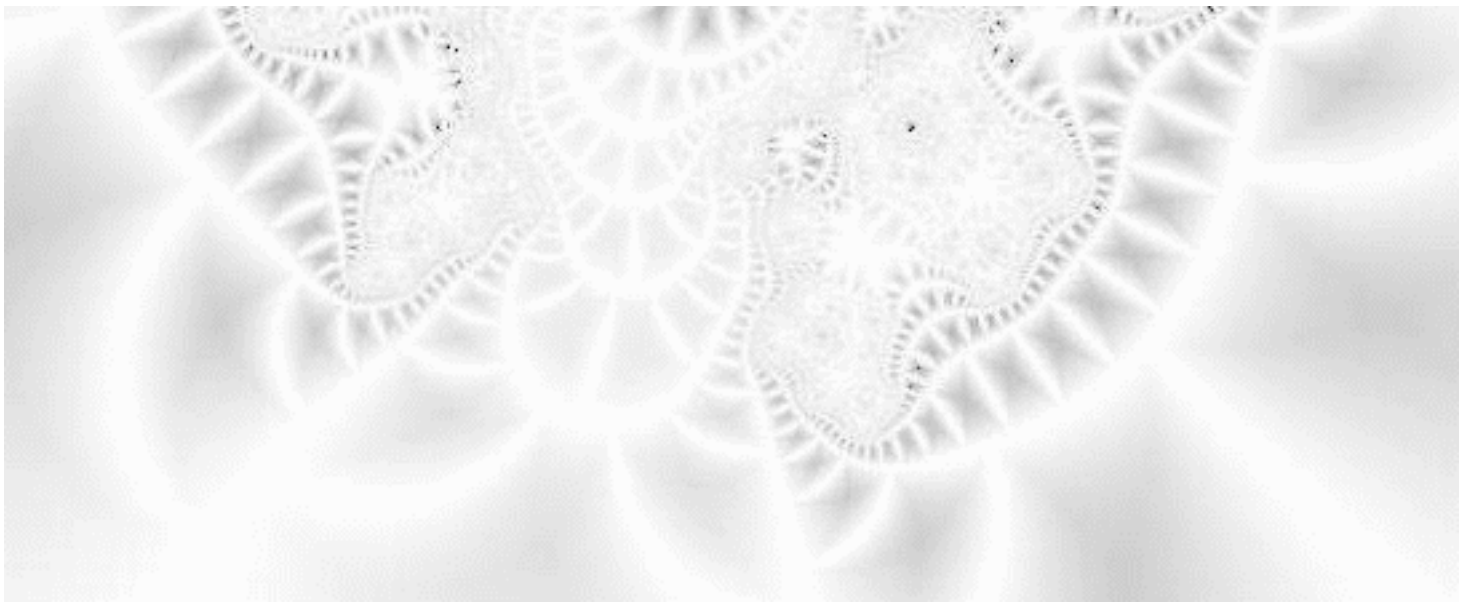


# Preface

This book is written for a second course in computer science, the CS 2 course at many universities. The text's emphasis is on the *specification*, *design*, *implementation*, and *use* of the basic data types that normally are covered in a second-semester course. In addition, we cover a range of important programming techniques and provide self-contained coverage of abstraction techniques, object-oriented programming, big-*O* time analysis of algorithms, and sorting.

We assume that the student has already had an introductory computer science and programming class, but we do include coverage of those topics (such as recursion and pointers) that are not always covered completely in a first course. The text uses C++, but our coverage of C++ classes begins from scratch, so the text may be used by students whose introduction to programming was in C rather than C++. In our experience, such students need a brief coverage of C++ input and output techniques (such as those provided in Appendix F) and some coverage of C++ parameter types (which we provide in Chapter 2). When C programmers are over the input/output hurdle and the parameter hurdle (and perhaps a small "fear" hurdle), they can step readily into classes and other object-oriented features of C++. As this indicates, there are several pathways through the text that can be tailored to different backgrounds, including some optional features for the student who comes to the class with a stronger than usual background.

