From Data Abstraction to Algorithms

Philip Machanick
Department of Computer Science
University of the Witwatersrand
2050 Wits
philip@cs.wits.ac.za
(011)716-3309 fax (011)339-7965
http://www.cs.wits.ac.za/~philip/

Abstract

In this paper, I report on a major revision of a new (first run in 1995) Computer Science 2 C++-based course, Data Abstraction and Algorithms. The new guiding philosophy is abstraction-first learning, aimed at development of a reuse mentality. An important prerequisite of this approach is the availability of class libraries and frameworks, which allow non-trivial programs to be written without programming from scratch—but which are simple enough to be learnt in a reasonable amount of time. Issues covered in the paper include the content on the course, libraries developed for the course, problems with C++, and results compared with the year before. The new approach appears to be a success in that I was able to considerably increase the range of concepts covered.

1. Introduction

Presenting concepts in the right order is a major challenge in any programming course. In the early years of the use of Pascal as a teaching language, books [Atkinson 1980] tended to present concepts starting with those which could just as well be done in a non-structured language. Examples in early chapters used global variables and monolithic main programs. When procedures were introduced, global variables remained and parameters came later. My observation was that students tended to pick up programming habits based on the earlier examples, which were hard to break. I tried to move more to a model of introducing procedures early, and avoiding the communication of data to procedures, until I felt parameters could be used. Later books followed the order I preferred [Tenenbaum and Augenstein 1986; Garland 1986; Koffman 1992]: my measure of how well an author had adapted with the times became how early procedures were handled. Others have made a similar case—including starting with procedures in Pascal [Ford 1982; Schmaltz 1986] and starting with packages, then moving on to procedures in Ada [Texel 1982]—for starting with tools for abstraction and ending with control structures.

The fundamental issue was that it was necessary to change the style of teaching, if the concept of structured programming was really seen to be superior. If it aided program design and understanding, then shouldn't the beginner's first exposure be to this "easier" methodology? A further issues is to be sure what the fundamental concepts are—if the notion of pushing concepts before detail is accepted [Brookshear 1985]. In other areas, order of learning concepts is believed to be important—for example, it has been observed that learning program correctness is far easier if it is done from the outset than if it is pushed to later in the curriculum[Dupras *et al*. 1984].

In moving to object-oriented programming, similar problems arise. Abstraction and reuse are meant to make programming easier, yet many object-oriented books [Budd 1994; Coplien 1992; Deitel and Deitel 1994; Ford and Topp 1996; Headington and Riley 1994; Lippman 1991; Sedgewick 1992; Stroustrup 1991; Wang 1994; Wiener and Pinson 1990] defer object-oriented concepts to later chapters, and hardly touch on reuse. it is my argument that if these things really make programming easier, they should be done first. If programming from scratch is harder than reuse, do it last.

Consequently, I have redesigned my Computer Science 2 Data Abstraction and Algorithms course around *abstraction-first learning*: abstraction is introduced first, and the sequence of introduction of material is designed to lead from understanding abstraction, through using other people's abstractions to implementing your own.

In this paper, I describe changes in the course, based on abstraction-first, as well as promotion of reuse. By making the course more focused on these two issues, I have been able to cover a wider range of concepts than in a similar course presented the year before [Machanick 1995]—including some concepts previously only covered in third year and honours courses.

The next section describes and justifies the order in which concepts were presented. The section after that outlines some essential tools needed for the course. The following section presents some problems with C++ as a teaching language, in terms of the educational goals and order of presentation aimed for in the course. The section after that discusses the course in more detail. In conclusion, findings and potential improvements are presented.

2. Order of Concepts

The starting point for designing the course is the list of topics to be covered in the course, namely

- Abstract Data Types
- Advanced Data Structures
- Recursive Algorithms

- Object-Oriented Programming
- Complexity Analysis
- Object-Oriented Design and Analysis
- Sorting and Searching
- Scope and Binding
- Problem-Solving Strategies

In terms of the model of starting with abstraction and reuse, these concepts cannot be presented in exactly this form or order, and some topics do not stand alone as units to be handled in a lecture or group of lectures. For example, object-oriented programming is a general concept that applies across several of the other topics. Scope and binding is a complex area, which again can apply in several places.

