AAA and CS 1
The Applied Apprenticeship Approach to CS 1

Owen Astrachan
Duke University
ola@cs.duke.edu

David Reed
Dickinson College
reedd@dickinson.edu

(listed alphabetically)

Abstract

We have developed an application-based approach to introductory courses in computer science. This approach follows an *apprenticeship* model of learning, where students begin by reading, studying, and extending programs written by experienced and expert programmers. Applications play a central role since programming constructs are motivated and introduced in the context of applications, not the other way around as is the tradition in most texts and courses. Under our applied approach, (1) students are able to learn from interesting real-world examples, (2) the synthesis of different programming constructs is supported using incremental examples, and (3) good design is stressed via code and concept reuse. In this paper, we provide several examples of our method as well as pointers to all the material we have developed which is freely available electronically. The philosophy underlying this method transcends a particular programming language, but we present our examples using C++ since that is the language used in the CS 1 and CS 2 courses at Duke. This method has been used with equal success using ISETL at Dickinson.

1 Introduction

The introductory computer science courses, CS 1 and CS 2, present many challenges to the instructor. One need only look through any recent SIGCSE proceedings to see the wide-variety of approaches being used in these courses: depth vs. breadth; "traditional" vs. object-based vs. object-oriented; functional vs. procedural; and so on. At our institutions we have chosen to concentrate on a methodology that transcends a particular language or focus. Our *Applied Apprenticeship Approach* (we call the development of our method the AAA project) has three objectives.

Change Expectations: One of our aims is to instill a change in the expectations of both the students of these programming courses and the teachers of the courses. The traditional model in CS 1 and CS 2 courses is that students will develop proficiency in a specific computer language as well as "problem solving skills". Students are expected to hone these skills while designing and implementing programs. The view espoused in our project is explained in [Pat91] as "read/call before write" — students should be expected to read and modify programs before writing them. This apprenticeship model is similar to the notion of using a *case study* as described in [LC92, SLC93]. This apprenticeship model is even more important when the language used is C++, a language much too unwieldy to study and use in all its detail by students encountering programming for the first time. We believe that *designing* programs and classes is a difficult task, and one that students should not be expected to master after one course. The apprenticeship model ensures that by extending programs, and eventually developing them from scratch, good design skills are inculcated over time.

Change Focus: We want to change the focus of the kinds of programming problem encountered by students. Students coming out of a one or two course sequence in computer science often have no feel for what the discipline is about. An emphasis on programming toy problems teaches students that computer science is about semi-colons and choosing an appropriate looping construct. It indicates nothing of the power of computers, the interdisciplinary nature of computer science, the theoretical underpinnings of the discipline, and the aesthetics of programming. By using an apprenticeship approach in conjunction with an object-oriented language such as C++, students can construct interesting programs that illustrate the real power of computing. This

power is not examined in typical texts where averaging ten numbers is often the culminating experience of the array chapter. Our material provides a re-usable set of C++ classes that can be used in real-world applications from several disciplines. This approach facilitates a student view of computer science as both an elegant and useful science. Since we have adopted it enrollment in our introductory courses has significantly increased.

Change Delivery: We want to change the manner in which programming is taught, focusing more on applications as a means of introducing and motivating language constructs. Courses which use "small" languages, (i.e., languages with minimal syntax like Scheme), tend to be application driven, as exemplified by texts such as [AS85] and [HW94]. However, due to the amount of language details in procedural languages such as Pascal, Ada, and C/C++, traditional courses using these languages tend to be language driven. Glancing at the table of contents of almost any traditional CS 1 text illustrates this: chapters tend to be divided along the lines of programming constructs (first variables, then conditionals, then loops, and so on). This by no means implies that these are bad texts, but the discipline of computer science and the study of programming can appear fragmented if language constructs are the focus of a course or text. Our experience has shown that applications can successfully be used as a means of motivating language constructs and providing a context for their study. This approach is described in [BSK94] as "No topic before its time". The use of examples which can be incrementally developed is especially beneficial, since numerous language constructs can be presented in the same familiar context. For students this kind of concept reuse is as important as code reuse. As each new construct is added to the application, it can be understood in relation to the existing constructs. Incremental examples also encourage code reuse and stress good design, since a badly structured program will be difficult to adapt to a new situations.

