From Modula-2 to C++: Advanced Programming with Class

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

1. Introduction

At the Wits Computer Science Department, we have had a second-year topic called Advanced Programming for many years. Although the original language used for this was Pascal, for some time we have been using Modula-2.

While Modula-2 is a very good language for the basics of advanced data structures and algorithms, and for introducing software engineering, it lacks crucial features to support abstract data types and object-oriented programming. To address these deficiencies, we have moved to C++. We recognized that the additional material would make extra demands of students, so the topic has been lengthened by almost 50%, from 7 to 10 weeks.

The approach taken in the course is a relatively radical one. Instead of starting with the parts of C++ that most resemble Pascal, the course starts immediately with classes and objects, and introduces constructs such as loops in the context of extending existing classes.

Initial experience is that students whose first language is Pascal are confused by object-oriented programming, and introducing classes early has the benefit that they have the entire course to grasp the concept. This has the negative effect that they take longer to learn simple syntax (loops etc.).

This paper reports on the strategy taken in the course, as well as student responses to it. It also reports on the underlying educational methodology, and how this has been accepted by the students.

2. Background

The new course is a result of a curriculum workshop held in the Wits Computer Science department, in which we re-evaluated our course content in the light of published ACM and IEEE curricula [ACM 1991].

The methodology of this workshop was to identify subtopics within Computer Science, decide which year or years each should go into, and decide how many weeks should be spent on each. Once the subtopics were identified, we grouped them to form lecture topics, conforming as closely as possible to one topic per 7-week lecture block.

For the Computer Science 2 curriculum, we compromised on a neat block structure to allow more time for Data Abstraction and Algorithms (DAA), which is an extension of our old Advanced Programming topic, with the addition of more material on data abstraction and object-oriented programming. To fit the new focus, we also decided to switch from Modula-2 to C++.

One implication of the change of language was that we had to replace our existing equipment, as the old Macintosh SEs which had been adequate to run Modula-2 were not capable of supporting a reasonable C++ environment (the only way they could have been used would have been as telnet terminals to a UNIX system). We replaced the SEs by Power Macs; this gave the additional advantage of being able to move to a RISC architecture for our revised Computer Architecture topic, based on a modern text [Patterson and Hennessy 1994].

For completeness, outlines of the curricula of our revised second year course (with more detail of the Data Abstraction and Algorithms topic) are presented in an appendix.

Of relevance here are the subtopics covered (not in this order) in the DAA topic:

- Abstract Data Types
- Recursive Algorithms
- Complexity Analysis
- Sorting and Searching
- Problem-Solving Strategies
- Advanced Data Structures
- Object Oriented Programming
- Object Oriented Design and Analysis
- Scope and Binding

3. Course Strategy

When structured programming was the focus of most programming courses, it amused me that many programming texts started with the material that could just as well have been covered in FORTRAN—integer variables, no procedures, arithmetic statements, loops (even gotos in some earlier texts). Even many books which did advocate a structured programming style were late to go into procedures [Atkinson 1980].

To someone who grew up with FORTRAN, no doubt this seemed a natural approach, but in time, authors gradually realized that it was important to develop good programming habits early, and procedures and functions moved nearer to the front of the book [Bishop 1989]. If structured programming is the style being advocated, it makes sense to me that the necessary tools for program decomposition and abstraction be introduced early, before bad programming habits (global variables, writing large main programs) have developed.

Similarly, if implementing abstraction using object-oriented programming is the focus of a course, it makes sense to me to start with abstraction, show how existing classes can be used, and progress from there to defining your own classes. For this reason, the order of subtopics in my course is as follows:

- abstraction
- extending an abstraction
- implementing an abstraction
- abstract data types
- container classes and algorithm analysis
- object-oriented analysis and design

Here is more detail of each of these topics, each of which is a chapter of my notes.

3.1 abstraction

Abstraction is introduced in very general terms, including computer and non-computer examples. For example, use of real-world metaphors in direct-manipulation user interfaces is used to illustrate how abstraction can make a computer easier to use.

3.2 extending an abstraction

This is where C++ is introduced.

