Fourth Edition

Fortran

for Scientists and Engineers

Stephen J. Chapman

Mc Graw Hill Education

Fortran for Scientists and Engineers

Fourth Edition

Fortran for Scientists and Engineers

Fourth Edition

Stephen J. Chapman
BAE Systems Australia

Mc Graw Hill Education

FORTRAN FOR SCIENTISTS AND ENGINEERS, FOURTH EDITION

Published by McGraw-Hill Education, 2 Penn Plaza, New York, NY 10121. Copyright © 2018 by McGraw-Hill Education. All rights reserved. Printed in the United States of America. Previous edition © 2008 and 2004. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of McGraw-Hill Education, including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 LCR 21 20 19 18 17

ISBN 978–0–07–338589–1
MHID 0–07–338589–1

Chief Product Officer, SVP Products & Markets: *G. Scott Virkler*
Vice President, General Manager, Products & Markets: *Marty Lange*
Vice President, Content Design & Delivery: *Betsy Whalen*
Managing Director: *Thomas Timp*
Brand Manager: *Raghothaman Srinivasan/ Thomas M. Scaife, Ph.D*
Director, Product Development: *Rose Koos*
Product Developer: *Tina Bower*
Marketing Manager: *Shannon O'Donnell*
Director, Content Design & Delivery: *Linda Avenarius*
Program Manager: *Lora Neyens*
Content Project Managers: *Jane Mohr and Sandra Schnee*
Buyer: *Jennifer Pickel*
Design: *Studio Montage, St. Louis, MO*
Content Licensing Specialist: *DeAnna Dausener*
Cover Image: *hh5800/Getty Images*
Compositor: *Aptara®, Inc.*
Printer: *LSC Communications*

All credits appearing on page or at the end of the book are considered to be an extension of the copyright page.

Library of Congress Cataloging-in-Publication Data

Chapman, Stephen J., author.
Fortran for scientists and engineers / Stephen J. Chapman, BAE Systems Australia.
Fourth edition. | New York, NY : McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., [2017] | Includes index.
LCCN 2016052439 | ISBN 9780073385891 (alk. paper) | ISBN 0073385891 (alk. paper)
LCSH: FORTRAN (Computer program language) | Science—Data processing. | Engineering—Data processing.
LCC QA76.73.F25 C425 2017 | DDC 005.13/3—dc23 LC record available at https://lccn.loc.gov/2016052439

The Internet addresses listed in the text were accurate at the time of publication. The inclusion of a website does not indicate an endorsement by the authors or McGraw-Hill Education, and McGraw-Hill Education does not guarantee the accuracy of the information presented at these sites.

mheducation.com/highered

This book is dedicated to my son Avi, who is the only one of our eight children actually making a living writing software!

ABOUT THE AUTHOR

STEPHEN J. CHAPMAN received a B.S. in Electrical Engineering from Louisiana State University (1975), an M.S.E. in Electrical Engineering from the University of Central Florida (1979), and pursued further graduate studies at Rice University.

From 1975 to 1980, he served as an officer in the U.S. Navy, assigned to teach Electrical Engineering at the U.S. Naval Nuclear Power School in Orlando, Florida. From 1980 to 1982, he was affiliated with the University of Houston, where he ran the power systems program in the College of Technology.

From 1982 to 1988 and from 1991 to 1995, he served as a Member of the Technical Staff of the Massachusetts Institute of Technology's Lincoln Laboratory, both at the main facility in Lexington, Massachusetts, and at the field site on Kwajalein Atoll in the Republic of the Marshall Islands. While there, he did research in radar signal processing systems. He ultimately became the leader of four large operational range instrumentation radars at the Kwajalein field site (TRADEX, ALTAIR, ALCOR, and MMW).

From 1988 to 1991, Mr. Chapman was a research engineer in Shell Development Company in Houston, Texas, where he did seismic signal processing research. He was also affiliated with the University of Houston, where he continued to teach on a part-time basis.

Mr. Chapman is currently Manager of Systems Modeling and Operational Analysis for BAE Systems Australia, in Melbourne, Australia. He is the leader of a team that has developed a model of how naval ships defend themselves against antiship missile attacks. This model contains more than 400,000 lines of MATLAB code written over more than a decade, so he has extensive practical experience applying MATLAB to real-world problems.