### New to This Edition

The C++ Standard Template Library (STL) plays a larger role in our curriculum than past editions, and we have added selected new material to support this. For us, it's important that our students understand both how to use the STL classes in an application program and the possible approaches to imple-

menting these (or similar) classes. With this in mind, the primary changes that you'll find for this edition are:

- A new Section 2.6 that gives an early introduction to the Standard Template Library using the `pair` class. We have been able to introduce students to the STL here even before they have a full understanding of templates.
- An earlier introduction of the multiset class and STL iterators in Section 3.4. This is a good location for the material because the students have just seen how to implement their first collection class (the bag), which is based on the multiset.
- We continue to introduce the STL string class in Section 4.5, where it's appropriate for the students to implement their own string class with a dynamic array.
- A new Section 5.6 that compares three similar STL classes: the vector, the list, and the deque. At this point, the students have enough knowledge to understand typical vector and list implementations.
- A first introduction to the STL algorithms appears in Section 6.3, and this is now expanded on in Sections 11.2 (the heap algorithms) and 13.4 (expanded coverage of sorting and binary search in the STL).
- A new Section 8.4 provides typical implementation details for the STL deque class using an interesting combination of dynamic arrays and pointers.
- A discussion of hash tables in the proposed TR1 expansions for the STL is now given in Section 12.6.

Most chapters also include new programming projects, and you may also keep an eye on our project web site, www.cs.colorado.edu/~main/dsoc.html, for new projects as we develop them.

### The Steps for Each Data Type

Overall, the fourth edition remains committed to the data types: *sets*, *bags* (or *multisets*), *sequential lists*, *ordered lists* (with ordering from a "less than" operator), *stacks*, *queues*, *tables*, and *graphs*. There are also additional supplemental data types such as a priority queue. Each of these data types is introduced following a consistent pattern:

**Step 1: Understand the data type abstractly.** At this level, a student gains an understanding of the data type and its operations at the level of concepts and pictures. For example, a student can visualize a stack and its operations of pushing and popping elements. Simple applications are understood and can be carried out by hand, such as using a stack to reverse the order of letters in a word.

**Step 2: Write a specification of the data type as a C++ class.** In this step, the student sees and learns how to write a specification for a C++ class that can

implement the data type. The specification includes prototypes for the constructors, public member functions, and sometimes other public features (such as an underlying constant that determines the maximum size of a stack). The prototype of each member function is presented along with a precondition/postcondition contract that completely specifies the behavior of the function. At this level, it's important for the students to realize that the specification is not tied to any particular choice of implementation techniques. In fact, this same specification may be used several times for several different implementations of the same data type.

**Step 3: Use the data type.** With the specification in place, students can write small applications or demonstration programs to show the data type in use. These applications are based solely on the data type's specification, as we still have not tied down the implementation.

**Step 4: Select appropriate data structures, and proceed to design and implement the data type.** With a good abstract understanding of the data type, we can select an appropriate data structure, such as a fixed-sized array, a dynamic array, a linked list of nodes, or a binary tree of nodes. For many of our data types, a first design and implementation will select a simple approach, such as a fixed-sized array. Later, we will redesign and reimplement the same data type with a more complicated underlying structure.

Since we are using C++ classes, an implementation of a data type will have the selected data structures (arrays, pointers, etc.) as private member variables of the class. With each implemented class, we stress the necessity for a clear understanding of the rules that relate the private member variables to an abstract notion of the data type. We require each student to write these rules in clear English sentences that we call the *invariant of the abstract data type*. Once the invariant is written, students can proceed to implementing various member functions. The invariant helps in writing correct functions because of two facts: (a) Each function (except constructors) knows that the invariant is true when the function begins its work; and (b) each function (except the destructor) is responsible for ensuring that the invariant is again true when the function finishes.

**Step 5: Analyze the implementation.** Each implementation can be analyzed for correctness, flexibility (such as a fixed size versus dynamic size), and time analysis of the operations (using big-*O* notation). Students have a particularly strong opportunity for these analyses when the same data type has been implemented in several different ways.

