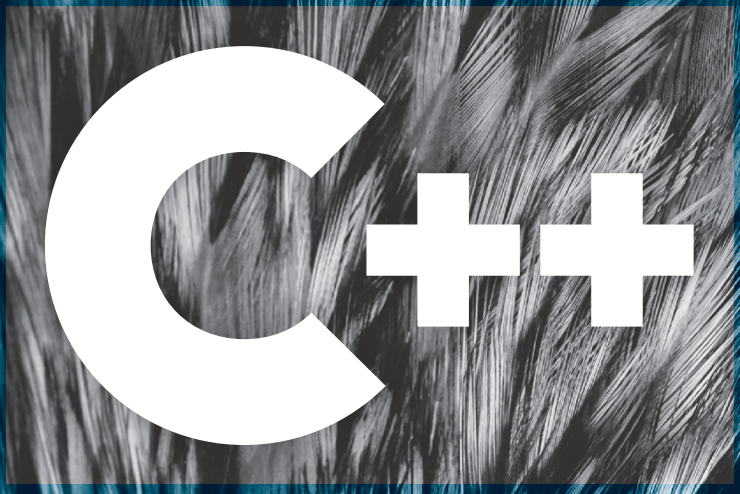

C++ PLUS DATA STRUCTURES

NELL DALE
University of Texas, Austin

CHIP WEEMS
University of Massachusetts, Amherst

TIM RICHARDS
University of Massachusetts, Amherst

SIXTH EDITION

JONES & BARTLETT
LEARNING

World Headquarters
Jones & Bartlett Learning
5 Wall Street
Burlington, MA 01803
978-443-5000
info@jblearning.com
www.jblearning.com

Jones & Bartlett Learning books and products are available through most bookstores and online booksellers. To contact Jones & Bartlett Learning directly, call 800-832-0034, fax 978-443-8000, or visit our website, www.jblearning.com.

Substantial discounts on bulk quantities of Jones & Bartlett Learning publications are available to corporations, professional associations, and other qualified organizations. For details and specific discount information, contact the special sales department at Jones & Bartlett Learning via the above contact information or send an email to specialsales@jblearning.com.

09816-7

Production Credits
VP, Executive Publisher: David D. Cella
Executive Editor: Matt Kane
Acquisitions Editor: Laura Pagluica
Editorial Assistant: Taylor Ferracane
Senior Production Editor: Amanda Clerkin
Marketing Manager: Amy Langlais
VP, Manufacturing and Inventory Control: Therese Connell
Composition: S4Carlisle Publishing Services
Cover Design: Kristin E. Parker
Rights & Media Specialist: Merideth Tumasz
Media Development Editor: Shannon Sheehan
Cover Image: © Repina Valeriya/Shutterstock
Printing and Binding: Edwards Brothers Malloy
Cover Printing: Edwards Brothers Malloy

Library of Congress Cataloging-in-Publication Data
Names: Dale, Nell (Nell B.), author. | Weems, Chip, author. | Richards, Tim (Computer scientist), author.
Title: C++ plus data structures / Nell Dale, Chip Weems, Tim Richards, UMass Amherst, Dept of Computer Science, MA.
Description: Sixth edition. | Burlington, Massachusetts : Jones & Bartlett Learning, [2018] | Includes bibliographical references and index.
Identifiers: LCCN 2016009602 | ISBN 9781284089189 (casebound)
Subjects: LCSH: C++ (Computer program language) | Data structures (Computer science)
Classification: LCC QA76.73.C153 D334 2017 | DDC 005.13/3--dc23 LC record available at http://lccn.loc.gov/2016009602

6048

Printed in the United States of America
20 19 18 17 16 10 9 8 7 6 5 4 3 2 1

Contents

Preface

With this edition, two new authors come on board to carry forward the tradition of excellence in *C++ Plus Data Structures,* as Nell steps back from leading its future development. Chip Weems has been Nell's coauthor on numerous other titles for 30 years, including *Java Plus Data Structures,* and contributed significantly to the pedagogical approach that set the tone for earlier versions of this text and its predecessors. Tim Richards was heavily involved with Chip and Nell in developing the most recent edition of *Progamming and Problem Solving in C++.* Together, they share a strong commitment to the success of students everywhere, which is, of course, the foundation for the love of teaching that motivates us all to walk into our classrooms each day.