Mr. Chapman is a Senior Member of the Institute of Electrical and Electronic Engineers (and several of its component societies). He is also a member of the Association for Computing Machinery and the Institution of Engineers (Australia).

TABLE OF CONTENTS

PREFACE

The first edition of this book was conceived as a result of my experience in writing and maintaining large Fortran programs in both the defense and geophysical fields. During my time in industry, it became obvious that the strategies and techniques required to write large, *maintainable* Fortran programs were quite different from what new engineers were learning in their Fortran programming classes at school. The incredible cost of maintaining and modifying large programs once they are placed into service absolutely demands that they be written to be easily understood and modified by people other than their original programmers. My goal for this book is to teach simultaneously both the fundamentals of the Fortran language and a programming style that results in good, maintainable programs. In addition, it is intended to serve as a reference for graduates working in industry.

It is quite difficult to teach undergraduates the importance of taking extra effort during the early stages of the program design process in order to make their programs more maintainable. Class programming assignments must by their very nature be simple enough for one person to complete in a short period of time, and they do not have to be maintained for years. Because the projects are simple, a student can often "wing it" and still produce working code. A student can take a course, perform all of the programming assignments, pass all of the tests, and still not learn the habits that are really needed when working on large projects in industry.

From the very beginning, this book teaches Fortran in a style suitable for use on large projects. It emphasizes the importance of going through a detailed design process before any code is written, using a top-down design technique to break the program up into logical portions that can be implemented separately. It stresses the use of procedures to implement those individual portions, and the importance of unit testing before the procedures are combined into a finished product. Finally, it emphasizes the importance of exhaustively testing the finished program with many different input data sets before it is released for use.

In addition, this book teaches Fortran as it is actually encountered by engineers and scientists working in industry and in laboratories. One fact of life is common in all programming environments: Large amounts of old legacy code that have to be maintained. The legacy code at a particular site may have been originally written in Fortran IV (or an even earlier version!), and it may use programming constructs that are no longer common today. For example, such code may use arithmetic `IF` statements, or computed or assigned `GO TO` statements. Chapter 18 is devoted to those older features of the language that are no longer commonly used, but that are encountered in legacy code.

The chapter emphasizes that these features should *never* be used in a new program, but also prepares the student to handle them when he or she encounters them.

CHANGES IN THIS EDITION

This edition builds directly on the success of *Fortran 95/2003 for Scientists and Engineers*, 3/e. It preserves the structure of the previous edition, while weaving the new Fortran 2008 material (and limited material from the proposed Fortran 2015 standard) throughout the text. It is amazing, but Fortran started life around 1954, and it is *still* evolving.

Most of the additions in Fortran 2008 are logical extensions of existing capabilities of Fortran 2003, and they are integrated into the text in the proper chapters. However, the use of parallel processing and Coarray Fortran is completely new, and Chapter 17 has been added to cover that material.

The vast majority of Fortran courses are limited to one-quarter or one semester, and the student is expected to pick up both the basics of the Fortran language and the concept of how to program. Such a course would cover Chapters 1 through 7 of this text, plus selected topics in Chapters 8 and 9 if there is time. This provides a good foundation for students to build on in their own time as they use the language in practical projects.

Advanced students and practicing scientists and engineers will need the material on COMPLEX numbers, derived data types, and pointers found in Chapters 11 through 15. Practicing scientists and engineers will almost certainly need the material on obsolete, redundant, and deleted Fortran features found in Chapter 18. These materials are rarely taught in the classroom, but they are included here to make the book a useful reference text when the language is actually used to solve real-world problems.

FEATURES OF THIS BOOK

Many features of this book are designed to emphasize the proper way to write reliable Fortran programs. These features should serve a student well as he or she is first learning Fortran, and should also be useful to the practitioner on the job. They include:

1. *Emphasis on Modern Fortran.*
   The book consistently teaches the best current practice in all of its examples. Many modern Fortran 2008 features duplicate and supersede older features of the Fortran language. In those cases, the proper usage of the modern language is presented. Examples of older usage are largely relegated to Chapter 18, where their old/undesirable nature is emphasized. Examples of modern Fortran features that supersede older features are the use of modules to share data instead of `COMMON` blocks, the use of `DO . . . END DO` loops instead of `DO . . . CONTINUE` loops, the use of internal procedures instead of statement functions, and the use of `CASE` constructs instead of computed `GOTO`s.

