preface

Historically, a course on data structures has been a mainstay of most computer science departments. However, the focus of this course has broadened considerably. The topic of data structures now has been subsumed under the broader topic of *abstract data types* (*ADTs*)—the study of classes of objects whose logical behavior is defined by a set of values and a set of operations.

The term *abstract data type* describes a comprehensive collection of data values and operations. The term *data structures* refers to the study of data and how to represent data objects within a program; that is, the implementation of structured relationships. The shift in emphasis is representative of the move towards more abstraction in computer science education. We now are interested in the study of 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: ". . . depending on the point of view, a data object is characterized by its type (for the user) or by its structure (for the implementor)."[1]

Three Levels of Abstraction

The focus of *C++ Plus Data Structures* is on abstract data types as viewed from three different perspectives: their specification, their application, and their implementation. The specification describes the logical or abstract level. This level is concerned with *what* the operations do. The application level, sometimes called the user level, is concerned with how the data type might be used to solve a problem. This level is concerned with *why* the operations do what they do. The implementation level is where the operations are actually coded. This level 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

[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.

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 that they learn to practice good software techniques from the beginning.

An understanding of theoretical concepts helps students put the new ideas they encounter into place, and practical advice allows them to apply what they have learned. 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 as easily understandable as possible.

Prerequisite Assumptions

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

- Built-in simple data types
- Stream 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 have included sidebars within the text to refresh the student's memory concerning some of the details of these topics.

Updates to the Fifth Edition

Terminology Object-oriented terminology now dominates computing vocabulary. For example, Put and Get are used to insert and retrieve items in/from a structure. The ADT operation names have been updated to reflect this terminology.

The use of an output parameter to define the meaning of a function return value is common in most language libraries, such as the Standard Template Library and the Java Collections Framework. Although this technique flies in the face of the principle that a function has only one return value, the students need to know this technique in order to read library code. Thus, the list operation Get returns the complete list item found or the same item for which it searched, depending on the value of found, a Boolean parameter.

The GetNextItem operation is now a function rather than a procedure with a return parameter. This structure is more convenient, because the function can appear within a statement that processes the item.

Case Studies A Case Study has been added to Chapter 3 and the Case Studies in Chapters 4 and 5 have been replaced. These three case studies are related in that they refer to

a deck of playing cards. The ADTs Card and Deck are introduced in Chapter 3. Chapter 4 uses these ADTs to construct and evaluate Poker hands. Chapter 5 uses these ADTs to simulate a Solitaire game.

Exercises Additional exercises have been added to most chapters.

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 walkthroughs and inspections to identify errors before testing; debugging techniques, data coverage (black box), and code coverage (clear or white box) approaches; 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: structs and arrays. The C++ class type is presented as the way to represent the abstract data types 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 abstract data type of them all: the unsorted list. The chapter begins with a general discussion of operations on abstract data types and then presents the framework with which all of 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 together to form lists is presented in clear detail with many diagrams. This technique is then used to re-implement 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 the 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 that 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 Case Study uses doubly linked lists to implement large integers.

Chapter 7 presents recursion, giving the student 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 recursion can be replaced with iteration and stacks. The Case Study develops and implements the process of escaping form a maze.

Chapter 8 introduces binary search trees 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. A nonlinked array-based binary tree implementation is described. The Case Study discusses the process of building an index for a manuscript and implements the first phase.

Chapter 9 presents a collection of other branching structures: priority queues (implemented with both lists and heaps), graphs, and sets. The graph algorithms make use of stacks, queues, and priority queues, thus both reinforcing earlier material and demonstrating how general these structures are. Two set implementations are discussed: the bit-vector representation in which each item in the base set is assigned a present/absent flag and the operations are the built-in logic operations, and a list-based representation in which each item in a set is represented in a list of set items. If the item is not in the list, it is not in the set.

Chapter 10 presents a number of sorting and searching algorithms and asks the question: Which are better? The sorting algorithms that are illustrated, implemented, and compared include straight selection sort, two versions of bubble sort, quick sort, heap sort, and merge sort. The sorting algorithms are compared using Big-O notation. The discussion of algorithm analysis continues in the context of searching. Previously presented searching algorithms are reviewed and new ones are described. Hashing techniques are discussed in some detail. Finally, radix sort is presented and analyzed.

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 Most chapters have more than 40 exercises. They 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 the student to complete and test the final versions.

Student and Instructor Resources Source code for all programs, partial programs, and case studies within the text are available for student and instructor download at go.jblearning.com/ndale. In addition, instructors may access the following resources:

- 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.
- PowerPoint Presentations
- Test Bank

Acknowledgments

First we would like to thank the twenty-four people who replied to our Web survey concerning this new edition. Respondents included both users and non-users of one of the previous editions. Your comments were invaluable: Thank you.

Thanks to my friends and family, who have been such a support over the last year. Thanks to my tennis friends, who kept me fit, and my bridge friends, who challenged my mind.

A virtual bouquet of roses to the people who have worked on this book: Mike and Sigrid Wile along with our Jones & Bartlett Learning family: Tim Anderson, our editor, and Amy Rose, our production goddess.

N. D.