The purpose of this chapter is to introduce the concept of abstraction as a language mechanism, and to introduce the new concept of extending built-in abstractions. The rationale for introducing this early is that C++ allows extending abstraction by inheritance, which is a relatively difficult concept for the beginner—especially the beginner schooled in a procedural programming style.

Although object-oriented design is not introduced at this stage, Booch diagrams are used before C++ syntax is introduced.

Other key concepts of object-oriented programming are also introduced: encapsulation and information hiding.

Limited C++ syntax is introduced, mainly to illustrate class definitions for a specific example (hierarchy of shape classes). Also, the file structure of a C++ program is introduced.

The idea of an application framework is briefly introduced, as a way of avoiding re-implementation of common code every time an application is implemented, especially where there is a standard application architecture, as on the Macintosh.

3.3 implementing an abstraction

Implementation of an abstraction is now introduced, with more C++ syntax, including the structure of the main program.

More detail of the relationship between classes and objects is presented, along with an introduction to scope and binding in C++. Along with issues of the lifetime of an object, constructors and destructors are introduced.

More detail of extending an abstraction is introduced: a complete new class, `DoubleCircle`, derived from class `Circle`, but which replaces its `draw` member function by one which draws two circles with the same centre but differing radius, is defined.

This opportunity is used to introduce more C++ syntax, and the use of dynamic dispatch through virtual functions. At this stage, pointers are introduced, as they are essential to implement dynamic dispatch.

The idea of hiding implementation so that it can relatively easily be changed is used to relate abstraction to efficiency. This then becomes a starting point for introducing algorithm analysis. An example used to illustrate the principles is finding the convex hull of a set of points (the smallest convex polygon that encloses all the points).

Only once the major issues of abstraction and algorithm analysis are introduced, are C-like statements of C++ introduced: loops and selection. At this time, pointers are presented in a little more detail.

first programming assignment

Once the major aspects of abstraction and the language are introduced, a programming assignment is possible. This year, the assignment was the implementation of a naïve (inefficient) algorithm for the convex hull, which was mainly used to illustrate how to use existing code (list classes) and how to implement an algorithm using principles of abstraction.

The assignment was also used to introduce some programming strategies, such as using stubs before code is fully implemented, and using a driver program to test parts of the code out of the context of the final implementation.

3.4 abstract data types

This chapter introduces abstract data types, and makes a first pass at implementing common ones, such as queues, as classes. More detail of the language is introduced, including parameter passing, and the peculiarities of C-style arrays.

3.5 container classes and algorithm analysis

Here, more detail is presented of implementation strategies for container classes such as arrays and trees, especially techniques for generalizing containers without using unsafe type casts (a major problem in C++—unless you use templates, which have problems of their own).

This chapter also introduces multiple inheritance and mixin-style programming (construction of new classes out of mixtures of other classes—e.g., `IterableData` and `TableData` combined make a class that can be iterated over as well as stored in a table).

After an overview of container classes, the chapter introduces more detail of algorithm analysis, especially sorting and searching, in the context of explaining how to choose a container class. Several examples requiring solution of recurrence relations are introduced, though the more complex cases, such as the average case of quicksort, are not fully solved (to leave something for Honours).

The notion of optimal algorithms is introduced—without proof—to show that O($n$log$n$) is the best you can do if sorting by comparing pairs of keys. However, to show that the underlying assumption that sorting requires key comparisons is not always valid, non-comparing O(n) sorts such as bucket sort and radix sorts are presented.

Once sorting and searching are properly introduced, a better way of implementing a C++ container is demonstrated by an example of specializing a general tree to a tree of `StudentRecord`. Key aspects of this approach are the use of delegation to re-implement the interface to a class with new types, hiding unsafe type casts inside a class definition, and reimplementation of standard comparison operators for a user-defined class.

second programming assignment

At this stage the class is ready for a larger assignment. This year, the assignment was to implement an efficient picking algorithm (select the correct shape in a window, given a mouse click), given a simple window-based application framework.

The application framework was designed to make it easy to implement the graphical interface, without having to know anything about the Macintosh programming model. It included support for creating a window, creating menus and interpreting mouse clicks.

As before, the class was required to implement a separate driver program, this time to test algorithms independently of the graphical user interface.

3.6 object-oriented analysis and design

The final chapter introduces object-oriented analysis and design.