2. *Emphasis on Strong Typing.*

   The `IMPLICIT NONE` statement is used consistently throughout the book to force the explicit typing of every variable used in every program, and to catch common typographical errors at compilation time. In conjunction with the explicit declaration of every variable in a program, the book emphasizes the importance of creating a data dictionary that describes the purpose of each variable in a program unit.
3. *Emphasis on Top-Down Design Methodology.*

   The book introduces a top-down design methodology in Chapter 3, and then uses it consistently throughout the rest of the book. This methodology encourages a student to think about the proper design of a program *before* beginning to code. It emphasizes the importance of clearly defining the problem to be solved and the required inputs and outputs before any other work is begun. Once the problem is properly defined, it teaches the student to employ stepwise refinement to break the task down into successively smaller subtasks, and to implement the subtasks as separate subroutines or functions. Finally, it teaches the importance of testing at all stages of the process, both unit testing of the component routines and exhaustive testing of the final product. Several examples are given of programs that work properly for some data sets, and then fail for others.

   The formal design process taught by the book may be summarized as follows:

   - *Clearly state the problem that you are trying to solve.*
   - *Define the inputs required by the program and the outputs to be produced by the program.*
   - *Describe the algorithm that you intend to implement in the program. This step involves top-down design and stepwise decomposition, using pseudocode or flow charts.*
   - *Turn the algorithm into Fortran statements.*
   - *Test the Fortran program. This step includes unit testing of specific subprograms, and also exhaustive testing of the final program with many different data sets.*

4. *Emphasis on Procedures.*

   The book emphasizes the use of subroutines and functions to logically decompose tasks into smaller subtasks. It teaches the advantages of procedures for data hiding. It also emphasizes the importance of unit testing procedures before they are combined into the final program. In addition, the book teaches about the common mistakes made with procedures, and how to avoid them (argument type mismatches, array length mismatches, etc.). It emphasizes the advantages associated with explicit interfaces to procedures, which allow the Fortran compiler to catch most common programming errors at compilation time.
5. *Emphasis on Portability and Standard Fortran.*

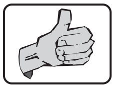
   The book stresses the importance of writing portable Fortran code, so that a program can easily be moved from one type of computer to another one.

It teaches students to use only standard Fortran statements in their programs, so that they will be as portable as possible. In addition, it teaches the use of features such as the `SELECTED_REAL_KIND` function to avoid precision and kind differences when moving from computer to computer.

The book also teaches students to isolate machine-dependent code (such as code that calls machine-dependent system libraries) into a few specific procedures, so that only those procedures will have to be rewritten when a program is ported between computers.

6. *Good Programming Practice Boxes.*

   These boxes highlight good programming practices when they are introduced for the convenience of the student. In addition, the good programming practices introduced in a chapter are summarized at the end of the chapter. An example Good Programming Practice Box is shown below:



Good Programming Practice

Always indent the body of an `IF` structure by two or more spaces to improve the readability of the code.

7. *Programming Pitfalls Boxes*

   These boxes highlight common errors so that they can be avoided. An example Programming Pitfalls Box is shown below:

Programming Pitfalls

Beware of integer arithmetic. Integer division often gives unexpected results.

8. *Emphasis on Pointers and Dynamic Data Structures.*

   Chapter 15 contains a detailed discussion of Fortran pointers, including possible problems resulting from the incorrect use of pointers such as memory leaks and pointers to deallocated memory. Examples of dynamic data structures in the chapter include linked lists and binary trees.

   Chapter 16 contains a discussion of Fortran objects and object-oriented programming, including the use of dynamic pointers to achieve polymorphic behavior.

9. *Use of Sidebars.*

   A number of sidebars are scattered throughout the book. These sidebars provide additional information of potential interest to the student. Some sidebars are historical in nature. For example, one sidebar in Chapter 1 describes the IBM Model 704, the first computer to ever run Fortran. Other sidebars reinforce lessons from the main text. For example, Chapter 9 contains a sidebar reviewing and summarizing the many different types of arrays found in modern Fortran.

10. *Completeness.*

Finally, the book endeavors to be a complete reference to the modern Fortran language, so that a practitioner can locate any required information quickly. Special attention has been paid to the index to make features easy to find. A special effort has also been made to cover such obscure and little understood features as passing procedure names by reference, and defaulting values in list-directed input statements.