### Where Will the Students Be at the End of the Course?

At the end of our course, students understand the data types inside out. They know how to use the data types, they know how to implement them several ways, and they know the practical effects of the different implementation choices. The students can reason about efficiency with a big-*O* analysis and

argue for the correctness of their implementations by referring to the invariant of the class.

One of the important lasting effects of the course is the specification, design, and implementation experience. The improved ability to reason about programs is also important. But perhaps most important of all is the exposure to classes that are easily used in many situations. The students no longer have to write everything from scratch. We tell our students that someday they will be thinking about a problem, and they will suddenly realize that a large chunk of the work can be done with a bag, or a stack, or a queue, or some such. And this large chunk of work is work that they won't have to do. Instead, they will pull out the bag or stack or queue or some such that they wrote this semester—using it with no modifications. Or, more likely, they will use the familiar data type from a library of standard data types, such as the C++ Standard Template Library. In fact, the behavior of the data types in this text is a cut-down version of the Standard Template Library, so when students take the step to the real STL, they will be on familiar ground. And at that point of realization, knowing that a certain data type is the exact solution he or she needs, the student becomes a real programmer.

### Other Foundational Topics

Throughout the course, we also lay a foundation for other aspects of "real programming," with coverage of the following topics beyond the basic data structures material:

**Object-oriented programming.** The foundations of object-oriented programming (OOP) are laid by giving students a strong understanding of C++ classes. The important aspects of classes are covered early: the notion of a member function, the separation into private and public members, the purpose of constructors, and a small exposure to operator overloading. This is enough to get students going and excited about classes.

Further major aspects of classes are introduced when the students first use dynamic memory (Chapter 4). At this point, the need for three additional items is explained: the copy constructor, the overloaded assignment operator, and the destructor. Teaching these OOP aspects with the first use of dynamic memory has the effect of giving the students a concrete picture of dynamic memory as a resource that can be taken and must later be returned.

Conceptually, the largest innovation of OOP is the software reuse that occurs via inheritance. And there are certainly opportunities for introducing inheritance right from the start of a data structures course (such as implementing a set class as a descendant of a bag class). However, an early introduction may also result in juggling too many new concepts at once, resulting in a weaker understanding of the fundamental data structures. Therefore, in our own course we introduce inheritance at the end as a vision of things to come. But the introduction to inheritance (Sections 14.1 and 14.2) could be covered as soon as copy constructors are

understood. With this in mind, some instructors may wish to cover Chapter 14 earlier, just before stacks and queues.

Another alternative is to identify students who already know the basics of classes. These students can carry out an inheritance project (such as the ecosystem of Section 14.2 or the game engine in Section 14.3) while the rest of the students first learn about classes.

**Templates.** Template functions and template classes are an important part of the proposed Standard Template Library, allowing a programmer to easily change the type of the underlying item in a container class. Template classes also allow the use of several different instantiations of a class in a single program. As such, we think it's important to learn about and use templates (Chapter 6) prior to stacks (Chapter 7), since expression evaluation is an important application that uses two kinds of stacks.

**Iterators.** Iterators are another important part of the proposed Standard Template Library, allowing a programmer to easily step through the items in a container object (such as the elements of a set or bag). Such iterators may be *internal* (implemented with member functions of the container class) or *external* (implemented by a separate class that is a friend of the container class). We introduce internal iterators with one of the first container classes (a sequential list in Section 3.2). An internal iterator is added to the bag class when it is needed in Chapter 6. At that point, the more complex external iterators also are discussed, and students should be aware of the advantages of an external iterator. Throughout the text, iterators provide a good opportunity for programming projects, such as implementing an external bag iterator (Chapter 6) or using a stack to implement an internal iterator of a binary search tree (Chapter 10).

**Recursion.** First-semester courses sometimes introduce students to recursion. But many of the first-semester examples are tail recursion, where the final act of the function is the recursive call. This may have given students a misleading impression that recursion is nothing more than a loop. Because of this, we prefer to avoid early use of tail recursion in a second-semester course. For example, list traversal and other operations on linked lists can be implemented with tail recursion, but the effect may reinforce wrong impressions about recursion (and the tail recursive list operations may need to be unlearned when the students work with lists of thousands of items, running into potential run-time stack overflow).