Since no text book appears to adopt the abstraction-first strategy I advocate, I have written my own. The book is divided into three parts:

- Using Abstraction—abstraction from the outside, including the value of hiding detail and abstraction as a tool in both non-programming and programming situations
- Implementing Abstractions—how C++ can be used to reuse and implement classes, how to work from a design, and how to analyze algorithms and data structures
- Design and Generalization—how to design for generality, including how to implement templates, and more advanced algorithm and data analysis

An important aspect of the course is that implementation from scratch is left as late as possible; in keeping with the notion that abstraction and reuse are the way to start, the course introduces libraries, frameworks and toolboxes early, and examples are developed using these tools (which are described in more detail in the next section).

All the concepts to be covered in the course are fitted into the framework of the three major sections.

On the whole, this order worked well. However, some issues were forced into too early a position because of the language being used. More on this in Section 4.

3. Tools

When the new course started, new equipment was bought for it. For a variety of reasons, we decided to go with Power Macintoshes, and the CodeWarrior environment (which includes a very wide range of tools, not just C++).

```
typedef list <int> IntList;
IntList scores;
scores.push_front (42);   //etc. -- build up the list
typedef IntList::iterator Int_iterator;
int total = 0;
for (Int_iterator i = scores.begin(); i != scores.end(); i++)
   total += *i;
```

(a) Using the C++ Standard Template Library, iterating through containers looks like using pointer arithmetic to iterate through an array

```
typedef T_DoublyLinkedList<int> IntList;   //instantiate template
IntList scores;
scores.insert (42);    // etc. -- build up the list
typedef T_Iterator <IntList, int> Int_iterator;
int total = 0;
Int_iterator numbers (scores);
for (int i=numbers.first (); numbers.more (); i=numbers.next ())
      total += i;
```

(b) Using the Collectibles template library, iterating through containers uses member functions, which may be a more obvious strategy to the uninitiated

Figure 1. Iteration Using STL versus Collectibles

For the latest revision to the course, I have added class libraries in three broad categories: a container class template library, toolboxes, and an application framework.

3.1 Container Class Library

The container class library, called Collectibles, illustrates how to use a template library, as well as common strategies for generalizing containers, including the use of iterators and generators. Implementation of containers in C++ with reasonable generality while keeping them understandable for relative beginners is something of a challenge. The Standard Template Library (STL), for example, achieves generality at the expense of making all containers look like C-style arrays (down to implementing an iterator as if you were incrementing a pointer). No doubt to someone of the old school of C hacking, this seems very clever, but to students whose primary prior exposure to programming is Pascal, iterating through an arbitrary container using a loop as in Figure 1*a* is not necessarily an intuitive way of adding the integers in a container.

Collectibles containers instead provide iterators which require a little more coding to use, but which are more obvious in how they work. Figure 1*b* contains a complete example of an iterator for adding up the contents of a doubly-linked list.

Observe that an iterator is a separate template class, which has to be instantiated for the container type (unfortunately, since C++ has no concept of pattern matching on the structure of a template parameter, the contents type has to be specified separately).

An iterator is applicable in cases where a container can be accessed sequentially, but there are some containers which can't be accessed sequentially. For such containers, it is useful to define a generator—a function which applies a given action to every

```
class IntAddAction : public T_action<int>
{public:
   IntAddAction ();
   virtual ~IntAddAction ();
   virtual void do_each (int data);
   virtual void completion ();
private:
   int total;
};
// constructor: initialize the count
IntAddAction::IntAddAction ()
{  total = 0;
}
// destructor: nothing to clean up in this case
IntAddAction::~IntAddAction ()
{
}

// add a data value from the list onto the total
void IntAddAction::do_each (int data)
{  total += data;
}
// write out the total
void IntAddAction::completion ()
{  cout << total << endl;
}
typedef T_DoublyLinkedList<int> IntList;
IntList scores;
scores.insert (42);    // etc. -- build up the list
IntAddAction addList;
scores.generate (&addList);  // "&" makes a pointer to addList
```

Figure 2. A Complete Generator Example

There are more preliminaries, but there is really less to understand: the loop (or recursion) is hidden in the container class implementation of the generator

element of the container. Since a generator hides the loop or recursion needed to access its elements, it is also the preferred construct, if either could be used. I implemented generators as a function in the container—and an abstract action class template, which had to be both instantiated, and replaced by a derived class which defined the action to be taken.

