# Teaching Data Structures Using Object Oriented Toolkits

Josh Tenenberg
Computing and Software Systems
Institute of Technology
University of Washington, Tacoma
1900 Commerce St
Tacoma WA 98402-3100
(253) 692-4521
(253) 692-4424 (fax)
jtenenbg@u.washington.edu

## Abstract

The emergence of object-oriented toolkit libraries of classic data structures and algorithms such as the Standard Template Library and Java's Collection classes has provided a set of general and efficient software components to practicing software developers. This paper advocates the incorporation of such toolkits in the first Data Structures course at the university level. This represents a paradigm shift from learning the details of each data structure to an increased emphasis on the use, integration, and extension of these existing toolkits. In addition, this paper argues that studying the toolkits themselves enables students to learn the higher-order design and engineering skills, particular using object oriented techniques, that are embedded within the toolkits.

## Shifting the data structures curricular paradigm

Why is it that I ask the students in my advanced classes not to use their own data structure implementations in their programming assignments, but rather to use those in standard libraries such as Standard Template Library (STL) and Java's Collection classes? How many times in the past have I seen a junior or senior struggling to debug their "thought this was working" linked list, or alter a not-very-generic hash table leaving them with little time for the important parts of the assignment, such as the genetic algorithm, the parser, the graph search algorithm? Why do I urge my graduating students to reuse, reuse, reuse, especially the code from generic, standardized libraries?

New programming tools and advances both enable and stimulate the demand for new programming practices and pedagogies. As high-level programming languages became more prevalent in the 1960s and 70s, both the demand for high-level language programmers and the support of high-level programming increased. The emergence of these new programming languages effectively brought into existence a new set of virtual machines, with language translators allowing programmers to directly program these new virtual machines with less regard to the specifics of the implementing hardware and more regard to algorithm design than in the previous generation.

It is time for the next curriculum paradigm shift in the data structures curriculum. The existence of robust, standardized, generic toolkits of data structure and algorithm libraries such as STL gives rise to a new set of virtual machines. Our students will be asked to use such toolkits when they leave the university since software development is increasingly driven by the need to design larger and more complex software systems through the integration of existing software components [7]. The data structures course provides an excellent opportunity to begin building these higher-level design skills.

A word on this paper's perspective. As an advocacy paper, I neither present anecdotal evidence from my own general adoption of these principles, i.e. this is not an experience paper, nor do I present results from original empirical studies in support of my hypothesis. Rather, I present a broader philosophical rationale in support of a change in objectives, and some ideas as to how this change might impact the way in which the course is taught. Given the presence of an increasing number of textbooks that include discussion of standardized data structures toolkits (e.g. [1, 2, 3, 4, 5, 22]), the present paper can be viewed as a rationale for adopting such texts and a perspective on their use, something that is, surprisingly, lacking in the texts themselves.

## Standardized Toolkits

Standard libraries have existed for decades to provide functionality that may be required in a large variety of programming applications. They often encode specific and subtle design and performance tradeoffs that have evolved within particular programming communities over many years. They thus serve as repositories of accumulated cultural expertise. As described in [18], programmers become acculturated into programmer communities by learning the knowledge and language that provides communicative economy within that community.

In using the library code, the developer both reduces their debugging time due to the decrease in developer-produced code, and off-loads a portion of the maintenance task to the organizations supporting the library. Further, portability is increased, since virtually all compiler vendors support the standard libraries. Additionally, standard library use can lead to more efficient implementations, since the standardization process enables highly specialized experts to accumulate and embed their expertise within the library itself. For example, both STL and the Collection classes implement associative Maps using Red-Black trees, a complex data structure to program; by using these library classes, the programmer without the skill themselves to code such routines can nonetheless gain their benefits. Standard library reuse thus leverages someone else's intellectual effort and lets the programmer employing reuse be smarter than they are.