PEDAGOGICAL FEATURES

The book includes several features designed to aid student comprehension. Each chapter begins with a list of the objectives that should be achieved in that chapter. A total of 27 quizzes appear scattered throughout the chapters, with answers to all questions included in Appendix F. These quizzes can serve as a useful self-test of comprehension. In addition, there are approximately 360 end-of-chapter exercises. Answers to selected exercises are available at the book's Web site, and of course answers to all exercises are included in the Instructor's Manual. Good programming practices are highlighted in all chapters with special Good Programming Practice boxes, and common errors are highlighted in Programming Pitfalls boxes. End-of-chapter materials include Summaries of Good Programming Practice and Summaries of Fortran Statements and Structures. Finally, a detailed description of every Fortran intrinsic procedure is included in Appendix C, and an extensive Glossary is included in Appendix E.

The book is accompanied by an Instructor's Manual, containing the solutions to all end-of-chapter exercises. Instructors can also download the solutions in the Instructor's Manual from the book's Web site. The source code for all examples in the book, plus other supplemental materials, can be downloaded by anyone from the book's Web site.

A NOTE ABOUT FORTRAN COMPILERS

Two Fortran compilers were used during the preparation of this book: the Intel Visual Fortran Compiler Version 16.0 and the GNU G95 Fortran compiler. Both compilers provide essentially complete implementations of Fortran 2008, with only a very few minor items not yet implemented. They are also both looking to the future, implementing features from the proposed Fortran 2015 standard.

I highly recommend both compilers to potential users. The great advantage of Intel Fortran is the very nice integrated debugging environment, and the great disadvantage is cost. The G95 compiler is free, but it is somewhat harder to debug.

A FINAL NOTE TO THE USER

No matter how hard I try to proofread a document like this book, it is inevitable that some typographical errors will slip through and appear in print. If you should spot any such errors, please drop me a note via the publisher, and I will do my best to get them eliminated from subsequent printings and editions. Thank you very much for your help in this matter.

I will maintain a complete list of errata and corrections at the book's World Wide Web site, which is www.mhhe.com/chapman4e. Please check that site for any updates and/or corrections.

ACKNOWLEDGMENTS

I would like to thank Raghu Srinivasan and the team at McGraw-Hill Education for making this revision possible. In addition, I would like to thank my wife Rosa and daughter Devorah for their support during the revision process. (In previous editions, I had thanked our other seven children as well, but they have all now flown the coop!)

Stephen J. Chapman
Melbourne, Victoria, Australia
August 7, 2016

1

Introduction to Computers and the Fortran Language

OBJECTIVES

- Know the basic components of a computer.
- Understand binary, octal, and hexadecimal numbers.
- Learn about the history of the Fortran language.

The computer was probably the most important invention of the twentieth century. It affects our lives profoundly in very many ways. When we go to the grocery store, the scanners that check out our groceries are run by computers. Our bank balances are maintained by computers, and the automatic teller machines and credit and debit cards that allow us to make banking transactions at any time of the day or night are run by more computers. Computers control our telephone and electric power systems, run our microwave ovens and other appliances, and control the engines in our cars. Almost any business in the developed world would collapse overnight if it were suddenly deprived of its computers. Considering their importance in our lives, it is almost impossible to believe that the first electronic computers were invented just about 75 years ago.

Just what is this device that has had such an impact on all of our lives? A **computer** is a special type of machine that stores information, and can perform mathematical calculations on that information at speeds much faster than human beings can think. A **program,** which is stored in the computer's memory, tells the computer what sequence of calculations is required, and which information to perform the calculations on. Most computers are very flexible. For example, the computer on which I write these words can also balance my checkbook, if I just execute a different program on it.

Computers can store huge amounts of information, and with proper programming, they can make that information instantly available when it is needed. For example, a bank's computer can hold the complete list of all the deposits and debits made by every one of its customers. On a larger scale, credit companies use their computers to hold the credit histories of every person in the United States—literally billions of pieces of information. When requested, they can search through those billions of pieces of information to recover the credit records of any single person, and present those records to the user in a matter of seconds.