Over the last two decades, the traditional data structures course has grown to include the broader topics of Abstract Data Types (ADTs), software engineering, and elementary analysis of algorithms.

The term *data structures* refers to the study of how to represent collections of data in organized relationships and how to write algorithms that manipulate them. The term *Abstract Data Type (ADT)* refers to a description of data in terms of a set of defining properties, as well as the operations that can be applied to the data. The shift in emphasis is representative of the move toward more abstraction in computer science education. We now study the abstract properties of classes of data objects, in addition to how the objects might be represented in a program. Johannes J. Martin puts it very succinctly: "[D]epending on the point of view, a data object is characterized by its type (for the user) or by its structure (for the implementor)."[1]

The design of the abstraction and the implementation are both tied critically to software engineering, which seeks to apply engineering methodologies to the development of reliable, robust, and correct software. A poor abstraction can lead to a cumbersome set of use cases that force application programmers to either write unnecessarily complex code or neglect important validity checks. A poor implementation can be inefficient or prone to error. One aspect of designing

[1] Johannes J. Martin, *Data Types and Data Structures*, Prentice-Hall International Series in Computer Science, C. A. R. Hoare, series editor, Prentice-Hall International (UK), Ltd., 1986, p. 1.

an efficient implementation is being able to analyze the work done by a given algorithm. Thus, throughout this text, we distinguish between the engineering of abstractions and implementations as motivated by their applications, and we take the time to analyze the algorithms that we introduce.

Three Levels of Abstraction

The focus of *C++ Plus Data Structures* is on ADTs as viewed from three different perspectives: their specification, their application, and their implementation. The specification describes the logical or abstract level and is concerned with *what* the data type represents. The application level is concerned with the use of the data in solving problems. This level is concerned with *why* the data type has particular properties and operations. The implementation level is where the operations are actually coded; it is concerned with the *how* questions.

Within this focus, we stress computer science theory and software engineering principles, including modularization, data encapsulation, information hiding, data abstraction, object-oriented decomposition, functional decomposition, the analysis of algorithms, and life-cycle software verification methods. We feel strongly that these principles should be introduced to computer science students early in their education so they learn to practice good software techniques from the beginning.

To teach these concepts to students who may not have completed many college-level mathematics courses, we consistently use intuitive explanations, even for topics that have a basis in mathematics, like the analysis of algorithms. In all cases, our highest goal has been to make our explanations as readable and easily understandable as possible.

Prerequisite Assumptions

In this book, we assume that students are familiar with the following C++ constructs:

- Built-in simple data types
- Stream input/output (I/O), as provided in `<iostream>`
- Stream I/O, as provided in `<fstream>`
- Control structures *while, do...while, for, if,* and *switch*
- User-defined functions with value and reference parameters
- Built-in array types
- Class construct

We include sidebars within the text to review some of the details of these topics.

Updates to the Sixth Edition

General Changes and Reorganization The key changes in this revised sixth edition are primarily focused on the second half of the book. In a previous revision, all the sorts were moved into one chapter to make it easier to analyze them together. But we find that quick sort is such a natural demonstration of recursion that we elected to move it back to Chapter 7, and now the sorting chapter, Chapter 12, simply reviews it as the basis for complexity analysis.

