

We often see the use of the following, also valid `main` signature:

```
int main()
{
      return 0;
}
```

While this signature indicates there are no parameters, in ANSI C, it could potentially allow us to call the function with any number of parameters. Since we want to avoid that, we will be using the `int main(void)` signature, which explicitly states the function does not accept parameters.

With that in mind, we will be using the following `main` skeleton to write our code throughout the book:

```
int main(void)
{

}
```

Note There is another `main` signature accepting two parameters: `int main(int argc, char* argv[])`. We will describe it later in the book when we learn about arrays, pointers, and character arrays.

2.2 Comments

We can have comments in our C program. A comment is a text that is useful to us but is ignored by the compiler. Comments are used to document the source code, serve as notes, or comment-out the part of the source code.

A C-style comment starts with `/*` characters and ends with `*/` characters. The comment text is placed between these characters. Example:

```
int main(void)
{
      /* This is a comment in C */
}
```

The comment can also be a multiline comment:

```
int main(void)
{
      /* This is a
      multi-line comment in C */
}
```

Starting with C99, we can write a single-line comment that starts with a double slash `//` followed by a comment text:

```
int main(void)
{
      // This is a comment
}
```

We can have multiple single-line comments on separate lines:

```
int main(void)
{
      // This is a comment
      // This is another comment
}
```

Comments starting with the double slash `//` are also referred to as *C++ style comments*.

2.3 Hello World

Let us write a simple program that outputs a "Hello World" message in the console window and explain what each line of code does. The full listing is:

```
#include <stdio.h>

int main(void)
{
      printf("Hello World!");
}
```

The first line `#include <stdio.h>` uses the `#include` preprocessor macro to include the content of the `<stdio.h>` header file into our `source.c` file. The standard-library header file name `stdio.h` is surrounded with matching `<>` parentheses. This standard-library header is needed to use the `printf()` function. We call this function inside the `main` function body using the following blueprint: `printf("Message we want to output");`

The `printf` function accepts an argument inside the parentheses (). In our case, this argument is a *string constant* or a *character string* `"Hello World!"`. The string text is surrounded by double quotes (""). The entire `printf("Hello World!")` function call then ends with the semicolon (`;`), and then we call it a *statement*. Statements end with a semicolon in C. Macros such as the `#include <stdio.h>` do not end with a semicolon.

We can output text on multiple lines. To do that, we need to output a newline character, which is `\n`. Example:

```
#include <stdio.h>

int main(void)
{
      printf("Hello World!\nThis is a new line!");
}
```

Output:

```
Hello World!
```

We can split the text into two `printf` function calls for readability reasons. Remember, each time we want the text to start on a new line, we need to output the newline character `\n`:

```
#include <stdio.h>

int main(void)
{
      printf("Hello World!\n");
      printf("This is a new line!");
}
```

Output:

```
Hello World!
This is a new line!
```

This has the same effect as if we placed a newline character at the beginning of the second `printf` function call:

```
#include <stdio.h>

int main(void)
{
      printf("Hello World!");
      printf("\nThis is a new line!");
}
```

Output:

```
Hello World!
This is a new line!
```

CHAPTER 3

Types and Declarations

In this section, we will learn about the built-in types in C and variable declarations.

3.1 Declarations

A *declaration* declares a (variable) name. When we declare a variable, we specify its type and variable's name. When we declare a variable, the compiler reserves memory for our variable. This occupied space is called an *object* or *data object* in memory. These data objects are accessed by names we call *variables*. We need to declare a variable before we can use it. To declare a variable, we put the `type_name` before the `variable_name` and end the entire statement with a semicolon (`;`). The declaration pseudo-code looks like:

```
type_name variable_name;
```

We can declare multiple variables of the same type by separating them with a comma:

```
type_name variable_name1, variable_name2, variable_name3;
```

Variable names can contain both letters and numbers but must not start with a number. C is a case-sensitive language, so `myvar` and `MyVar` are two different, independent names. Variable names should not start with underscore characters as in `_myvar` or `__myvar`.

3.2 Introduction

What is a *type*? A type is a range of values and allowed operations on those values. An instance of a type is called an *object* or a *data object*. When we declare a variable, we are creating an instance.

S. Dmitrović, *Modern C for Absolute Beginners*, https://doi.org/10.1007/978-1-4842-6643-4_3