Unlike most subroutine libraries, such as C's standard libraries, where specific functionality is encapsulated in stand-alone procedures, toolkits provide sets of classes that both collaborate with one another to leverage their computational power, and are extensible by the programmer so as to be useful in the widest possible settings. As Gamma et al [6, p.26] state "A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality. An example of a toolkit is a set of collection classes for lists, associative tables, stacks, and the like. ... They let you as an implementer avoid recoding common functionality. Toolkits emphasize *code reuse*." Toolkits are also sometimes called *frameworks* (e.g. in [9]); for consistency the toolkits terminology will be used throughout.

The generic and collaborative aspects of toolkits require the use of the most powerful features of object-oriented languages, such as template classes, function objects, method and operator overloading, and polymorphism. In addition, they embed a number of common Design Patterns [6], such as *Template Method* and *Iterator*. With the inclusion of STL in the C++ standard and the addition of the Collection classes to Sun's supported Java API's, programmers using these libraries obtain the benefits of both standardization and the toolkits themselves.

## Objectives of the data structures course

The first undergraduate data structures course in the United States has come to be known as "CS2". This designation was originally given to the second in an eight course undergraduate core curriculum as specified in the Association for Computing Machinery's (ACM) Curriculum '78 recommendations [14]. The objectives and content of CS2 were updated by an ACM Curriculum Task Force in 1984 [10, p.815]resulting in the following objectives (quoted verbatim):

- To continue developing a disciplined approach to the design, coding, and testing of programs written in a block-structured high-level language.
- To teach the use of data abstraction using as examples data structures other than those normally provided as basic types in current programming languages; for example, linked lists, stacks, queues, and trees.
- To provide an understanding of the different implementations of these data structures.
- To introduce searching and sorting algorithms and their analysis.
- To provide a foundation for further studies in computer science.

Despite the ACM's curricular revision in 1991 [19]in which CS2 was deconstructed into a set of knowledge units that would flexibly be reconstructed to meet the needs of particular institutions, CS2 continued to persist throughout the 1990's much as it was codified in 1984. For example, the Preface to the 1999 data structures text by Nyhoff [13, p.v] begins with the sentence: "This text is designed for the course CS2 as described in the curriculum recommendations of the ACM" and continues by enumerating each of the bulleted objectives from [10] quoted above along with an explanation of how each is realized in the text. Similarly, looking just at the chapter headings of another recent data structures text [8] one can discern the pervasive and continuing influence of CS2 as outlined in [10], updated with object orientation : Java Programming; Object-Oriented Design; Analysis Tools; Stacks, Queues, and Dequeus; Vectors, Lists, and Sequences; Trees; Priority Queues; Dictionaries; Search Trees; Sorting, Sets, and Selection; Text Processing; Graphs. Similar tables of contents with minor variations can also be found in [16, 11, 13, 17] among others.

In the ACM's most recent curriculum recommendations [15], CS2 re-emerges (though renumbered) as one of the courses in a suggested introductory sequence using a *programming-first* model: "$CS112_I$ then extends this base by presenting much of the material from the traditional CS2 course [Koffman85], but with an explicit focus on programming in the object-oriented paradigm." Note that the reference to Koffman85 is to the 1984 Curriculum recommendations whose objectives are quoted above.

The approach that I advocate departs from the way these CS2 courses are realized in practice (as exemplified in the texts just cited) by de-emphasizing the implementation details of many of the classic data abstractions, the topic that dominates both CS2 textbooks and class time. I suggest instead that students learn how to *use* the fundamental data structures, and that they learn how to implement data structures and algorithms with similar complexity as those of the standard data structures – especially recursive data structures, i.e., those with pointers or references to one or more instances of the same class – but not necessarily the details of each of these classic structures and algorithms. Further, students should learn to use the data structures that are already implemented in the toolkit libraries, and how to extend these toolkits with their own data structures and algorithms. As Weiss states in [21, p.xx] "My basic premise is that software development tools in all languages come with large libraries, and many data structures are part of

these libraries. I envision an eventual shift in emphasis of data structures courses from implementation to use."