Chapter 9, which had become something of an ADT catchall, is now focused entirely on heaps and the closely related Priority Queue ADT. We have also moved coverage of the heap sort algorithm into this chapter, and Chapter 12 reviews it sufficiently to motivate its analysis and comparison with other sorts. Based on user feedback, we have added an entirely new Chapter 10, called "Trees Plus," to address AVL, Red-Black, and B-trees. Chapter 11 then gathers together coverage of associative containers, adding a new section on the Map ADT to the existing material on the Set ADT, and hashing.

Sorting is covered in Chapter 12. But instead of introducing a wide range of algorithms, we limit the new sorts to merge sort and radix sort and emphasize the analysis of sorting complexity, as well as other efficiency considerations, in choosing among sorting algorithms. New to this chapter is a discussion of caching and its effects on performance, along with coverage of the C++ thread library and a parallel version of merge sort. The latter is a key element of the 2013 ACM curriculum update, which was based in part on the curriculum recommendations of the IEEE Technical Committee on Parallel Processing, to which Chip was a key contributor.

We felt that this book's extensive coverage of graphs warranted its own chapter, Chapter 13, rather than being lumped in with several unrelated ADTs. This also reflects a philosophy that the second half of a comprehensive data structures text must be more modular, so instructors can tailor coverage of the different ADTs to the circumstances of their particular classes.

C++ 11 Since the fifth edition came out, the C++ 11 standard has become much more widely accessible. Thus, we have begun to incorporate some of its new features in this book. In particular, we now cover range-based *for* loops (in Chapter 6) and the new thread library (in Chapter 12).

Content and Organization

Chapter 1 outlines the basic goals of high-quality software and the basic principles of software engineering for designing and implementing programs to meet these goals. Abstraction, functional decomposition, and object-oriented design are discussed. This chapter also addresses what we see as a critical need in software education: the ability to design and implement correct programs and to verify that they are actually correct. Topics covered include the concept of life-cycle verification; designing for correctness using preconditions and postconditions; the use of deskchecking and design/code walk-throughs and inspections to identify errors before testing; debugging techniques, data coverage (black box), and code coverage (clear or white-box) approaches; and test plans, unit testing, and structured integration testing using stubs and drivers. The concept of a generalized test driver is presented and executed in a case study that develops the ADT Fraction.

Chapter 2 presents data abstraction and encapsulation, the software engineering concepts that relate to the design of the data structures used in programs. Three perspectives of data are discussed: abstraction, implementation, and application. These perspectives are illustrated using a real-world example (a library) and then are applied to built-in data structures that C++ supports—namely, structs and arrays. The C++ class type is presented as the way to represent the ADTs that we examine in subsequent chapters. The principles of object-oriented programming—encapsulation, inheritance, and polymorphism—are introduced here, along with the accompanying C++ implementation constructs. The case study at the end of this chapter reinforces the ideas

of data abstraction and encapsulation in designing and implementing a user-defined data type representing a date. This class is tested using a version of the generalized test driver.

Chapter 2 includes a discussion of two C++ constructs that help users write better software: namespace and exception handling using the *try/catch* statement. Various approaches to error handling are demonstrated in subsequent chapters.

Because there is more than one way to solve a problem, we discuss how competing solutions can be compared through the analysis of algorithms using Big-O notation. Throughout the rest of the book, the ADT implementations are compared using Big-O notation. The case study defines the ADT Date, implements the class, defines a test plan, and implements the test plan.

We would like to think that the material in Chapters 1 and 2 is a review for most students. However, the concepts in these two chapters are so crucial to the future of any and all students that we feel that we cannot rely on their having seen the material before.

Chapter 3 introduces the most fundamental ADT of them all: the unsorted list. The chapter begins with a general discussion of operations on ADTs and then presents the framework with which all the other data types are examined: a presentation and discussion of the specification, a brief application using the operations, and the design and coding of the operations. The specification is of the ADT Unsorted List. An array-based implementation of the specification is developed.