It is important to realize that *computers do not think as humans understand thinking*. They merely follow the steps contained in their programs. When a computer appears to be doing something clever, it is because a clever person has written the program that it is executing. That is where we humans come into the act. It is our collective creativity that allows the computer to perform its seeming miracles. This book will help teach you how to write programs of your own, so that the computer will do what *you* want it to do.

1.1
THE COMPUTER

A block diagram of a typical computer is shown in Figure 1-1. The major components of the computer are the **central processing unit (CPU), main memory, secondary memory,** and **input** and **output devices.** These components are described in the paragraphs below.

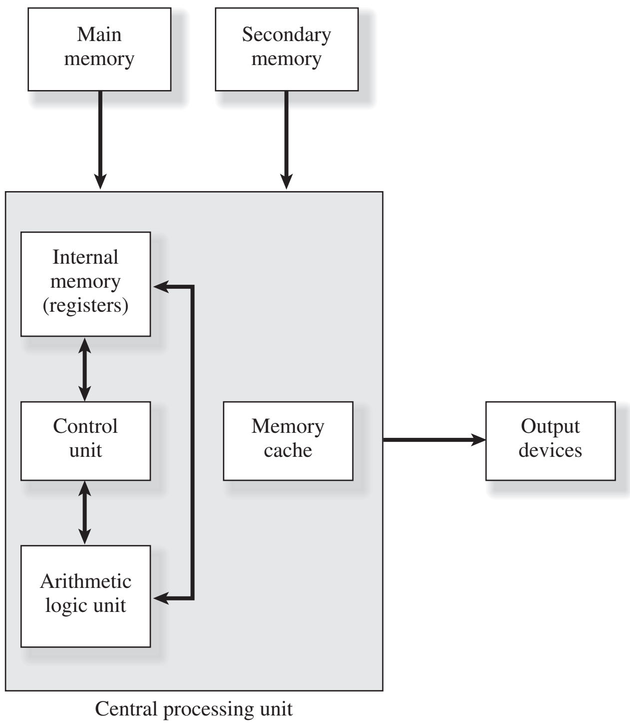


FIGURE 1-1
A block diagram of a typical computer.

1.1.1 The CPU

The central processing unit is the heart of any computer. It is divided into a *control unit*, an *arithmetic logic unit (ALU)*, and internal memory. The control unit within the CPU controls all of the other parts of the computer, while the ALU performs the actual mathematical calculations. The internal memory within a CPU consists of a series of *memory registers* used for the temporary storage of intermediate results during calculations, plus a *memory cache* to temporarily store data that will be needed in the near future.

The control unit of the CPU interprets the instructions of the computer program. It also fetches data values from main memory (or the memory cache) and stores them in the memory registers, and sends data values from memory registers to output devices or main memory. For example, if a program says to multiply two numbers together and save the result, the control unit will fetch the two numbers from main memory and store them in registers. Then, it will present the numbers in the registers to the ALU along with directions to multiply them and store the results in another register. Finally, after the ALU multiplies the numbers, the control unit will take the result from the destination register and store it back into the memory cache. (Other parts of the CPU copy the data from the memory cache to main memory in slower time.)

Modern CPUs have become dramatically faster by incorporating multiple ALUs running in parallel, allowing more operations to be performed in a given amount of time. They also incorporate larger memory caches on the CPU chip, allowing data to be fetched and saved very rapidly.

1.1.2 Memory

The memory of a computer is divided into three major types of memory: *cache* memory, *main* or *primary memory*, and *secondary memory*. Cache memory is memory stored on the CPU chip itself. This memory can be accessed very rapidly, allowing calculations to proceed at very high speed. The control unit looks ahead in the program to see what data will be needed, and pre-fetches it from main memory into the memory cache so that it can be used with minimal delay. The control unit also copies the results of calculations from the cache back to main memory when they are no longer needed.

Main memory usually consists of separate semiconductor chips connected to the CPU by conductors called a *memory bus*. It is very fast, and relatively inexpensive compared to the memory on the CPU itself. Data that is stored in main memory can be fetched for use in a few nanoseconds or less (sometimes *much* less) on a modern computer. Because it is so fast and cheap, main memory is used to temporarily store the program currently being executed by the computer, as well as the data that the program requires.

Main memory is not used for the permanent storage of programs or data. Most main memory is **volatile,** meaning that it is erased whenever the computer's power is turned off. Besides, main memory is relatively expensive, so we only buy enough to hold all of the programs actually being executed at any given time.

