



Beginning C

From Beginner to Pro

—

Sixth Edition

—

German Gonzalez-Morris
Ivor Horton



Apress®

Beginning C

From Beginner to Pro

Sixth Edition



German Gonzalez-Morris
Ivor Horton

Apress®

Beginning C: From Beginner to Pro

German Gonzalez-Morris
Santiago, Chile

Ivor Horton
STRATFORD UPON AVON, UK

ISBN-13 (pbk): 978-1-4842-5975-7
<https://doi.org/10.1007/978-1-4842-5976-4>

ISBN-13 (electronic): 978-1-4842-5976-4

Copyright © 2020 by German Gonzalez-Morris and Ivor Horton

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

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 designed by Freepik (www.freepik.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 editorial@apress.com; for reprint, paperback, or audio rights, please email 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/9781484259757. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

To my parents, Germán and Felicia

—German Gonzalez-Morris

For my daughter, Dany

—Ivor Horton

Table of Contents

About the Authors	xix
About the Technical Reviewer	xxi
Acknowledgments	xxiii
Introduction	xxv
■ Chapter 1: Programming in C	1
The C Language.....	1
The Standard Library.....	2
Learning C	2
Creating C Programs	2
Editing.....	3
Compiling.....	3
Linking.....	4
Executing	4
Creating Your First Program	5
Editing Your First Program.....	7
Dealing with Errors.....	7
Dissecting a Simple Program	8
Comments	9
Preprocessing Directives	10
Defining the main() Function	10
Keywords	11
The Body of a Function.....	11

Outputting Information	13
Function Arguments	13
Control Characters	14
Trigraph Sequences	16
The Preprocessor	16
Developing Programs in C	17
Understanding the Problem	17
Detailed Design	18
Implementation	18
Testing	18
Functions and Modular Programming	19
Common Mistakes	23
Points to Remember	23
Summary	24
■ Chapter 2: First Steps in Programming	25
Memory in Your Computer	25
What Is a Variable?	28
Naming Variables	28
Variables That Store Integers	29
Using Variables	34
Initializing Variables	35
Basic Arithmetic Operations	36
More on Division with Integers	41
Unary Operators	41
Unary Minus Operator	41
Variables and Memory	42
Signed Integer Types	43
Unsigned Integer Types	43
Specifying Integer Constants	44

Working with Floating-Point Numbers	46
Floating-Point Number Representation	47
Floating-Point Variables.....	48
Division Using Floating-Point Values	49
Controlling the Number of Decimal Places in the Output	50
Controlling the Output Field Width.....	50
More Complicated Expressions	51
Defining Named Constants.....	55
Knowing Your Limitations.....	57
Introducing the sizeof Operator.....	60
Choosing the Correct Type for the Job	61
Explicit Type Conversion.....	65
Automatic Conversions.....	65
Rules for Implicit Conversions	66
Implicit Conversions in Assignment Statements	67
More Numeric Data Types	68
Character Type.....	68
Character Input and Character Output.....	69
Enumerations.....	73
Choosing Enumerator Values	74
Unnamed Enumeration Types	75
Variables That Store Boolean Values	75
The op= Form of Assignment.....	76
Mathematical Functions.....	77
Designing a Program.....	78
The Problem	78
The Analysis.....	79
The Solution.....	81
Summary.....	85

■ Chapter 3: Making Decisions	87
The Decision-Making Process	87
Arithmetic Comparisons	88
The Basic if Statement	88
Extending the if statement: if-else	92
Using Blocks of Code in if Statements.....	95
Nested if Statements	95
Testing Characters.....	99
Logical Operators	102
The Conditional Operator	105
Operator Precedence: Who Goes First?	108
Multiple-Choice Questions	112
Using else-if Statements for Multiple Choices	113
The switch Statement.....	114
The goto Statement	122
Bitwise Operators	123
The op= Use of Bitwise Operators.....	126
Using Bitwise Operators	126
Designing a Program	131
The Problem	131
The Analysis.....	131
The Solution.....	132
Summary	135
■ Chapter 4: Loops	137
How Loops Work	137
Introducing the Increment and Decrement Operators	138
The for Loop	139
General Form of the for Loop	143
More on the Increment and Decrement Operators	144
The Increment Operator.....	144