The concept of dynamic allocation is introduced, along with the syntax for using C++ pointer variables. The concept of linking nodes to form lists is presented in clear detail, with many diagrams. This technique is then used to reimplement the unsorted list. The array-based and linked implementations are compared using Big-O notation. The case study designs the ADTs Card and Deck to represent a deck of playing cards. These are implemented as classes and tested.

Chapter 4 introduces the ADT Sorted List and develops an array-based implementation. The binary search is introduced as a way to improve the performance of the search operation in the sorted list. The ADT is then implemented using a linked implementation. These implementations are compared using Big-O notation.

Both the logical and physical distinctions between bound and unbound structures are discussed. The four-phase, object-oriented methodology is presented and demonstrated in the case study, which evaluates hands according to the rules of Texas hold 'em poker.

Chapter 5 introduces the ADTs Stack and Queue. Each ADT is first considered from its abstract perspective, and the idea of recording the logical abstraction in an ADT specification is stressed. The operations are used in an application program; then the set of operations is implemented in C++ using an array-based implementation, followed by a linked implementation. The case study simulates a solitaire game, using the classes created in the chapter.

Chapter 6 is a collection of advanced concepts and techniques. Templates are introduced as a way of implementing generic classes. Circular linked lists and doubly linked lists are discussed. The insertion, deletion, and list traversal algorithms are developed and implemented for each variation. An alternative representation of a linked structure, using static allocation (an array of structs), is designed. Class copy constructors, operator overloading, and dynamic binding are covered in detail. The concept of an iterator is introduced, and we see how the range-based *for* loop can be used to implement iteration over a list. The case study uses doubly linked lists to implement large integers.

Chapter 7 presents recursion, giving students an intuitive understanding of the concept, and then shows how recursion can be used to solve programming problems. Guidelines for writing

recursive functions are illustrated with many examples. After demonstrating that a by-hand simulation of a recursive routine can be very tedious, a simple three-question technique is introduced for verifying the correctness of recursive functions. Because many students are wary of recursion, the introduction to this material is deliberately intuitive and nonmathematical. A more detailed discussion of how recursion works leads to an understanding of how it can be replaced with iteration and stacks. As a concrete example of a significant algorithm that is simplified via recursion, we introduce quick sort. The case study develops and implements the process of escaping from a maze.

Chapter 8 introduces ADT Binary Search Tree as a way to arrange data, giving the flexibility of a linked structure with $O(\log_2 N)$ insertion and deletion time. In order to build on the previous chapter and exploit the inherent recursive nature of binary trees, the algorithms first are presented recursively. After all the operations have been implemented recursively, we code the insertion and deletion operations iteratively to show the flexibility of binary search trees. We conclude with a complexity-based comparison of operations on linear lists versus binary search trees. The case study discusses the process of building an index for a manuscript and implements the first phase.

Chapter 9 is motivated by the Priority Queue ADT, which can be implemented easily using the Heap ADT. The heap, in turn, is efficiently built using an array-based implementation of the Binary Tree ADT. Thus, we begin with an overview of priority queues at the abstract level and then consider how a complete binary tree can be implemented using a nonlinked, array-based representation. In introducing the shape property of the heap, which requires it to be a complete binary tree, it is clear that the preceding tree implementation is a perfect basis for a heap implementation. We then return to applying the heap as a means of implementing the priority queue. As a further demonstration of the utility of the heap, we introduce the *heap sort* algorithm.

Chapter 10 is entirely new. It extends the idea of the binary search tree to more advanced forms that are self-balancing. The basic concepts are introduced with the AVL Tree ADT, which maintains a strict balance. We then relax the balance constraint with the ADT Red-Black tree. Finally, the balanced tree concept is extended beyond the binary form to higher orders of tree structures, through a high-level exploration of the B-tree ADT. Although this chapter does refer to the heap, it is possible to introduce it immediately after Chapter 8 if an instructor has a strong preference to do so.