So, in our second-semester course, we emphasize recursive solutions that use more than tail recursion. The recursion chapter provides three examples along these lines. Two of the examples—generating random fractals and traversing a maze—are big hits with the students. In our class, we teach recursion (Chapter 9) just before trees (Chapter 10), since it is in recursive tree algorithms that recursion becomes vital. However, instructors who desire more emphasis on recursion can move that topic forward, even before Chapter 2.

In a course that has time for advanced tree projects (Chapter 11), we analyze the recursive tree algorithms, explaining the importance of keeping the trees balanced—both to improve worst-case performance, and to avoid potential run-time stack overflow.

**Searching and sorting.** Chapters 12 and 13 provide fundamental coverage of searching and sorting algorithms. The searching chapter reviews binary search of an ordered array, which many students will have seen before. Hash tables also are introduced in the search chapter. The sorting chapter reviews simple quadratic sorting methods, but the majority of the chapter focuses on faster algorithms: the recursive merge sort (with worst-case time of $O(n \log n)$), Tony Hoare's recursive quicksort (with average-time $O(n \log n)$), and the tree-based heap sort (with worst-case time of $O(n \log n)$). There is also a new introduction to the C++ Standard Library sorting functions.

### Advanced Projects

The text offers good opportunities for optional projects that can be undertaken by a more advanced class or by students with a stronger background in a large class. Particular advanced projects include the following:

- A polynomial class using dynamic memory (Section 4.6).
- An introduction to Standard Library iterators, culminating in an implementation of an iterator for the student's bag class (Sections 6.3 through 6.5).
- An iterator for the binary search tree (Programming Projects in Chapter 10).
- A priority queue, implemented with a linked list (Chapter 8 projects), or implemented using a heap (Section 11.1).
- A set class, implemented with B-trees (Section 11.3). We have made a particular effort on this project to provide information that is sufficient for students to implement the class without need of another text. In our courses, we have successfully directed advanced students to do this project as independent work.
- An inheritance project, such as the ecosystem of Section 14.2.
- An inheritance project using an abstract base class such as the `game` base class in Section 14.3 (which allows easy implementation of two-player games such as Othello or Connect Four).
- A `graph` class and associated graph algorithms from Chapter 15. This is another case where advanced students may do work on their own.

### C++ Language Features

C++ is a complex language with many advanced features that will not be touched in a second-semester course. But we endeavor to provide complete coverage for those features that we do touch. In the first edition of the text, we included coverage of two features that were new to C++ at the time: the new `bool` data type (Figure 2.1 on page 37) and static member constants (see page 104). The requirements for using static member constants were changed in the 1998 Standard, and we have incorporated this change into the text (the constant must now be declared both inside and outside the class definition). The other primary new feature from the 1998 Standard is the use of namespaces, which were incorporated in the second edition. In each of these cases, these features might not be supported in older compilers. We provide some assistance in dealing with this (see Appendix E, "Dealing with Older Compilers"), and some assistance in downloading and installing the GNU g++ compiler (see Appendix K).

### Flexibility of Topic Ordering

This book was written to allow instructors latitude in reordering the material to meet the specific background of students or to add early emphasis to selected topics. The dependencies among the chapters are shown on page xi. A line joining two boxes indicates that the upper box should be covered before the lower box.

Here are some suggested orderings of the material:

**Typical course.** Start with Chapters 1–10, skipping parts of Chapter 2 if the students have a prior background in C++ classes. Most chapters can be covered in a week, but you may want more time for Chapter 5 (linked lists), Chapter 6 (templates), Chapter 9 (recursion), or Chapter 10 (trees). Typically, we cover the material in 13 weeks, including time for exams and extra time for linked lists and trees. Remaining weeks can be spent on a tree project from Chapter 11, or on binary search (Section 12.1) and sorting (Chapter 13).