The Prefix and Postfix Forms of the Increment Operator	144
The Decrement Operator	145
The for Loop Revisited.....	146
Modifying the for Loop Control Variable	148
A for Loop with No Parameters.....	149
The break Statement in a Loop.....	149
Limiting Input Using a for Loop.....	152
Generating Pseudo-random Integers.....	154
More for Loop Control Options.....	157
Floating-Point Loop Control Variables.....	157
Chars loop Control Variables.....	158
The while Loop	158
Nested Loops.....	161
Nested Loops and the goto Statement.....	167
The do-while Loop.....	168
The continue Statement	171
Designing a Program.....	171
The Problem	171
The Analysis.....	172
The Solution.....	173
Summary.....	184
■ Chapter 5: Arrays	187
An Introduction to Arrays.....	187
Programming Without Arrays.....	187
What Is an Array?	189
Using an Array	190
The Address of Operator.....	193
Arrays and Addresses.....	196
Initializing an Array.....	197
Finding the Size of an Array	198

Multidimensional Arrays.....	199
Initializing Multidimensional Arrays	201
Constant Arrays	207
Variable-Length Arrays.....	209
Designing a Program.....	212
The Problem	212
The Analysis.....	212
The Solution.....	213
Summary.....	219
■ Chapter 6: Applications with Strings and Text	221
What Is a String?	221
Variables That Store Strings.....	223
Arrays of Strings.....	226
Operations with Strings.....	228
Checking for C11/C17 Support	228
Finding the Length of a String	230
Copying Strings	231
Concatenating Strings	231
Comparing Strings.....	235
Searching a String.....	239
Tokenizing a String	244
Reading Newline Characters into a String.....	249
Analyzing and Transforming Strings.....	250
Converting Character Case	252
Converting Strings to Numerical Values	254
Designing a Program.....	257
The Problem	257
The Analysis.....	257
The Solution.....	258
Summary.....	263

■ **Chapter 7: Pointers** **265**

A First Look at Pointers **265**

 Declaring Pointers 266

 Accessing a Value Through a Pointer 267

 Using Pointers 271

 Testing for a NULL Pointer 275

 Pointers to Constants 275

 Constant Pointers 276

 Naming Pointers 276

Arrays and Pointers **277**

Multidimensional Arrays..... **280**

 Multidimensional Arrays and Pointers 284

 Accessing Array Elements 286

Using Memory As You Go..... **289**

 Dynamic Memory Allocation: The malloc() Function 289

 Releasing Dynamically Allocated Memory 290

 Memory Allocation with the calloc() Function 295

 Extending Dynamically Allocated Memory 296

Handling Strings Using Pointers..... **300**

 Using Arrays of Pointers 300

 Pointers and Array Notation 307

Designing a Program..... **311**

 The Problem 312

 The Analysis 312

 The Solution 313

 The Complete Program 318

Summary..... **320**

■ Chapter 8: Structuring Your Programs	323
Program Structure.....	323
Variable Scope and Lifetime.....	324
Variable Scope and Functions	327
Functions.....	327
Defining a Function.....	328
The return Statement	332
The Pass-by-Value Mechanism	336
Function Prototypes	338
Pointers As Parameters and Return Types	339
const Parameters	340
Perils of Returning Pointers.....	346
Summary.....	349
■ Chapter 9: More on Functions	351
Pointers to Functions	351
Declaring a Pointer to a Function	351
Calling a Function Through a Function Pointer.....	352
Arrays of Pointers to Functions	355
Pointers to Functions As Arguments.....	357
Variables in Functions	360
Static Variables: Keeping Track Within a Function.....	360
Sharing Variables Between Functions	362
Functions That Call Themselves: Recursion	365
Functions with a Variable Number of Arguments	368
Copying a va_list	371
Basic Rules for Variable-Length Argument Lists	372
The main() Function	372
Terminating a Program.....	374
The abort() Function	374
The exit() and atexit() Functions	374

The <code>_Exit()</code> Function.....	375
The <code>quick_exit()</code> and <code>at_quick_exit()</code> Functions.....	375
Enhancing Performance.....	375
Declaring Functions Inline.....	376
Using the <code>restrict</code> Keyword.....	376
The <code>_Noreturn</code> Function Specifier.....	377
Designing a Program.....	377
The Problem.....	377
The Analysis.....	378
The Solution.....	380
Summary.....	394
■ Chapter 10: Essential Input and Output.....	397
Input and Output Streams.....	397
Standard Streams.....	398
Input from the Keyboard.....	399
Formatted Keyboard Input.....	399
Input Format Control Strings.....	400
Characters in the Input Format String.....	406
Variations on Floating-Point Input.....	408
Reading Hexadecimal and Octal Values.....	409
Reading Characters Using <code>scanf_s()</code>	411
String Input from the Keyboard.....	412
Single-Character Keyboard Input.....	414
Output to the Screen.....	419
Formatted Output Using <code>printf_s()</code>	419
Escape Sequences.....	422
Integer Output.....	422
Outputting Floating-Point Values.....	425
Character Output.....	426