To set the scene, the classic waterfall model of the software life cycle is presented, followed by a short discussion of problems with this model. Then, object-oriented analysis and design are briefly introduced, followed by a detailed example based on this year's second programming assignment.

4. Equipment

This course was given for the first time in 1995, with mostly new equipment. The CodeWarrior C++ environment was used on Power Macintoshes. For a class of about 80, we had 26 Macs, which was adequate, as the class worked in groups of 4 to 5.

The Macs were on ethernet (with internet access), with an AppleShare file server, and an Apple LaserWriter IIf—the only old piece of equipment.

CodeWarrior—although difficult for the uninitiated—worked well for the course. Features of the Mac—fast RISC processor, windows, graphics, the ability to change a user interface by editing resources, built-in networking and file sharing, good quality sound—added excitement and interest to the course.

5. Tutorials, Assignments and Assessment

In the Wits Computer Science Department, for some years some of us have used assignment tests to assess assignments. This has a number of advantages. It makes group work easier to assess, it takes pressure off the class to add unnecessary frills to programs, it makes it easier to focus assignment work on specific skills (the aim of the test is advertised in advance), and there is less incentive to cheat on programming.

Since this is similar to assessment in our first year courses, the class had little problem with accepting this form of assessment. For each assignment, students had to have a sheet signed off by a tutor, itemizing the quality of their programming. This was a prerequisite for the test (which was the sole assessment for the assignment).

An innovation in this course—relative to the previous advanced programming course—was making each tutorial into a mini-test (called a "tut-test", each counting 0.5% of the overall mark for the year. Students were encouraged to ask questions in these tests, and to use them as an opportunity to develop their test-answering skills.

All tests and the final exam were open book. Although some members of the class felt that this made for more difficult questions, a large majority voted for this when asked if this was their preference.

6. Typical C++ Books

Unfortunately, I have not been able to find a single book which covers all of the material required—or in the correct order.

The nearest I have been able to find is the book by Decker and Hirshfield [1995], which not only covers much of the right material, but in much the order I prefer. Unfortunately, it is a bit weak on algorithm analysis, abstraction and object-oriented design. Also, it does not have any treatment of graphical user interfaces, which I feel reduces the excitement factor of using modern window-based computers.

Most other C++ books are lacking in one or more areas, or are specialized in a way not suited to this course.

Here are some examples of books which I rejected as prescribed texts:

- Wang [1994] starts at too low a level—classes are only introduced in Chapter 5, and algorithm analysis is not dealt with at all; doubtful features like `friends` are given too much space: more of a C++ book than a data abstraction and algorithms book
- Stroustrup [1991] again too much of a C++ book; worse for beginners than Wang
- Lippman [1991] is more accessible than Stroustrup, but is again better as a language reference than as a DAA text
- Sedgwick [1992] is a good general algorithms text, but is little more than a C++ translation of related books by the same author for other languages—it has little on object-oriented programming, object-oriented design or abstraction

Since the Decker and Hirshfield book arrived too late to be used for the course and found I no other book which came close to covering the right material, I wrote my own notes for the course, and referred the class to the above books where relevant, as well as to general texts on areas such as object-oriented design [Booch 1991] and algorithm analysis [Aho *et al.* 1983; Baase 1988; Weiss 1993]. If I had to run the course again, I would either extend my notes so that they covered as much ground as a full text book, or use Decker and Hirshfield, with my own notes to fill in some of the gaps.

7. Experience

The major change I would make if running the course again would be to use object-oriented designs throughout as my tool for describing examples and coursework. In this way, when software engineering was introduced at the end, the class would already know how a design was used, before being asked to do one themselves.

Another major change I would make would be to introduce my own application framework early in the course, instead of only bringing it in for the second assignment. I tried to introduce the concept using the PowerPlant framework that comes with the compiler, but a fully-functional framework is too large and complex for an introduction.

I would also organize classes I introduce into a library, which could be more easily reused. This would make introduction of container classes more natural.

I would also consider including more complex features of C++, especially templates and exceptions. I am still not convinced that templates are a stable and usable, given the problems I have had with other people's libraries. Exceptions, however are a useful tool, and can only be used properly if introduced at an early stage of a design.