Compared with the iterator, the preliminaries are more complex, but the final use of the generator is simpler. Since the action class is for a specific contents type—not for a specific container—it can be reused for other purposes.

Use of a generator to do the same as the iterator example is illustrated in Figure 2.

Compared with coding from scratch, there's more to learn, but once the concepts are understood, the students can apply them to any container, without having to think about how it's implemented.

The major containers in the library are a dynamically resizeable array, a binary tree, a stack and a doubly linked list.

Later in the course, where development from scratch is dealt with, new containers are added as needed.

3.2 Toolboxes

The major toolbox I developed for the course is a graphics toolbox, which implements a few simple graphics primitives, to allow the development of examples without deep knowledge of the Macintosh, as well as to illustrate the concept of a toolbox. For more complex examples, I also threw in a simple database toolbox, and a text toolbox.

Each toolbox is intended to be a standalone collection of classes, though the fact that I only have one framework has made it unnecessary to spend much time on generality. However, I did feel it important to emphasize the point that a toolbox, if carefully designed, can apply across different application architectures.

3.3 Application Framework

The application framework is a simplified basis for building a Macintosh style application, using the classic Smalltalk Model-View-Controller (MVC) architecture.

I did not push the use of the framework in much detail, since frameworks in common use differ so much. Rather, I focused on several essentials:

- a framework should be designed around a specific architecture (such as MVC, software bus or compound document)
- a framework should not be designed as a complete class library: toolboxes for specific functionality and container classes, for example, should be packaged separately

Since I did not use the framework to develop very big examples, I was able to keep it much simpler than a conventional industrial-strength framework, such as MacApp or the Microsoft Foundation Classes. However, I was able to illustrate the essential principle: that it was possible to build a complete application, including menus and windows, without any specific knowledge of the control structure of such an application. I was also able to illustrate the relationship between a specific software architecture and a framework (the Smalltalk MVC architecture).

4. C++ Pitfalls

One of the biggest problems with presenting concepts in the order in which I proposed presenting them—the abstraction-first order—was that C++ is not a very abstract language. The underlying machine model is hard to ignore in developing concepts such as parameter passing.

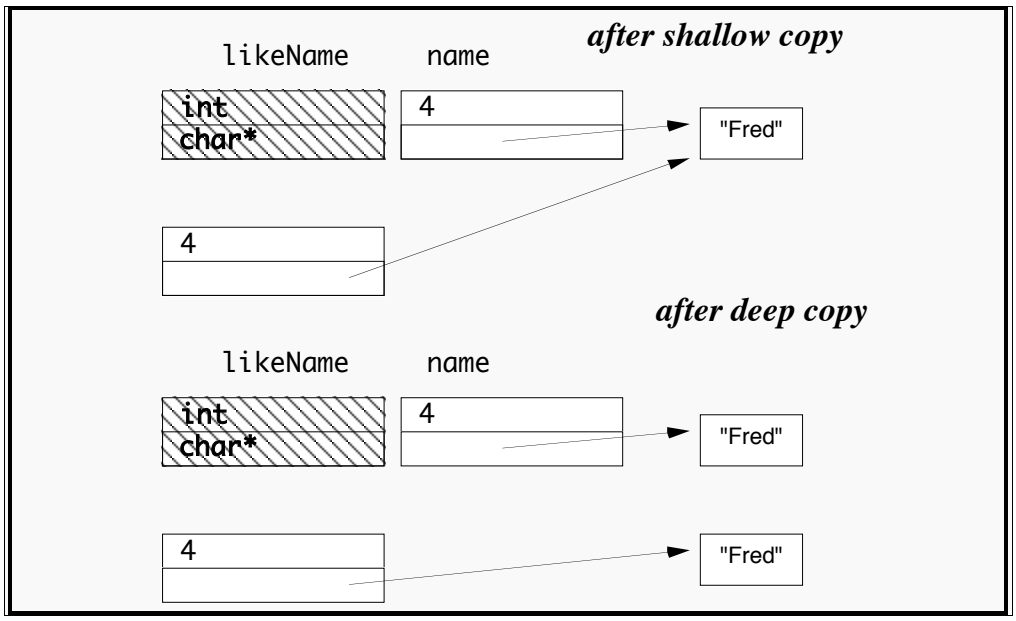

Figure 3. Shallow Versus Deep Copy
after an assignment, with shallow copy, `likeName` and `name` contain an alias, whereas after a deep copy, internal pointers are reallocated; shallow copy is harder to do right, and deep copy is probably what you want anyway for smaller data structures

For example, it appears to be natural to allocate memory for pointers contained in an object in the object's constructor, and to deallocate in the destructor.