Other Output Functions	428
Unformatted Output to the Screen.....	428
Formatted Output to an Array	429
Formatted Input from an Array	430
Summary.....	430
■ Chapter 11: Structuring Data.....	433
Data Structures: Using struct	433
Defining Structure Types and Structure Variables	435
Accessing Structure Members	436
Unnamed Structures.....	439
Arrays of Structures.....	439
Structure Members in Expressions.....	442
Pointers to Structures.....	442
Dynamic Memory Allocation for Structures.....	443
More on Structure Members	446
Structures As Members of a Structure	446
Declaring a Structure Within a Structure.....	448
Pointers to Structures As Structure Members	449
Doubly Linked Lists	454
Bit Fields in a Structure	457
Structures and Functions	459
Structures As Arguments to Functions	459
Pointers to Structures As Function Arguments.....	459
Structure As a Function Return Value	461
Binary Trees.....	466
Sharing Memory.....	475
Designing a Program.....	479
The Problem	479
The Analysis.....	480
The Solution.....	480
Summary.....	494

■ Chapter 12: Working with Files	495
The Concept of a File.....	495
Positions in a File	496
File Streams.....	496
Accessing Files	497
Opening a File.....	497
Buffering File Operations.....	500
Renaming a File.....	501
Closing a File	501
Deleting a File.....	502
Writing a Text File	502
Reading a Text File	504
Reading and Writing Strings to a Text File.....	507
Formatted File Input and Output	511
Formatted Output to a File.....	511
Formatted Input from a File.....	512
Dealing with Errors.....	515
More Open Modes for Text Files	516
The freopen_s() Function.....	517
Binary File Input and Output.....	518
Opening a File in Binary Mode.....	518
Writing a Binary File	519
Reading a Binary File.....	520
Moving Around in a File.....	526
File Positioning Operations	526
Finding Out Where You Are	527
Setting a Position in a File	528
Using Temporary Work Files	535
Creating a Temporary Work File.....	535
Creating a Unique File Name	535

Updating Binary Files	537
Changing the Contents of a File.....	542
Creating a Record from Keyboard Input	544
Writing a Record to a File	544
Reading a Record from a File	545
Writing a File	546
Listing the File Contents.....	546
Updating the Existing File Contents.....	547
File Open Modes Summary	554
Designing a Program.....	555
The Problem	555
The Analysis.....	556
The Solution.....	556
Summary.....	561
■ Chapter 13: The Preprocessor and Debugging	563
Preprocessing	563
Including Header Files	563
Defining Your Own Header Files	564
Managing Multiple Source Files	565
External Variables.....	565
Static Functions.....	566
Substitutions in Your Program Source Code.....	566
Macros	567
Macros That Look Like Functions	568
Strings As Macro Arguments	569
Joining Two Arguments in a Macro Expansion	571
Preprocessor Directives on Multiple Lines	571
Logical Preprocessor Directives	571
Conditional Compilation.....	572
Testing for Multiple Conditions	573
Undefining Identifiers	573

Testing for Specific Values for Identifiers	573
Multiple-Choice Selections	574
Standard Preprocessing Macros	575
_Generic Macro	576
Debugging Methods	577
Integrated Debuggers	577
The Preprocessor in Debugging	578
Assertions	582
Date and Time Functions	585
Getting Time Values	585
Getting the Date	589
Getting the Day for a Date	593
Summary	596
■ Chapter 14: Advanced and Specialized Topics	597
Working with International Character Sets	597
Understanding Unicode	597
Setting the Locale	598
The Wide Character Type <code>wchar_t</code>	599
Operations on Wide Character Strings	602
File Stream Operations with Wide Characters	606
Fixed Size Types That Store Unicode Characters	607
Specialized Integer Types for Portability	610
Fixed-Width Integer Types	611
Minimum-Width Integer Types	611
Maximum-Width Integer Types	612
The Complex Number Types	612
Complex Number Basics	612
Complex Types and Operations	613
Programming with Threads	617
Creating a Thread	617
Exiting a Thread	619

■ TABLE OF CONTENTS

Joining One Thread to Another	619
Suspending a Thread	622
Managing Thread Access to Data	623
Summary	630
■ Appendix A: Computer Arithmetic	631
Binary Numbers	631
Hexadecimal Numbers	632
Negative Binary Numbers	634
Big-Endian and Little-Endian Systems	635
Floating-Point Numbers	637
■ Appendix B: ASCII Character Code Definitions	641
■ Appendix C: Reserved Words in C	647
■ Appendix D: Input and Output Format Specifications	649
Output Format Specifications	649
Input Format Specifications	652
■ Appendix E: Standard Library Header Files	655
■ Index	659

About the Authors

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.