A somewhat orthogonal objective of our project, but one we view as very important, is to develop support for courses in the form of assignments, teaching materials, and a software component library that can be used by students of all abilities. These materials should be useful in any college or university, without requiring a profound change in philosophy, a re-organization of courses, or a profound investment in new equipment. In other words, some of these materials should be useful at any institution. Electronic access to all existing course assignments, programs, and materials is made available via ftp and the World-Wide Web.

In the remainder of this paper we compare our AAA project with similar approaches and show several examples of the kinds of programs and classes we use in our courses.

2 Similar Approaches

Several NSF grants have resulted in curricula that may be indigenous to the institution supported by the grant. A curriculum developed at the University of Virginia [KPW94] is based on a four course sequence. Adopting material for one course may be difficult (but should be feasible) since the curriculum is designed as a comprehensive whole. Materials developed at SUNY Geneseo [SBK94, BSK94] require that a distinction be made between a programming course and a course covering details of computer science.

Our use of C++ and an object-based approach allows us to address the need for these skills in courses later in our curricula. Roberts [Rob93, Rob95] discusses using instructor-provided packages for the purpose of hiding the complexity of C in a first course. C++ is a much better solution since it is a (relatively) standardized language, provides call-by-reference, an I/O stream package that is extensible to user-defined objects, and the ability to use classes for encapsulation and data abstraction. As simple a feature as automatic calls of constructors/destructors can make student programming much simpler. At least one C++ text adopts this approach [Mer95].

The AAA project is based on a different philosophy of computer science. This holds that programming should be viewed as a means of linking together different branches of Computer Science and related disciplines. A quote from Hoare explains this philosophy quite well [Hoa89]:

> *Having surveyed the relationships of computer science with other disciplines, it remains to answer the basic questions: What is the central core of the subject? What is it that distinguishes it from the separate subjects with which it is related? What is the linking thread which gathers these disparate branches into a single discipline? My answer to these questions is simple — it is the art of programming a computer. It is the art of designing efficient and elegant methods of getting a computer to solve problems, theoretical or practical, small or large, simple or complex. It is the art of translating this design into an effective and accurate computer program. This is the art that must be mastered by a practising computer scientist; the skill that is sought by numerous advertisements in the general and technical press; the ability that must be fostered and developed by computer science courses in universities.*

3 Application Examples

In our courses, students are provided with C++ classes for implementing "real-world" applications. These serve

as the basis for several programming assignments including:

- A cardioverter-defibrillator (pacemaker) that simulates the monitoring of a heart and administers appropriate treatment when the heart beats irregularly. This program is adapted from one presented in [Pat91].
- A database program designed to keep track of multiple libraries of books and/or Compact Disks. In a recent class more than 3,000 entries were included in the database culled from an electronically administered student survey and from internet resources.
- Simulations of stochastic processes including examples drawn from the theory of random walks, queuing theory and the card game War (some of these examples are discussed in depth in [AB94]).
- Arbitrary precision integer arithmetic used, for example, in implementing certain cryptographic/encryption algorithms.
- A program to manipulate bit-mapped graphic images. This program involves elementary data compression as well as manipulations such as the inversion and enlargement of images.

Each of these assignments is an exemplar of the kind of student project envisioned as part of our approach. The assignments would not be approachable by students in a first course without the design, use, and re-use of C++ classes provided and studied throughout the course. The assignments are engaging as well as representative of real-world applications (especially from a student point-of-view).

3.1 Language Details: Control and Data

We do require students to master the rudiments of different control constructs and data types. However, rather than asking for a program that reads 10 numbers entered by the user and then extending this to reading strings entered by the user; we use the same idea but read from data files containing Shakespeare's plays. Students solve problems that require a computer: finding the average length of the words in Hamlet or, later in the course, the most frequently occurring word. The concepts used are the same as in a traditional "add ten numbers" program, but the application is more motivating. This approach is emblematic of the manner in which we use an application (counting words) to motivate the study of language details. Conceptual reuse occurs when we return to the program when studying arrays. Tracking all character or all word occurrences easily leads into sorting and searching providing a familiar context for studying these subjects.