Unfortunately, constructors and destructors are invoked at points that may not be immediately obvious. An example is when a parameter (or argument) is passed by value. Pass by value is implemented as initialization of the formal parameter by the value of the actual parameter using a *copy constructor*. I you don't define your own copy constructor, the compiler implements one as a bit-for-bit copy of the original. The result is a *shallow copy*—any internal pointers in the new copy are aliases.

When the function returns, the destructor is called for the formal parameter. If the destructor deallocates the pointers, it leaves dangling pointers in the original object .

Figure 3 illustrates the difference between shallow copy, and *deep copy*, where the new copy also allocates new memory for its internal pointers. To force deep copy, it is necessary to define your own copy constructor. In fact, it is also necessary to define `operator=`, since assignment can have similar consequences.

All of this complication is necessary to properly understand parameter passing and dynamic allocation. Fortunately, I found it possible to defer correct implementation of shallow copy (e.g., using reference counts) to later. Why then should dynamic allocation appear early? Since dynamic dispatch is an important aspect of object-oriented programming, it is important that it be understood early. Dynamic dispatch in

C++ in essence (through the virtual function mechanism) requires dynamic allocation. Also, it's hard to do interesting data structures without any form of dynamic allocation.

Another big problem with C++ is with compiler checking of templates. It's possible that this is the current state of the art, rather than a fundamental flaw in the language, but much checking can only be done when a template is instantiated. If a template class is properly implemented, errors in the template itself are not a problem, but a compiler may sometimes flag errors in instantiation as if they were errors in the template code, which is confusing for the uninitiated.

A final problem is the C++ exception model. I disagree with the semantics. A C++ exception propagates until handled, which in my view is very unsafe—I would prefer that it escalated to a fatal error if not handled in the function that raised it. Consequently, I avoided using exceptions except where they were hidden in a library implementation, which made it hard to convey the benefits of safe error handling. In terms of abstraction first, C++ exceptions are a problem in that they have to be used consistently if used at all, which means they have to be explained very early if used at all.

5. The Course in Detail

Here is how the sections are broken down into chapters.

Part 1—Using Abstraction

- Introduction
- Abstraction—abstraction examples in the real world, and the use of virtual machines in user interface design
- Abstraction and Programming—abstract data types, container classes (templates) including iterators and generators, toolboxes (graphics example) and frameworks
- First Look at C++—file structure of a complete program, class definitions, limited language syntax, testing strategies

Part 2—Implementing Abstractions

- Classes and Objects—dynamic allocation, inheritance and dynamic dispatch, parameter passing, deep and shallow copy, templates versus abstract classes
- Object-Oriented Design—software life cycle, use of Booch diagrams in a design, implementation from a design, difficult areas of the design process

Part 3—Design and Generalization

- Complexity Analysis Introduced—design choices, standard definitions, simple examples with and without recursion (including average case), data structures, relation to detailed design
- Libraries and Frameworks—design for generality, software architecture, designing a template, shallow copy using reference counts, exceptions, implementation of a container class
- More Advanced Complexity—more challenging algorithms, more advanced data structures (balanced trees, hash tables), encryption

In general, the biggest problems were the C++ pitfalls raised earlier. Pacing of the course was not quite as good as I would have liked. The more challenging algorithms analysis section needed more time. However, with all the material ready ahead of time, this problem should not be hard to address next time round.

Compared with the previous run of the course, students' results were very similar. The average this year was 62%, compared with last year's 63%; in both cases, the standard deviation was 12%.

6. Conclusions

Although considerably more ground was covered, the students were able to cope with the course as well as with the previous version. Furthermore, several concepts that were previously only covered in more advanced courses (iterators, generators, exceptions, templates) were successfully integrated into the course. Additional new concepts relative to the 1995 course also include safe implementation of deep and shallow copy, software architectures, and working from a design. A wider range of data structures and algorithms was also covered.

Although a more scientific evaluation is required to assess the value of the abstraction-first approach, the fact that the students' results were similar to those of the year before, despite the introduction of more concepts, is encouraging.