Ivor Horton is self-employed in consultancy and writes programming tutorials. He worked for IBM for many years and holds a bachelor's degree, with honors, in mathematics. Ivor's experience at IBM includes programming in most languages (like assembler and high-level languages) on a variety of machines, real-time programming, and designing and implementing real-time closed-loop industrial control systems. He has extensive experience teaching programming to engineers and scientists (Fortran, PL/1, APL, etc.). Ivor is an expert in mechanical, process, and electronic CAD systems; mechanical CAM systems; and DNC/CNC systems.

About the Technical Reviewer

Michael Thomas has worked in software development for more than 20 years as an individual contributor, team lead, program manager, and vice president of engineering. Michael has more than 10 years of experience working with mobile devices. His current focus is in the medical sector, using mobile devices to accelerate information transfer between patients and healthcare providers.

Acknowledgments

I want to thank my family—my parents, Germán and Felicia Morris, for giving me education opportunities and support; Patricia Cruces, my partner, for her infinite patience and love; and my sons, Raimundo and Gregorio, for their happiness and inspiration.

I value the support and opportunity given to me by the complete Apress team, Steve Anglin and Mark Powers, and thank them for their guidance and advice. I also thank Michael Thomas, the technical reviewer, for his important feedback, suggestions, and corrections.

Thanks to my friends and colleagues for their understanding, perceptions, and recommendations on completing ideas in the book: Ariel Aguayo, Carlos Hasan, and Daniel Lagos.

Introduction

Welcome to *Beginning C: From Beginner to Pro*, Sixth Edition. With this book, you can become a competent C programmer using the latest version of the C language. In many ways, C is an ideal language with which to learn programming. It's very compact, so there isn't a lot of syntax to learn before you can write real applications. In spite of its conciseness, it's extremely powerful and is used by professionals in many different areas. The power of C is such that it can be applied at all levels, from developing device drivers and operating system components to creating large-scale applications. A relatively new area for C is in application development for mobile phones.

C compilers are available for virtually every kind of computer, so when you've learned C, you'll be equipped to program in just about any context. Once you know C, you have an excellent base from which you can build an understanding of the object-oriented C++.

My objective in this book is to minimize what I think are the three main hurdles the aspiring programmer must face: coming to grips with the jargon that pervades every programming language, understanding how to use the language elements (as opposed to merely knowing what they are), and appreciating how the language is applied in a practical context.

Jargon is an invaluable and virtually indispensable means of communication for the expert professional as well as the competent amateur, so it can't be avoided. My approach is to ensure that you understand the jargon and get comfortable using it in context. In this way, you'll be able to more effectively use the documentation that comes along with the typical programming product and also feel comfortable reading and learning from the literature that surrounds most programming languages.

Comprehending the syntax and effects of the language elements is obviously an essential part of learning C, but appreciating how the language features work and how they are used is equally important. Rather than just using code fragments, I provide you with practical working examples in each chapter that show how the language features can be applied to specific problems. These examples provide a basis for you to experiment and see the effects of changing the code.

Your understanding of programming in context needs to go beyond the mechanics of applying individual language elements. To help you gain this understanding, I conclude most chapters with a more complex program that applies what you've learned in the chapter. These programs will help you gain the competence and confidence to develop your own applications and provide you with insight into how you can apply language elements in combination and on a larger scale. Most important, they'll give you an idea of what's involved in designing real programs and managing real code.

It's important to realize a few things that are true for learning any programming language. First, there is quite a lot to learn, but this means you'll gain a greater sense of satisfaction when you've mastered it. Second, it's great fun, so you really will enjoy it. Third, you can only learn programming by doing it, and this book helps you along the way. Finally, it's certain you will make a lot of mistakes and get frustrated from time to time during the learning process. When you think you are completely stuck, you just need to be persistent. You will eventually experience that eureka moment and realize it wasn't as difficult as you thought.