4 An Early Example of Re-use

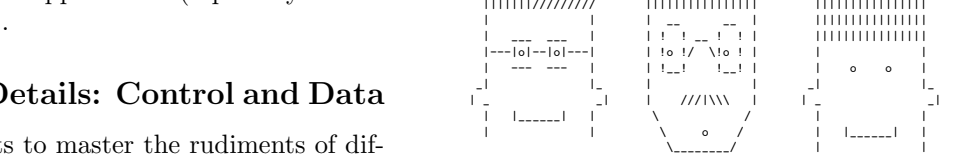
As an example of code and concept reuse, we begin with the program below early in the semester.

```
#include <iostream.h>

// print a head

main()
{
    cout << "  |||||||||||||||  " << endl;
    cout << "  |              |  " << endl;
    cout << "  |    o    o    |  " << endl;
    cout << " _|              |_ " << endl;
    cout << "|_                _|" << endl;
    cout << "  |   |______|   |  " << endl;
    cout << "  |              |  " << endl;

}
```

We have found that introducing user-defined functions early in the course is important in order to be able to use classes in C++. We have adopted a view from [Pat93].

> *It is disingenuous to extol the advantages of subprograms to our students, when they find it harder to write simple programs using subprograms — because they are not familiar with abstraction, but more importantly because using subprograms truly makes simple programs harder to write.*

Building on early programs that force students to use functions in order to print different verses of songs such as Old-Macdonald and Happy Birthday, we ask students to modify the head-drawing program above so that it can draw different heads as shown below.



If different functions such as `Hair`, `Eyes`, `Neck`, etc. are not used, this program becomes much more difficult. Students quickly see this and come to good design principles on their own.

We continue with this example later as a means of introducing control constructs. Students re-use the code from the program described above in the context of a rudimentary "police-sketch" program.

```
prompt> sketch
Choices of hair style follow

(1)  parted
(2)  brush cut
(3)  balding

enter choice: 1
Choices of eye style follow

(1)  beady eyed
(2)  wide eyed
(3)  wears glasses

enter choice: 3
Choices of mouth style follow
```

```
(1)  smiling
(2)  straight-faced
(3)  surprised

enter choice:  3
  |||||||/////////
  |               |
  |   ---   ---   |
  |---|o|--|o|---|
  |   ---   ---   |
 _|               |_
|_                 _|
  |       o       |
  |               |
```

Still later in the course we ask students to use a supplied graphics class/package to draw these figures using line and circle drawing primitives. This leads to a discussion of good class design since the police-sketch program can be re-implemented to use graphics most easily if the drawing routines have been encapsulated into a "head" class.

5 Database Example

To illustrate and motivate more advanced programming constructs, as well as concepts such as design, efficiency, searching and sorting, and object-oriented programming; we have successfully used the database example described in this section.

We begin this project each semester by requiring students to submit, via e-mail, a list of twenty books/authors or musical artists/CD titles. We incorporate these into one large file (with 100 students/semester in our CS 1 course this file grows rapidly). During the past semester, we incorporated data culled from the internet to add to this file.

5.1 The First Step

We begin by discussing how to store information about a collection of CDs. The type of data that must be stored (e.g., artist, title, music category) is determined, as well as the types of operations that might be performed (e.g., display a database, load/store the database from/to a text file, add/delete an item, search for a specific artist/title).

In the past, we have had students implement these operations using a traditional top-down approach, without using C++ classes. With this approach, the next assignment involves re-implementing the program using a DataBase class. In our current use of this assignment, we supply students with the definitions of the class at the outset and ask them to implement the member functions as well as extending a menu-driven client program.

5.2 Improvements

We come back to this program in discussing the tradeoffs of storing data in a fixed order. Structuring the data provides ample opportunities for design and efficiency discussions. For example, if listing all CDs by a given artist is a common task, it might make more sense to structure the database as a list of artist entries, with each entry storing all of the CDs by that artist. Given an array implementation, deleting a CD can be implemented in different ways, with various efficiency trade-offs. For example, array elements can be shifted to fill the space left by a deleted CD entry. Or, a special marker can be placed in the deleted entry location, so that the entry is not printed and can be reclaimed when a new CD is added.

We introduce searching and sorting in the context of this example as well. Students explore the tradeoffs of binary/sequential search and the need to sort or maintain the list in sorted order for different operations. The advantages of good design are especially apparent here, since the inclusion of sorting can be localized to those methods involved with adding and finding CDs. Because the database is large, efficient methods have immediate "tactile" results for students. Waiting for entries to be ordered using bubble-sort, for example, gives students a real feel for what slow means.

Sorting and comparing CD records leads to a discussion of operator overloading. Students re-implement comparisons of records by overloading the `<` and `==` relational operators for an individual CD record. This permits further optimizations to be localized: ignoring case in comparisons or, for an advanced topic, looking for "close" matches rather than exact matches.