The reaction of the class to the course changed from despair at the start, when they found themselves in the deep end without water wings, to appreciation of the range of new concepts they had learned. However, since this was a first run of the course, with teething troubles (e.g., working out which parts of the very extensive Code Warrior product were suitable for a course at this level), I did not do a formal survey of the class. I felt that it would be hard to isolate useful information from startup problems.

8. Conclusions

This course was a radical change from our previous course, so it was as successful as I could have hoped.Given the opportunity to run it again, it could be improved in detail and timing. However, I remain convinced of the rightness of the fundamentals:

- order of topics, in summary: abstraction, extending abstractions, implementing your own abstractions, algorithm analysis, software engineering
- micro-assessment through tut-tests; macro-assessment through assignment tests, a class test and the exam: the students this way have many opportunities to gauge their performance, and to build skills in question-answering

One thing of which I am *not* so convinced is that C++ is a great language—it carries too much baggage of C (arrays as constant pointers, the need to understand referencing and dereferencing at a detailed level, the lack of distinction between integer and boolean types, header files—to name a few problem areas).

However, given the need to compromise in the direction of skills wanted in the real world, it is hard to see which other object-oriented language could be used.

References

ACM 1991. A Summary of the ACM/IEEE-CS Joint Curriculum Task Force Report, *Comm. ACM*, vol. 34 no. 6 June 1991, pp 69–84.

AV Aho, JE Hopcroft and JD Ullman. *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

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

S Baase. *Computer Algorithms*, Addison-Wesley, Reading, MA, 1988.

J Bishop. *Pascal Precisely* (2nd edition), Addison-Wesley, Reading, MA, 1989.

G Booch. *Object Oriented Design with Applications*, Benjamin/Cummings, Redwood City, CA, 1991

R Decker and S Hirshfield. *The Objet Concept*, PWS, Boston, MA, 1995.

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

Patterson and JL Hennessy, *Computer Organization & Design: The Hardware / Software Interface*, Morgan Kaufmann, San Mateo, CA, 1994.

R Sedgwick. Algorithms in C++, Addison-Wesley, Reading, MA, 1992.

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

P Wang. C++ With Object-Oriented Programming, PWS, Boston, MA, 1994.

MA Weiss. *Data Structures and Algorithm Analysis in C*, Benjamin/Cummings, Redwood City, CA, 1993

Appendix
Curricula for New Wits Computer Science 2

Data Abstraction and Algorithms

This topic is primarily concerned with applying classic data abstraction and algorithms concepts using an object-oriented language and class library. Efficient use of modern programming tools requires a thorough understanding of abstraction at various levels, as a tool for managing detail. Object-oriented languages provide powerful facilities to support abstraction. This course aims to prepare students for understanding how such tools work by a series of exercises requiring increasing understanding of the underlying implementation. The course starts from making minor extensions to an existing program and progresses through to designing reusable code for others to use. Concepts covered include (not necessarily in this order):

- Abstract Data Types
- Recursive Algorithms
- Complexity Analysis
- Sorting and Searching
- Problem-Solving Strategies
- Advanced Data Structures
- Object Oriented Programming
- Object Oriented Design and Analysis
- Scope and Binding

Database

The major emphasis in this topic is on relational theory, methods and database management systems, using SQL on a UNIX system. This is a truncated version of a course previously offered in Computer Science 3.

Computer Architecture

This topic includes the study of modern microprocessor architectures, less intensive development of assembly language programming skills, than previously and achieving a good understanding of how programs are executed and what is the relationship between hardware and software. We have moved to a modern text on this subject, with emphasis on aspects of recent architectures including pipelines, registers, simple instruction sets and memory hierarchies [Patterson and Hennessy 1994].

Computer Systems

This topic provides background into two Operating Systems and Networks, and is similar to the course we previously offered in this area.

Mark Allocation

Topic	*Exam*	*Class*	*Topic weight*	*approx. weeks*
Data Abstraction & Algorithms	19%	19%	38%	10
Database	7%	5%	12%	3
Machine Organization	20%	10%	30%	7
Computer Systems	14%	6%	20%	7
Total	*60%*	*40%*	*100%*	