A proper study of the impact of abstraction-first learning would require more than evaluation in the classroom. Concerns have been expressed in industry that reuse is hard [Auer 1995; Berg *et al*. 1995; Fayad and Tsai 1995; Frakes and Fox 1995]. It would therefore be useful to do a longer-term follow up study as to whether students educated in this way perform better in the workplace—i.e., whether abstraction-first learning results in a better appreciation of and ability to adapt to reuse in the long run.

References

Atkinson [1980] L Atkinson. *Pascal Programming*, Wiley, Chichester, 1980.

Auer [1995]. Ken Auer. Smalltalk Training: As Innovative as the Environment, *Comm. ACM*, vol. 38 no. 10 October 1995, pp 115–117.

Berg *et al*. [1995]. William Berg, Marshall Cline and Mike Girou. Lessons Learned from the OS/400 OO Project., *Comm. ACM*, vol. 38 no. 10 October 1995, pp 54–64.

Brookshear [1985] JG Brookshear. The University Computer Science Curriculum: Education Versus Training, *Proc. 16th SIGCSE Symposium on Computer Science Education*, New Orleans, 1985, pp 23–30.

Budd [1994] TA Budd. *Classic Data Structures in C++*, Addison-Wesley, Reading, MA, 1994.

Coplien [1992] JO Coplien. *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.

Deitel and Deitel [1994] HM Deitel and PJ Deitel. *C++ How to Program*, Prentice Hall, Englewood Cliffs, NJ, 1994.

Dupras *et al*. [1984] M Dupras, F LeMay and A Mili. Some Thoughts on Teaching First Year Programming, *Proc. 15th SIGCSE Symposium on Computer Science Education*, New Orleans, 1984, pp 148–152.

Fayad and Tsai [1995]. Mohamed E Fayad and Wei-Tek Tsai. Object-Oriented Experiences, *Comm. ACM*, vol. 38 no. 10 October 1995, pp 51–53.

Ford and Topp [1996] W Ford and W Topp. *Data Structures with C++*, Prentice Hall, Englewood Cliffs, NJ, 1996.

Ford [1982] G Ford. A Software Engineering Approach to First Year Computer Science Courses, *Proc. 13th SIGCSE Symposium on Computer Science Education*, Indianapolis, 1982, pp 8–12.

Frakes and Fox [1995]. Frakes and Fox. Sixteen Questions About Software Reuse, *Comm. ACM*, vol. 38 no. 6 June 1995, pp 75–87,112.

Garland [1986] SJ Garland, *Introduction to Computer Science with Applications in Pascal*, Addison-Wesley, Reading, MA, 1986.

Headington and Riley [1994] MR Headington and DD Riley. *Data Abstraction and Structures Using C++*, DC Heath, Lexington, MA, 1994.

Koffman [1992] EB Koffman. *Pascal* (4th edition), Addison-Wesley, Reading, MA, 1992.

Lippman [1991] SB Lippman. *C++ Primer* (2nd edition), Addison-Wesley, Reading, MA, 1991.

Machanick [1995] P Machanick. From Modula-2 to C++: Advanced Programming with Class, *Proc. 25th Annual SACLA Conference*, July 1995 pp 175–180.

Schmaltz [1986] R Schmaltz. Subprograms in the First Programming Course, *SGICSE Bulletin*, vol. 18 no. 2 June 1986, pp 31–32.

Sedgewick [1992] R Sedgewick. *Algorithms in C++*, Addison-Wesley, Reading, MA, 1992.

Stroustrup [1991] B Stroustrup. *The C++ Programming Language* (2nd edition), Addison-Wesley, Reading, MA, 1991.

Tenenbaum and Augenstein [1986] AM Tenenbaum and MJ Augenstein, *Data Structures Using Pascal* (2nd edition), Prentice-Hall, Englewood Cliffs, NJ, 1986.

Texel [1982] PP Texel. Ada_EDUCATION := DESIGN_CONCEPTS “+” Ada_CONSTRUCTS *Proc. 13th SIGCSE Symposium on Computer Science Education*, Indianapolis, 1982, pp 201–204.

Wang [1994] PS Wang. *C++ with Object-Oriented Programming*, PWS, Boston, MA, 1994.

Wiener and Pinson [1990]. RS Wiener and LJ Pinson. *The C++ Workbook*, Addison-Wesley, Reading, MA, 1990.