Chapter 11 gathers together the ADTs Set and Map, which are associative structures that are commonly used for lookup operations. In the case of the set, lookup determines whether an element is in the structure, while with the map, we are interested in also retrieving a value that is paired with a specified key. Given that lookup is one of the motivations for these structures, it is natural to also introduce the idea of hashing here, as it can be used for efficient implementation of associative structures. Because the map implementation is based on binary search trees, this chapter can be covered at any point after Chapter 8.

Chapter 12 presents a number of sorting and searching algorithms and asks the question: Which are better? The comparison-based sorting algorithms that are illustrated include straight selection sort, two versions of bubble sort, quick sort, heap sort, and merge sort. The merge sort is the the only new algorithm in this section. We then compare the sorting algorithms using Big-O notation. Then we address other practical efficiency considerations in sorting, including the size of the data set, function call overhead, programmer time, memory space, stability of keys, sorting pointer arrays, and caching. Radix sort is presented as an example of sorting wihout directly comparing values, and then it is analyzed. Finally, we return to the merge sort as an example

of an algorithm that is easy to parallelize using the C++ thread library. The sorting discussion depends only on prior covereage of recursion, quick sort, and heap sort, so it can be used at any point after Chapter 9.

Chapter 13 relaxes the regularity that was present in the structures of the earlier ADTs by introducing the concept of the Graph ADT and then exploring an adjacency matrix implementation, as well as two approaches to creating an adjacency list. Breadth-first and depth-first searches are also introduced and compared. The coverage in this chapter depends only on stacks and queues, so it could be covered at any point after Chapter 5.

Additional Features

Chapter Goals A set of goals presented at the beginning of each chapter helps the students assess what they have learned. These goals are tested in the exercises at the end of each chapter.

Chapter Exercises Chapter exercises vary in levels of difficulty, including short programming problems, the analysis of algorithms, and problems to test the student's understanding of concepts. The answer key for the exercises can be found in the *Instructor's Manual*.

Case Studies There are eight case studies. Each includes a problem description, an analysis of the problem input and required output, and a discussion of the appropriate data types to use. Most of the case studies are completely coded and tested. Two are left partially complete, requiring students to complete and test the final versions.

Student and Instructor Resources Source code for all programs, partial programs, and case studies within the text is available for students and instructors to download at go.jblearning.com/c++plusds6e. In addition, instructors may access the following resources:

- The *Instructor's Manual,* with goals, teaching notes, workouts (suggestions for in-class activities), programming assignments for each chapter, and answers to the end-of-chapter exercises
- Slides in Microsoft PowerPoint format
- A test bank

Acknowledgments

First, we would like to thank those who replied to our survey concerning this new edition. Respondents included both users and nonusers of the previous editions:

Barbara Bracken, PhD
Associate Professor
Department of Mathematics and Computer Science
Wilkes University

Lionel L. Craddock
Bluefield State College

John Dean
Professor of Computer Science and Chair
Department of Computer Science and Information Systems
Park University

Muhammad Ghanbari
California Polytechnic State University

Carolyn Golden, PhD
Associate Professor of Computer Science
Huston-Tillotson University

Terry R. Hostetler
Professor of Computer Science
Coe College

Amitava Karmaker
Associate Professor
University of Wisconsin-Stout

Fred L. Strickland, PhD
Adjunct Instructor
Computer Science Department
Troy University Montgomery Campus

Dr. Rajendran Swamidurai
Associate Professor of Computer Science
Alabama State University

Jiaofei Zhong, PhD
California State University, East Bay

Your comments were invaluable: Thank you.

Thanks to our families, who have been such a support over the last year and a half. As anyone who has ever worked on a textbook knows, it involves many long hours and late nights that are stolen from family time. We cannot begin to express our appreciation for your willingness to enable us to undertake this project.

A virtual bouquet of roses to the people who have worked on this book: Laura Pagluica, Taylor Ferracane, Amanda Clerkin, and Escaline Charlette Aarthi.

N. D.
C. W.
T. R.