How to Use This Book

Because I believe in the hands-on approach, you'll write your first programs almost immediately. Every chapter has several complete programs that put theory into practice, and these are key to the book. You should type in and run all the examples that appear in the text because the very act of typing them in is a tremendous memory aid. You should also attempt all the exercises that appear at the end of each chapter. When you get a program to work for the first time—particularly when you're trying to solve your own problems—you'll find that the great sense of accomplishment and progress makes it all worthwhile.

The pace is gentle at the start, but you'll gain momentum as you get further into the subject. Each chapter covers quite a lot of ground, so take your time and make sure you understand everything before moving on. Experimenting with the code and trying out your own ideas are important parts of the learning process. Try modifying the programs and see what else you can make them do—that's when it gets really interesting. And don't be afraid to try things out—if you don't understand how something works, just type in a few variations and see what happens. It doesn't matter if it's wrong. You'll find you often learn a lot from getting it wrong. A good approach is to read each chapter through, get an idea of its scope, and then go back and work through all the examples.

You might find some of the end-of-chapter programs quite difficult. Don't worry if it's not all completely clear on the first try. There are bound to be bits that you find hard to understand at first because they often apply what you've learned to rather complicated problems. If you really get stuck, you can skip the end-of-chapter exercises, move on to the next chapter, and come back to them later. You can even go through the entire book without worrying about them. However, if you can complete the exercises, it shows you are making real progress.

Who This Book Is For

Beginning C, Sixth Edition is designed to teach you how to write useful programs in C as quickly and easily as possible. By the end of *Beginning C*, you'll have a thorough grounding in programming the C language. This is a tutorial for those of you who've done a little bit of programming before, understand the concepts behind it, and want to further your knowledge by learning C. However, no previous programming knowledge on your part is assumed, so if you're a newcomer to programming, the book will still work for you.

What You Need to Use This Book

To use this book, you'll need a computer with a C compiler and library installed, so you can execute the examples, and a program text editor for preparing your source code files. The compiler you use should provide good support for the current international standard for the C language, C17 (ISO/IEC 9899:2018), which is a bug fix version for C11, commonly referred to as C17 or C18. You'll also need an editor for creating and modifying your code. You can use any plain text editor such as Notepad or vi to create your source program files. However, you'll get along better if your editor is designed for editing C code.

I can suggest two sources for a suitable C compiler, both of which are freeware:

- The GNU C compiler, GCC, is available from www.gnu.org and supports a variety of operating system environments.
- The Pelles C compiler for Microsoft Windows is downloadable from www.smorgasbordet.com/pellec/ and includes an excellent integrated development environment (IDE).

Conventions Used

I use a number of different styles of text and layout in the book to help differentiate between the different kinds of information. For the most part, their meanings will be obvious. Program code will appear like this:

```
int main(void)
{   printf("Beginning C\n");
    return 0;
}
```

When a code fragment is a modified version of a previous instance, I occasionally show the lines that have changed in bold type like this:

```
int main(void)
{
    printf("Beginning C by Ivor Horton\n");
    return 0;
}
```

When code appears in the text, it has a different typestyle that looks like this: `double`.

I'll use different types of "brackets" in the program code. They aren't interchangeable, and their differences are very important. I'll refer to the symbols `()` as parentheses, the symbols `{}` as braces, and the symbols `[]` as square brackets.

Important new words in the text are shown in italic *like this*.

CHAPTER 1



Programming in C

C is a powerful and compact computer language that allows you to write programs that specify exactly what you want your computer to do. You're in charge: you create a program, which is just a set of instructions, and your computer will follow them.

Programming in C isn't difficult, as you're about to find out. I'm going to teach you all the fundamentals of C programming in an enjoyable and easy-to-understand way, and by the end of this chapter, you'll have written your first few C programs. It's as easy as that!

In this chapter, you'll learn

- What the C language standard is
- What the standard library is
- How to create C programs
- How C programs are organized
- How to write your own program to display text on the screen

The C Language

C is remarkably flexible. It has been used for developing just about everything you can imagine by way of a computer program, from accounting applications to word processing and from games to operating systems. It is not only the basis for more advanced languages, such as C++, it is also used currently for developing mobile phone apps in the form of Objective C. *Objective C* is standard C with a thin veneer of object-oriented programming capability added and too many new devices/microcontrollers, such as Raspberry Pi and Arduino. C is easy to learn because of its compactness. Thus, C is an ideal first language if you have ambitions to be a programmer. You'll acquire sufficient knowledge for practical application development quickly and easily.