Secondary memory consists of devices that are slower and cheaper than main memory. They can store much more information for much less money than main memory can. In addition, most secondary memory devices are **nonvolatile,** meaning that they retain the programs and data stored in them whenever the computer's power is turned off. Typical secondary memory devices are **hard disks,** solid-state drives (SSD), USB memory sticks, and DVDs. Secondary storage devices are normally used to store programs and data that are not needed at the moment, but that may be needed some time in the future.

1.1.3 Input and Output Devices

Data is entered into a computer through an input device, and is output through an output device. The most common input devices on a modern computer are the keyboard and the mouse. We can type programs or data into a computer with a keyboard. Other types of input devices found on some computers include touchscreens, scanners, microphones, and cameras.

Output devices permit us to use the data stored in a computer. The most common output devices on today's computers are displays and printers. Other types of output devices include plotters and speakers.

1.2 DATA REPRESENTATION IN A COMPUTER

Computer memories are composed of billions of individual switches, each of which can be ON or OFF, but not at a state in between. Each switch represents one **binary digit** (also called a **bit**); the ON state is interpreted as a binary 1, and the OFF state is interpreted as a binary 0. Taken by itself, a single switch can only represent the numbers 0 and 1. Since we obviously need to work with numbers other than 0 and 1, a number of bits are grouped together to represent each number used in a computer. When several bits are grouped together, they can be used to represent numbers in the *binary* (base 2) *number system.*

The smallest common grouping of bits is called a **byte.** *A byte is a group of 8 bits that are used together to represent a binary number.* The byte is the fundamental unit used to measure the capacity of a computer's memory. For example, the personal computer on which I am writing these words has a main memory of 24 gigabytes (24,000,000,000 bytes) and a secondary memory (disk drive) with a storage of 2 terabytes (2,000,000,000,000 bytes).

The next larger grouping of bits in a computer is called a **word.** A word consists of 2, 4, or more consecutive bytes that are used to represent a single number in memory. The size of a word varies from computer to computer, so words are not a particularly good way to judge the size of computer memories. Modern CPUs tend to use words with lengths of either 32 or 64 bits.

1.2.1 The Binary Number System

In the familiar base 10 number system, the smallest (rightmost) digit of a number is the ones place (10^0). The next digit is in the tens place (10^1), and the next one is in the hundreds place (10^2), etc. Thus, the number 122_{10} is really $(1 \times 10^2) + (2 \times 10^1) + (2 \times 10^0)$. Each digit is worth a power of 10 more than the digit to the right of it in the base 10 system (see Figure 1-2*a*).

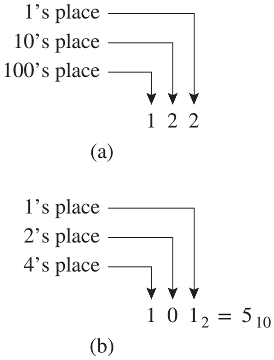


FIGURE 1-2
(*a*) The base 10 number 122 is really $(1 \times 10^2) + (2 \times 10^1) + (2 \times 10^0)$. (*b*) Similarly, the base 2 number 101_2 is really $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$.

Similarly, in the binary number system, the smallest (rightmost) digit is the ones place (2^0). The next digit is in the twos place (2^1), and the next one is in the fours place (2^2), etc. Each digit is worth a power of 2 more than the digit to the right of it in the base 2 system. For example, the binary number 101_2 is really $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 5$, and the binary number $111_2 = 7$ (see Figure 1-2*b*).

Note that three binary digits can be used to represent eight possible values: 0 (= 000_2) to 7 (= 111_2). In general, *if n bits are grouped together to form a binary number, then they can represent 2^n possible values.* Thus, a group of 8 bits (1 byte) can represent 256 possible values, a group of 16 bits (2 bytes) can be used to represent 65,536 possible values, and a group of 32 bits (4 bytes) can be used to represent 4,294,967,296 possible values.

In a typical implementation, half of all possible values are reserved for representing negative numbers, and half of the values are reserved for representing zero plus the positive numbers. Thus, a group of 8 bits (1 byte) is usually used to represent numbers between −128 and +127, including 0, and a group of 16 bits (2 bytes) is usually used to represent numbers between −32,768 and +32,767, including 0.[1]

TWO'S COMPLEMENT ARITHMETIC

The most common way to represent negative numbers in the binary number system is the two's complement representation. What is two's complement, and what is so special about it? Let's find out.