Although there is virtue in the study of elegant and efficient algorithms in general, the goal should not be that students retain the details of specific algorithms and data structures so that they can implement them in a commercial setting. It is more important that students can implement data abstractions of similar complexity, that they can integrate their own abstractions or the abstractions of others into existing toolkit libraries, that they understand the tradeoffs that went into toolkit library design, and that they cultivate the judgment and esthetics to create and use such designs themselves. As Gamma et al write [6, p.26] "Toolkit design is arguably harder than application design, because toolkits have to work in many applications to be useful. Moreover, the toolkit writer isn't in a position to know what those applications will be or their special needs. That makes it all the more important to avoid assumptions and dependencies that can limit the toolkit's flexibility and consequently its applicability and effectiveness."

There are a number of subtle tradeoffs that a toolkit designer must make to balance efficiency, generality, ease of use, and maintainability. Some of these specific tradeoffs for the data structures toolkits include the following. What container classes should be provided, and what generic algorithms? How are the generic algorithms to be applied to the different containers? What operations should be provided by the different containers? What relationship do the different containers have to one another? Novice students do not have the base of expertise to make informed judgments about these tradeoffs. That is, existing data structure libraries encode expertise far beyond what students possess, and hence the architectures for these libraries can be far more complex, subtle, and useful than any library of data structures that novice students would be able to write. Just as students can learn a considerable amount about operating systems in general by studying the code of existing operating systems that are beyond their ability to write from first principles (and similarly with compilers), they can likewise gain considerable knowledge about data structures, architecture and engineering tradeoffs in the study of existing standard data structure libraries.

## Implementing the toolkit approach

How does the use of toolkits in a data structures course differ from a course in which data structures are built from first principles? First, I advocate that the toolkit designs be studied in order to see that modern programming involves sophisticated design techniques that extend above the level of the individual method and object. This study helps students to lift their programming a level above the individual class to the relationship among larger groups of objects that collaborate in complex and powerful ways. Students may thus more fully enter the discipline, not only by engaging in the habits of code reuse and larger scale design, but in learning the specific idioms of a the programmer community to which they are becoming acculturated. Some of this design discussion is included in Bergin's CS2 text ([1], especially Chapter 4 and the beginning of Chapter 7).

Second, students should study the existing source code implementations of the toolkit classes, and alter and extend these, perhaps simplified as Bergin has done [1]. Alternative implementations of some of the containers and generic algorithms can likewise be assigned, and the results compared with the toolkit implementations. For example, the recent CS2 texts by Collins [3], Ford and Topp [5], and Bergin [1] all provide alternative partial implementations of the STL list container.

Third, the students can be assigned the writing of one or more of the classic data structures that the toolkit designers chose not to implement, but doing so in a way that integrates them into the existing toolkit. For example, both Bergin [1] and Collins [3] extend STL with a hash table, not yet part of the STL standard, and Bergin includes an explicit discussion on how to extend STL with new containers (p.292). This integration will also provide them with the discipline of writing code for reuse, e.g. always include an iterator for every collection, accept parameters that are as general as possible, etc.

And finally, programs should also be written that leverage the computational power of the virtual machines that these toolkit libraries bring into existence, such as application problems like file compression or Tic-Tac-Toe with alpha-beta pruning as provided by Weiss in his C++ data structures text [20, Part III]. All too often, students reuse code with cut-and-paste strategies, making domain-specific alterations to standard data structures. By using existing generic toolkits, students can be taught how to both separate the domain-specific aspects of the code into the new data abstractions that they design, and how these new abstractions can collaborate with the existing – and unaltered – toolkits. Since novice student designs often have leaky interfaces, with I/O details and other assumptions about the particular domain of application creeping into what should be generic algorithms and data structures, the use of the toolkit thus encourages better design by enforcing a cleaner separation of responsibilities.