**Heavy OOP emphasis.** If students cover sorting and searching elsewhere, there will be time for a heavier emphasis on object-oriented programming. The first four chapters are covered in detail, and then derived classes (Section 14.1) are introduced. At this point, students can do an interesting OOP project, based on the ecosystem of Section 14.2 or the games in Section 14.3. The basic data structures are then covered (Chapters 5–8), with the queue implemented as a derived class (Section 14.3). Finish up with recursion (Chapter 9) and trees (Chapter 10), placing special emphasis on recursive member functions.

**Accelerated course.** Assign the first three chapters as independent reading in the first week, and start with Chapter 4 (pointers). This will leave two to three

extra weeks at the end of the term, so that students may spend more time on searching, sorting, and the advanced topics (shaded on page xi.)

We also have taught the course with further acceleration by spending no lecture time on stacks and queues (but assigning those chapters as reading).

**Early recursion / early sorting.** One to three weeks may be spent at the start of class on recursive thinking. The first reading will then be Chapters 1 and 9, perhaps supplemented by additional recursive projects.

If recursion is covered early, you may also proceed to cover binary search (Section 12.1) and most of the sorting algorithms (Chapter 13) before introducing C++ classes.

### Supplements via the Internet

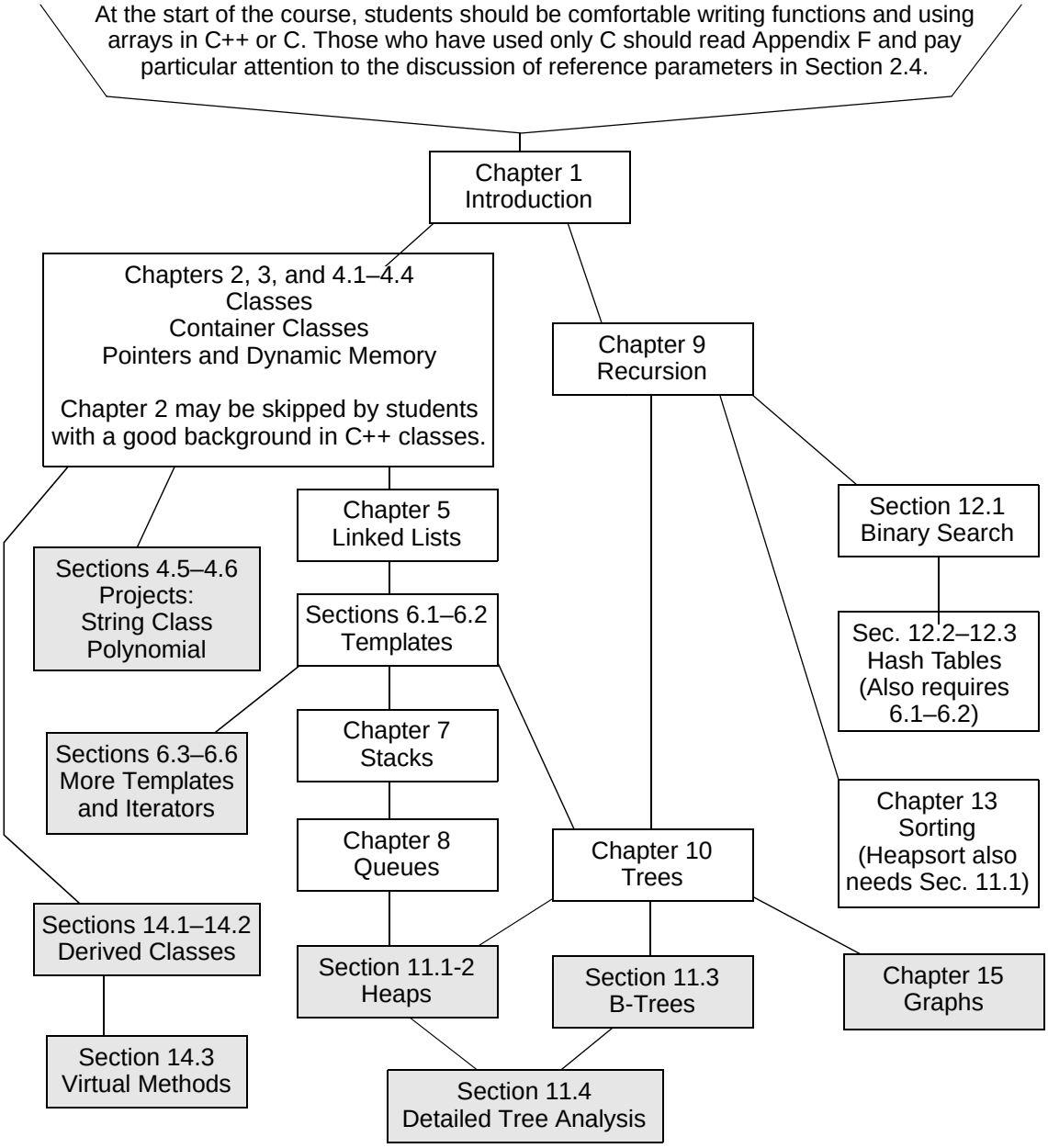
The following supplemental materials for this text are available to all readers at www.aw-bc.com/cssupport:

- Source code. All the C++ classes, functions, and programs that appear in the book are available to readers.
- Errata. We have tried not to make mistakes, but sometimes they are inevitable. A list of detected errors is available and updated as necessary. You are invited to contribute any errors you find.

In addition, the following supplements are available to qualified instructors at www.pearsonhighered.com/irc. Please contact your Addison-Wesley sales representative, or send email to computing@aw.com, for information on how to access them:

- PowerPoint lecture slides
- Exam questions
- Solutions to selected programming projects
- Sample assignments and lab exercises
- Suggested syllabi

**Chapter Dependencies**




**The shaded boxes provide good opportunities for advanced work.**

**Acknowledgments**

We started this book while Walter was visiting Michael at the Computer Science Department of the University of Colorado in Boulder. The work was completed after Walter moved back to the Department of Engineering and Computer Science at the University of California, San Diego. We are grateful to these institutions for providing facilities, wonderful students, and interaction with congenial colleagues.

Our students have been particularly helpful—nearly 5000 of our students worked through the material, making suggestions, showing us how they learned. We thank the reviewers and instructors who used the material in their data structures courses and provided feedback: Zachary Bergen, Cathy Bishop, Martin Burtscher, Gina Cherry, Courtney Comstock, Stephen Davies, Robert Frohardt, John Gillett, Mike Hendricks, Ralph Hollingsworth, Yingdan Huang, Patrick Lynn, Ron McCarty, Shivakant Mishra, Evi Nemeth, Rick Osborne, Rachelle Reese, and Nicholas Tran. The book was also extensively reviewed by Wolfgang W. Bein, Bill Hankley, Michael Milligan, Paul Nagin, Jeff Parker, Andrew L. Wright, John R. Rose, and Evan Zweifel. We thank these colleagues for their excellent critique and their encouragement.

Thank you to Lesley McDowell and Chris Schenk, who are pleasant and enthusiastic every day in the computer science department at the University of Colorado. Our thanks also go to the editors and staff at Addison-Wesley. Heather McNally's work has encouraged us and provided us with smooth interaction on a daily basis and eased every step of the production. Karin Dejamaer and Jessica Hector provided friendly encouragement in Boulder, and we offer our thanks to them. We welcome and appreciate Michael Hirsch in the role of editor, where he has shown amazing energy, enthusiasm, and encouragement. Finally, our original editor, Susan Hartman, has provided continual support, encouragement, and direction—the book wouldn't be here without you!

In addition to the work and support from those who put the book together, we thank those who offered us daily interest and encouragement. Our deepest thanks go to Holly Arnold, Vanessa Crittenden, Meredith Boyles, Suzanne Church, Erika Civils, Lynne Conklin, Andrzej Ehrenfeucht, Paul Eisenbrey, Skip Ellis, John Kennedy, Rick Lowell, George Main, Mickey Main, Jesse Nuzzi, Ben Powell, Marga Powell, Megan Powell, Grzegorz Rozenberg, Hannah, Timothy, and Janet.

---

Michael Main
main@colorado.edu
*Boulder, Colorado*

Walter Savitch
wsavitch@ucsd.edu
*San Diego, California*