The C language is defined by an international standard, and the latest is currently defined by the C17 (ISO/IEC 9899:2018), which is a bug fix version for C11 more than new features (for instance, it deprecates `ATOMIC_VAR_INIT`). The current standard is commonly referred to as C17 or C18—the informal names of this version. This occurs because it was finished in 2017, but published in 2018. It is known that GCC uses C17 as a parameter to target this new version. Nevertheless, the aforementioned is not declared in the standard, and the language that I describe in this book conforms to C17 or can be considered C11 with several solved issues. You need to be aware that some elements of the language as defined by C17 are optional. This implies that a C compiler that conforms to the C17 standard may not implement everything in the standard. (A *compiler* is just a program that converts your program written in terms you understand into

a form your computer understands.) I will identify any language feature in the book that is optional so far as C17 is concerned, just so you are aware that it is possible that your compiler may not support it. We will use C11/C17 as a synonym in the book.

It is also possible that a C17 compiler may not implement all of the language features mandated by the C17 standard; in particular, only the newest compilers have C11/C17 compatibility at 100 percent. It takes time to implement new language capabilities, so compiler developers will often take an incremental approach to implementing them. This provides another reason why a program may not work. Having said that, I can confirm from my own experience that the most common reason for things not working in a C program, at least 99.9 percent of the time, is that a mistake has been made.

The Standard Library

The *standard library* for C is also specified within the C17 standard. The standard library defines constants, symbols, and functions that you frequently need when writing a C program. It also provides some optional extensions to the basic C language. Machine-dependent facilities such as input and output for your computer are implemented by the standard library in a machine-independent form. This means that you write data to a disk file in C in the same way on your PC as you would on any other kind of computer, even though the underlying hardware processes are quite different. The standard functionality that the library contains includes capabilities that most programmers are likely to need, such as processing text strings or math calculations. This saves you an enormous amount of effort that would be required to implement such things yourself.

The standard library is specified in a set of standard files called *header files*. Header files always have names with the extension `.h`. To make a particular set of standard features available in your C program file, you just include the appropriate standard header file in a way that I'll explain later in this chapter. Every program you write will make use of the standard library. A summary of the header files that make up the standard library is in Appendix E.

At the beginning, there was the C POSIX library that implemented many features for ANSI C. One of those libraries is `pthread` that today is obsolete and implemented in the standard library. Other POSIX libraries (ISO/IEC 9945 (POSIX)) are in the road map for C2x for future releases.

Learning C

If you are completely new to programming, there are some aspects of C that you do not need to learn, at least not the first time around. These are capabilities that are quite specialized or used relatively infrequently. I have put all these together in Chapter 14 so you will learn about them when you are comfortable with the rest.

Although the code for all the examples is available via the **Download Source Code** link located at www.apress.com/9781484259757, I recommend that you type in all the examples in the book, even when they are very simple. Keying stuff in makes it less likely that you will forget things later. Don't be afraid to experiment with the code. Making mistakes is very educational in programming. The more mistakes you make early on, the more you are likely to learn.

Creating C Programs

There are four fundamental stages, or processes, in the creation of any C program:

- Editing
- Compiling
- Linking
- Executing

You'll soon know all these processes like the back of your hand because you'll be carrying them out so often. First, I'll explain what each process is and how it contributes to the development of your C program.

Editing

Editing is the process of creating and modifying C source code—the name given to the program instructions you write. Some C compilers come with a specific editor program that provides a lot of assistance in managing your programs. In fact, an editor often provides a complete environment for writing, managing, developing, and testing your programs. This is sometimes called an *integrated development environment* (IDE).

You can also use a general-purpose text editor to create your source files, but the editor must store the code as plain text without any extra formatting data embedded in it. Don't use a word processor such as Microsoft Word; word processors aren't suitable for producing program code because of the extra formatting information they store along with the text. In general, if you have a compiler system with an editor included, it will provide a lot of features that make it easier to write and organize your source programs. There will usually be automatic facilities for laying out the program text appropriately and color highlighting for important language elements, which not only makes your code more readable but also provides a clear indicator when you make errors when keying in such words.

If you're working with Linux, the most common text editor is the Vim editor. Alternately, you might prefer to use the GNU Emacs editor. With Microsoft Windows, you could use one of the many freeware and shareware programming editors. These will often provide help in ensuring your code is correct, with syntax highlighting and autoindenting. There is also a version of Emacs for Microsoft Windows. The vi and Vim editors from the UNIX environment are available for Windows too, and you could even use Notepad++ (<http://notepad-plus-plus.org/>).