5.3 Dynamic Data

After working with the various database programs, students begin to see the limits of a static array implementation. To avoid the problem of a full database, a natural extension (which is often suggested by students) is to use "grow-able" arrays. We supply such an array class (complete with array-bound checking) that uses a default array-size and, if that becomes full, simply allocates a new, larger array. Students must modify their code to be based on this new array implementation. The use of C++ makes this particularly simple since the indexing operator `[]` is overloaded for the new array class making the syntactic shift very simple.

An investigation of this dynamic array class leads to a discussion of pointers and dynamic allocation, but their introduction is eased by their presentation in a familiar context. This then leads seamlessly to a linked list implementation. Again, the clean separation between interface and implementation in object-oriented programming helps to localize the changes that must be made, and permits students to focus on concepts in a familiar setting.

6 Summary

Our experience in teaching CS 1 to both non-majors and majors is that focusing on applications eases the task of assimilating and understanding programming constructs and concepts. Students seem to appreciate interesting applications, and the incremental development of the applications allows for understanding each new construct in a familiar context. Returning to previous examples and modifying code also sends the right message to students concerning design: that a well-designed program can be more easily understood and re-used.

The examples presented in this paper are by no means definitive, nor is the choice of language particularly important. These examples were chosen because they have been effective in practice and they illustrate well the manner in which programming constructs can be motivated by applications, and not the other way around. All these materials, as well as other classes, assignments, and other supporting material are accessible via anonymous ftp from `cs.duke.edu` in the directory `pub/ola/apprentice` and via the World-Wide Web (e.g., via Mosaic) from `http://www.cs.duke.edu/~ola/apprentice`.

References

[AB94] Owen Astrachan and Claire Bono. Using simulation in an objects-early approach to CS1 and CS2. In *OOPSLA: Object Oriented Programming Systems, Languages, and Applications: Educator's Forum*, October 1994. Portland, Oregon.

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, McGraw Hill Book Company, 1985.

[BSK94] Doug Baldwin, Greg Scragg, and Hans Koomen. A three-fold introduction to computer science. In *The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, pages 290–294. ACM Press, March 1994. SIGCSE Bulletin V. 26 N 1.

[Hoa89] C.A.R. Hoare. *Essays in Computing Science*. Prentice-Hall, 1989. (editor) C.B. Jones.

[HW94] Brian Harvey and Matthew Wright. *Simply Scheme: Introducing Computer Science*. MIT Press, 1994.

[KPW94] John C. Knight, Jane C. Prey, and Wm. A. Wulf. Undergraduate computer science education: A new curriculum and overview. In *The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, pages 155–159. ACM Press, March 1994. SIGCSE Bulletin V. 26 N 1.

[LC92] Marcia C. Linn and Michael J. Clancy. The case for case studies of programming problems. *Communications of the ACM*, 35(3):121–132, March 1992.

[Mer95] Rick Mercer. *Computing Fundamentals with C++* Franklin, Beedle & Associates, 1995.

[Pat91] Richard E. Pattis. A Philosophy and Example of CS-1 Programming Projects. In *The Papers of the Twenty-first SIGCSE Technical Symposium on Computer Science Education*, pages 34–39. ACM Press, February 1991. SIGCSE Bulletin V. 23 N. 1.

[Pat93] Richard E. Pattis. The "procedures early" approach in CS1: A heresy. In *The Papers of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education*, pages 122–126. ACM Press, March 1993. SIGCSE Bulletin V. 25 N. 1.

[Rob93] Eric S. Roberts. Using C in CS1 evaluating the Stanford experience. In *The Papers of the Twenty-Fourth Technical Symposium on Computer Science Education*, pages 117–121. ACM Press, March 1993. SIGCSE Bulletin V. 25, N. 1.

[Rob95] Eric S. Roberts. *The Art and Science of C*. Addison-Wesley, 1995.

[SBK94] Greg Scragg, Doug Baldwin, and Hans Koomen. Computer science needs an insight-based curriculum. In *The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, pages 150–154. ACM Press, March 1994. SIGCSE Bulletin V. 26 N 1.

[SLC93] Patricia K. Schank, Marcia C. Linn, and Michael J. Clancy. Supporting Pascal programming with an on-line template library and case studies. *International Journal of Man-Machine Studies*, 38(6):1031–1048, June 1993.