The Two's Complement Representation of Negative Numbers

In the two's complement representation, the leftmost bit of a number is the *sign bit*. If that bit is 0, then the number is positive; if it is 1, then the number is negative. To change a positive number into the corresponding negative number in the two's complement system, we perform two steps:

1. Complement the number (change all 1s to 0 and all 0s to 1).
2. Add 1 to the complemented number.

[1] The most common scheme for representing negative numbers in a computer's memory is the so-called *two's complement* representation, which is described in the sidebar.

1

Let's illustrate the process using simple 8-bit integers. As we already know, the 8-bit binary representation of the number 3 would be 00000011. The two's complement representation of the number −3 would be found as follows:

1. Complement the positive number: 11111100
2. Add 1 to the complemented number: 11111100 + 1 = 11111101

Exactly the same process is used to convert negative numbers back to positive numbers. To convert the number −3 (11111101) back to a positive 3, we would:

1. Complement the negative number: 00000010
2. Add 1 to the complemented number: 00000010 + 1 = 00000011

Two's Complement Arithmetic

Now we know how to represent numbers in two's complement representation, and to convert between positive and two's complement negative numbers. The special advantage of two's complement arithmetic is that *positive and negative numbers may be added together according to the rules of ordinary addition without regard to the sign, and the resulting answer will be correct, including the proper sign*. Because of this fact, a computer may add any two integers together without checking to see what the signs of the two integers are. This simplifies the design of computer circuits.

Let's do a few examples to illustrate this point.

1. Add 3 + 4 in two's complement arithmetic.

3	00000011
+4	00000100
7	00000111

2. Add (−3) + (−4) in two's complement arithmetic.

3	11111101
+−4	11111100
−7	111111001

In a case like this, we ignore the extra ninth bit resulting from the sum, and the answer is 11111001. The two's complement of 11111001 is 00000111 or 7, so the result of the addition was −7!

3. Add 3 + (−4) in two's complement arithmetic.

−3	00000011
+−4	11111100
−1	11111111

The answer is 11111111. The two's complement of 11111111 is 00000001 or 1, so the result of the addition was −1.

With two's complement numbers, binary addition comes up with the correct answer regardless of whether the numbers being added are both positive, both negative, or mixed.

1.2.2 Octal and Hexadecimal Representations of Binary Numbers

Computers work in the binary number system, but people think in the decimal number system. Fortunately, we can program the computer to accept inputs and give its outputs in the decimal system, converting them internally to binary form for processing. Most of the time, the fact that computers work with binary numbers is irrelevant to the programmer.

However, there are some cases in which a scientist or engineer has to work directly with the binary representations coded into the computer. For example, individual bits or groups of bits within a word might contain status information about the operation of some machine. If so, the programmer will have to consider the individual bits of the word, and work in the binary number system.

A scientist or engineer who has to work in the binary number system immediately faces the problem that binary numbers are unwieldy. For example, a number like 1100_{10} in the decimal system is 010001001100_2 in the binary system. It is easy to get lost working with such a number! To avoid this problem, we customarily break binary numbers down into groups of 3 or 4 bits, and represent those bits by a single base 8 (octal) or base 16 (hexadecimal) number.

To understand this idea, note that a group of 3 bits can represent any number between 0 (= 000_2) and 7 (= 111_2). These are the numbers found in an **octal** or base 8 arithmetic system. An octal number system has seven digits: 0 through 7. We can break a binary number up into groups of 3 bits, and substitute the appropriate octal digit for each group. Let's use the number 010001001100_2 as an example. Breaking the number into groups of three digits yields $010|001|001|100_2$. If each group of 3 bits is replaced by the appropriate octal number, the value can be written as 2114_8. The octal number represents exactly the same pattern of bits as the binary number, but it is more compact.