Of course, you can also purchase one of the professionally created programming development environments that support C, such as those from JetBrains or Microsoft (there is a free Community Edition), in which case you will have very extensive editing capabilities. Before parting with your cash though, it's a good idea to check that the level of C that is supported conforms to the current C standard, C17. With some of the products out there that are primarily aimed at C++ developers, C has been left behind somewhat.

Compiling

The *compiler* converts your source code into machine language and detects and reports errors in the compilation process. The input to this stage is the file you produce during your editing, which is usually referred to as a *source file*.

The compiler can detect a wide range of errors that are due to invalid or unrecognized program code, as well as structural errors where, for example, part of a program can never be executed. The output from the compiler is known as *object code*, and it is stored in files called *object files*, which usually have names with the extension `.obj` in the Microsoft Windows environment or `.o` in the Linux/UNIX environment. The compiler can detect several different kinds of errors during the translation process, and most of these will prevent the object file from being created.

The result of a successful compilation is a file with the same name as that used for the source file, but with the `.o` or `.obj` extension.

If you're working in UNIX, at the command line, the standard command to compile your C programs will be `cc` (or the GNU's Not UNIX [GNU] compiler, which is `.gcc`). You can use it like this:

```
cc -c myprog.c
```

where `myprog.c` is the name of the source file that contains the program you want to compile. Note that if you omit the `-c` flag, your program will automatically be linked as well. The result of a successful compilation will be an object file.

Most C compilers will have a standard compile option, whether it's from the command line (such as `cc myprog.c`) or a menu option from within an IDE (where you'll find a Compile menu option). Compiling from within an IDE is generally much easier than using the command line.

Compilation is a two-stage process. The first stage is called the *preprocessing phase*, during which your code may be modified or added to, and the second stage is the actual *compilation* that generates the object code (this second stage does assembly underneath; GCC and other compilers have options for these steps, but most of the time, it is not necessary). Your source file can include preprocessing *macros*, which you use to add to or modify the C program statements. Don't worry if this doesn't make complete sense now. It will come together for you as the book progresses.

Linking

The *linker* combines the object modules generated by the compiler from source code files, adds required code modules from the standard library supplied as part of C, and welds everything into an executable whole. The linker also detects and reports errors, for example, if part of your program is missing or a nonexistent library component is referenced.

In practice, a program of any significant size will consist of several source code files, from which the compiler generates object files that need to be linked. A large program may be difficult to write in one working session, and it may be impossible to work with as a single file. By breaking it up into a number of smaller source files that each provide a coherent part of what the complete program does, you can make the development of the program a lot easier. The source files can be compiled separately, which makes eliminating simple typographical errors a bit easier. Furthermore, the whole program can usually be developed incrementally. The set of source files that make up the program will usually be integrated under a *project name*, which is used to refer to the whole program.

Program libraries support and extend the C language by providing routines to carry out operations that aren't part of the language. For example, libraries contain routines that support operations such as performing input and output, calculating a square root, comparing two character strings, or obtaining date and time information.

A failure during the linking phase means that once again you have to go back and edit your source code. Success, on the other hand, will produce an executable file, but this does not necessarily mean that your program works correctly. In a Microsoft Windows environment, the executable file will have an `.exe` extension; in UNIX, there will be no such extension, but the file will be of an executable type. Many IDEs have a *build option*, which will compile and link your program in a single operation.

Executing

The execution stage is where you run your program, having completed all the previous processes successfully. Unfortunately, this stage can also generate a wide variety of error conditions that can include producing the wrong output, just sitting there and doing nothing, or perhaps crashing your computer for good measure. In all cases, it's back to the editing process to check your source code.

Now for the good news: This is also the stage where if your program works, you get to see your computer doing exactly what you told it to do! In UNIX and Linux, you can just enter the name of the file that has been compiled and linked to execute the program. In most IDEs, you'll find an appropriate menu command that allows you to run or execute your compiled program. This Run or Execute option may have a menu of its own, or you may find it under the Compile menu option. In Windows, you can run the `.exe` file for your program as you would any other executable.

The processes of editing, compiling, linking, and executing are essentially the same for developing programs in any environment and with any compiled language. Figure 1-1 summarizes how you would typically pass through processes as you create your own C programs.