## Conclusion

When our students enter the commercial workplace, they will be asked to write software within a distributed, highly dynamic, multi-person and increasingly multi-national context. This will require much more use of existing software components, such as STL and Enterprise JavaBeans, and the integration of a variety of components together into a coherent program. Budd [2, p.viii] states that "Many authors have predicted that in the future most programs will be constructed by piecing together off-the-shelf components, and the percentage of programs that are developed entirely from scratch will diminish considerably. Therefore, although it may be important for students to know how to construct a linked list, it will be much more important to know how to use the list container in the standard library."

Alexander Stepanov [12, p.xxvii], one of the creators of STL, is even more emphatic about the importance of standardized, component-based software development. "STL presupposes a very different way of teaching computer science. What 99 percent of programmers need to know is not how to build components but how to use them. People who write their own code, instead of using standard components, should be dealt with like people who propose designing nonstandard, proprietary CPUs."

Students of the next generation will be programming a different virtual machine -- one at a much higher level of abstraction – than that of most of us who are teaching these students. Many of our students will make this leap of abstraction without us (or in spite of us). We can, however, aid them in this enterprise, and the increasing availability of textbooks that incorporate these toolkits (e.g. [1, 2, 3, 4, 5, 22]) makes this task considerably easier. The use of standardized component object toolkits in the first university-level Data Structures course brings the component-based paradigm shift that is already occurring in programming practice into the university classroom.

## References


[1] Joseph Bergin. *Data Structures and the Standard Template Library*. Springer, 2003.
[2] T. Budd. *Data Structures in C++ Using the Standard Template Library*. Addison Wesley, 1998.
[3] William Collins. *Data Structures and the Java Collections Framework*. McGraw Hill, 2002.
[4] William Collins. *Data Structures and the Standard Template Library*. McGraw Hill, 2003.
[5] W. Ford and W. Topp. *Data Structures with C++ using STL*. Prentice Hall, 2nd edition, 2001.
[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley, 1994.
[7] W. Gibbs. Software's chronic crisis. *Scientific American*, pages 86--95, 1994.
[8] M. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. Wiley, 2nd edition, 2000.
[9] C. Horstmann. *Core Java 2*, volume II: Advanced Features. Prentice Hall, 2000.
[10] E. Koffman, D. Stemple, and C. Wardle. Recommended curriculum for CS2, 1984. *Communications of the ACM*, 28(8):815--818, 1985.
[11] Michael Main and Walter Savitch. *Data structures and other objects using C++*. Addison Wesley, 2nd edition, 2001.
[12] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide*. Addison Wesley, 2nd edition, 2001.
[13] L. Nyhoff. *C++: An Introduction to Data Structures*. Prentice Hall, 1999.
[14] ACM Curriculum Committee on Computer Science. Curriculum '78: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, 22(3):147--166, 1978.
[15] ACM Joint Task Force on Computing Curricula. *Computing Curricula 2001: Computer Science*. IEEE, 2002.
[16] Bruno Preiss. *Data structures and algorithms with object-oriented design patterns in C++*. Wiley, 1998.
[17] Clifford Shaffer. *Practical Introduction to Data Structures and Algorithm Analysis* (C++ edition). Prentice Hall, 2nd edition, 2001.
[18] J. Tenenberg. On the meaning of computer programs. In M. Beynon, C. Nehaniv, and K Dautenham, editors, *Cognitive Technology: Instruments of Mind. The Fourth International Conference*, pages 165--174. Springer, 2001.
[19] Allen Tucker. Computing curricula 1991. *Communications of the ACM*, 34(6):68--84, 1991.
[20] M. Weiss. *Algorithms, Data Structures, and Problem Solving with C++*. Addison Wesley, 1996.
[21] M. Weiss. *Data Structures and Problem Solving using C++*. Addison Wesley, 2nd edition, 2000.
[22] M. Weiss. *Data Structures and Problem Solving using Java*. Addison Wesley, 2 edition, 2002.