Similarly, a group of 4 bits can represent any number between 0 (= 0000_2) and 15 (= 1111_2). These are the numbers found in a **hexadecimal** or base 16 arithmetic system. A hexadecimal number system has 16 digits: 0 through 9 and A through F. Since the hexadecimal system needs 16 digits, we use digits 0 through 9 for the first 10 of them, and then letters A through F for the remaining 6. Thus, $9_{16} = 9_{10}$, $A_{16} = 10_{10}$, $B_{16} = 11_{10}$, and so forth. We can break a binary number up into groups of 4 bits, and substitute the appropriate hexadecimal digit for each group. Let's use the number 010001001100_2 again as an example. Breaking the number into groups of four digits yields $0100|0100|1100_2$. If each group of 4 bits is replaced by the appropriate hexadecimal number, the value can be written as $44C_{16}$. The hexadecimal number represents exactly the same pattern of bits as the binary number, but more compactly.

Some computer vendors prefer to use octal numbers to represent bit patterns, while other computer vendors prefer to use hexadecimal numbers to represent bit patterns. Both representations are equivalent, in that they represent the pattern of bits in a compact form. A Fortran language program can input or output numbers in any of the four formats (decimal, binary, octal, or hexadecimal). Table 1-1 lists the decimal, binary, octal, and hexadecimal forms of the numbers 0 to 15.

1

TABLE 1-1
Table of decimal, binary, octal, and hexadecimal numbers

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

1.2.3 Types of Data Stored in Memory

Three common types of data are stored in a computer's memory: **character data, integer data,** and **real data** (numbers with a decimal point). Each type of data has different characteristics, and takes up a different amount of memory in the computer.

Character Data

The **character data** type consists of characters and symbols. A typical system for representing character data in a non-Oriental language must include the following symbols:

1. The 26 uppercase letters A through Z
2. The 26 lowercase letters a through z
3. The 10 digits 0 through 9
4. Miscellaneous common symbols, such as `",(), {}, [], !, ~, @, #, $, %, ^, &, and *.`
5. Any special letters or symbols required by the language, such as à, ç, ë, and £.

Since the total number of characters and symbols required to write Western languages is less than 256, *it is customary to use 1 byte of memory to store each character*. Therefore, 10,000 characters would occupy 10,000 bytes of the computer's memory.

The particular bit values corresponding to each letter or symbol may vary from computer to computer, depending upon the coding system used for the characters. The most important coding system is ASCII, which stands for the American Standard Code for Information Interchange (ANSI X3.4 1986, or ISO/IEC 646:1991). The ASCII coding system defines the values to associate with the first 128 of the 256 possible values that can be stored in a 1-byte character. The 8-bit codes corresponding to each letter and number in the ASCII coding system are given in Appendix A.

The second 128 characters that can be stored in a 1-byte character are *not* defined by the ASCII character set, and they used to be defined differently depending on the language used in a particular country or region. These definitions are a part of the ISO 8859 standard series, and they are sometimes referred to as "code pages." For example, the ISO 8859-1 (Latin 1) character set is the version used in Western European countries. There are similar code pages available for Eastern European languages, Arabic, Greek, Hebrew, and so forth. Unfortunately, the use of different code pages made the output of programs and the contents of files appear different in different countries. As a result, these code pages are falling out of favor, and being replaced by the Unicode system described below.

Some Oriental languages such as Chinese and Japanese contain more than 256 characters (in fact, about 4000 characters are needed to represent each of these languages). To accommodate these languages and all of the other languages in the world, a coding system called Unicode[2] has been developed. In the Unicode coding system, each character is stored in 2 bytes of memory, so the Unicode system supports 65,536 possible different characters. The first 128 Unicode characters are identical to the ASCII character set, and other blocks of characters are devoted to various languages such as Chinese, Japanese, Hebrew, Arabic, and Hindi. When the Unicode coding system is used, character data can be represented in any language.

Integer Data

The **integer data** type consists of the positive integers, the negative integers, and zero. The amount of memory devoted to storing an integer will vary from computer to computer, but will usually be 1, 2, 4, or 8 bytes. Four-byte integers are the most common type in modern computers.

Since a finite number of bits are used to store each value, only integers that fall within a certain range can be represented on a computer. Usually, the smallest number that can be stored in an n-bit integer is

$$\text{Smallest integer value} = -2^{n-1} \tag{1-1}$$

and the largest number that can be stored in an n-bit integer is

$$\text{Largest integer value} = 2^{n-1} - 1 \tag{1-2}$$

For a 4-byte integer, the smallest and largest possible values are −2,147,483,648 and 2,147,483,647, respectively. Attempts to use an integer larger than the largest possible

[2] Also referred to by the corresponding standard number, ISO/IEC 10646:2014.