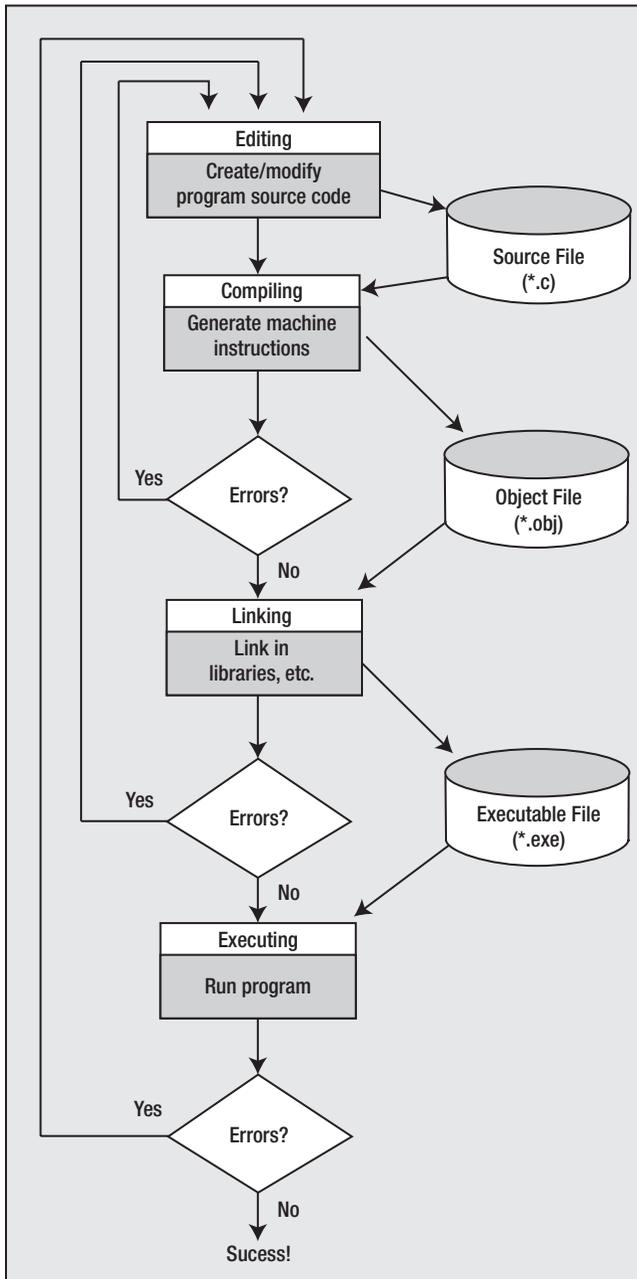


Figure 1-1. Creating and executing a program

Creating Your First Program

We'll step through the processes of creating a simple C program, from entering the program source code to executing it. Don't worry if what you type doesn't mean much to you at this stage—I'll explain everything as we go along.

TRY IT OUT: AN EXAMPLE C PROGRAM

Run your editor and type in the following program exactly as it's written. Be careful to use the punctuation exactly as you see here. Make sure you enter the brackets that are on the fourth and last lines as braces—the curly ones {}, not the square brackets [] or the parentheses ()—it really does matter. Also, make sure you put the forward slashes the right way (/), as later you'll be using the backslash (\) as well. Don't forget the semicolon (;):

```
/* Program 1.1 Your Very First C Program - Displaying Hello World */
#include <stdio.h>

int main(void)
{
    printf("Hello world!");
    return 0;
}
```

When you've entered the source code, save the program as `hello.c`. You can use whatever name you like instead of `hello`, but the extension must be `.c`. This extension is the common convention when you write C programs and identifies the contents of the file as C source code. Most compilers will expect the source file to have the extension `.c`, and if it doesn't, the compiler may refuse to process it.

Next, you'll compile your program as I described in the "Compiling" section previously in this chapter and then link the pieces necessary to create an executable program, as discussed in the "Linking" section. Compiling and linking are often carried out in a single operation, in which case it is usually described as a *build operation*. When the source code has been compiled successfully, the linker will add code from the standard libraries that your program needs and create the single executable file for your program.

Finally, you can execute your program. Remember that you can do this in several ways. There is the usual method of double-clicking the `.exe` file from Windows Explorer if you're using Windows, but you will be better off opening a command-line window and typing in the command to execute it because the window showing the output will disappear when execution is complete. You can run your program from the command line in all operating system environments. Just start a command-line session, change the current directory to the one that contains the executable file for your program, and then enter the program name to run it.

If everything worked without producing any error messages, you've done it! This is your first program, and you should see the following output:

```
Hello world!
```
