4-1-1994

# The object-oriented course in data abstraction.

Joseph Bergin

# SCHOOL OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

**TECHNICAL REPORT**
Number 72, April 1994

# *The Object-Oriented Course in Data Abstraction*

Joseph Bergin

Department of Computer Science
Pace University
Pace Plaza
New York, NY 10038



PACE UNIVERSITY

**Joseph Bergin** is Professor of Computer Science at Pace University, with an office on the New York Campus. Professor Bergin holds a doctorate in mathematics from Michigan State University. His research interests are in programming languages, especially object-oriented languages and type systems.

Professor Bergin's central project during the past several years has been the writing of a textbook introducing data structures and algorithms by way of the object-oriented paradigm. The book, Data Abstraction: The Object Oriented Approach Using C++ was published by McGraw-Hill late last year. Currently, Professor Bergin is preparing similar materials in other languages, most importantly Modula-3. He has received support for this latter from the Digital Equipment Corporation. A possible course based on this book is described in the paper presented herein.

The present paper has been published several times in different versions. The first was at the SOOPPA (Symposium on Object-Oriented Programming Emphasizing Practical Applications) conference at Marist College in 1990. It was presented at OOPSLA (Conference on Object-Oriented Programming Systems, Languages, and Applications) at Vancouver in 1992. Most recently it appeared last year in the journal of Computer Science Education.

*"The Object-Oriented Course in Data Abstraction," from Computer Science Education (1993: volume 4, pages 63-76), is reprinted with permission form the Ablex Publishing Corporation.*

COMPUTER SCIENCE EDUCATION 4, 63–76, (1993)

# The Object-Oriented Course in Data Abstraction

**Joseph Bergin**
*Pace University*

This article describes the design of a course in data structures and data abstraction from the object-oriented standpoint. It discusses objectives and how they are met as well as course materials and strategies. This course has been offered to sophomores at Pace University since 1990. It does not depend on prior knowledge of object-oriented languages or tools and develops skills that may be used with or without object-oriented language support and thus fits into a standard curriculum. The course uses a large library of data structures, based loosely on the collection and magnitude classes of Smalltalk. Materials, including a manuscript for a textbook, have been developed in both C++ and Object Pascal.

## 1. INTRODUCTION

Data structures and the idea of data abstraction have been fundamental to computer science education since its early days. In Curriculum '68 [1] there was a separate course covering data structuring techniques and simple algorithms for their manipulation. Material covered included lists, trees and graphs as well as searching and sorting algorithms. A decade later, Curriculum '78 [2] included much of this material in the second course offered to undergraduate computer science majors, incorporating simple notions of efficiency, complexity, and correctness. More recently, the Task Force on the Core of Computer Science [4] has in-

---

A preliminary report on this project was presented at the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA) held at Marist College, Poughkeepsie, NY, in September 1990. More recently, it was presented at the Educator's Symposium at OOPSLA '92 in Vancouver, BC. The book on which this course is based is due to be published by McGraw-Hill in 1993.

Correspondence and requests for reprints should be sent to Joseph Bergin, Pace University, Computer Science Department, One Pace Plaza, New York, NY 10038.

cluded this material as the first of nine fundamental areas of computer science.

Examination of these course proposals shows a trend in computer science education of migrating topics, techniques, and ideas from advanced, graduate, and professional levels down to lower levels and even introductory levels. Program verification [5], for example, was once thought to be a very advanced topic suitable for graduate students. Today it is commonly introduced in the second course [6].

The data structures course fits early in the overall curriculum—immediately after students have learned the rudiments of small scale programming and have become familiar with the use of a language. Ideally, the course should prepare students for creating larger programs by emphasizing various abstraction that can be defined, built, tested, and used. The course described in this paper is an evolutionary development of the data structures course that hopes to strengthen students' development in the early semesters by introducing and emphasizing the ideas of abstraction and encapsulation through the use of object-oriented techniques. The course is evolutionary in the sense that it does not depend on object-oriented techniques in earlier courses, nor does it assume that students will necessarily be using the techniques in later courses.

## 2. COURSE OBJECTIVES

The following, though not a complete list, gives an idea of the thinking of the author in developing this course and its materials. It was believed that evolutionary change from existing courses, as exemplified in *Data Structures & Program Design* by Kruse [7], was better than a revolutionary approach, in that it would be more likely to find acceptance and be less risky for the students involved. If the curriculum is not to be made wholly object-oriented then the students will need techniques that they can transfer to other languages if they are not able to use them directly.

1. Introduce and reinforce the idea of abstraction, especially procedural abstraction and data abstraction.
2. Develop and emphasize interrelationships between data abstractions.
3. Modularize programs and emphasize data encapsulation and data hiding.
4. Give students the opportunity to see, use, and modify larger programs. Programs should be sufficiently large and complex that simple software engineering techniques are required.
5. Introduce, develop, and thoroughly integrate object-oriented pro-

gramming technology and style as a way to approach data abstraction.
6. Introduce fundamental data structuring techniques and algorithms.
7. Introduce and develop programming with "indirection."
8. Introduce and develop algorithm verification.
9. Introduce and develop algorithm analysis.
10. Introduce development of programs from specifications.
11. Introduce students to topics fundamental to computer science and its history, such as finite automata, grammars, data representation, etc.
12. Use a language that is small and easy to master. The purpose is not to teach a language.

## 3. CHOOSING AN OBJECT-ORIENTED LANGUAGE

Because the course emphasizes data encapsulation, an early decision was to employ object-oriented technology. The concept of *class* in object-oriented programming closely matches the concept of data abstraction with close binding between data structures and the procedures that modify them. Object-oriented programming also typically relies on a large library of classes that a programmer uses as a software tool kit. This aids in the quick development of larger programs and can be useful in meeting the objectives of this course since the students may see relationships from the structure of the library. They may also build parts of the library, resulting in a personal tool kit that they may use in other courses and projects. Also, the emphasis in object-oriented programming on data-centered design seems to match the main goals of this course quite well since the emphasis is on the structuring of data and on algorithms for its manipulation.

It was felt that merely employing an object-oriented style in a language like standard Pascal was not appropriate. Standard Pascal's chief drawback in the curriculum is that it already requires too much from the programmer's "style" to mitigate its drawbacks. For example, the lack of binding between a "stack" and the operations that manipulate it require artificial means (comments) to establish conceptual links for users. Requiring more in this regard seems to be a step backwards. Choice of language then becomes a problem. The obvious choice is Smalltalk, but this seemed to require changes in following courses that many instructors would not be willing to make. If the goal were to choose an object-oriented language for the entire curriculum (or most of it) then Smalltalk would be a good choice.

Languages that were considered for this project include Smalltalk,

Simula, Eiffel, C++ [7], and Object Pascal [8]. Other possible languages are Modula-3 and the latest Object Extensions to Pascal as they are being defined by the Joint Pascal Committee of ANSI/IEEE. With the hope that the complexity could be contained through use of a suitable subset of the language, the choice was made to develop materials in C++ and in Object Pascal. The book I have written on which the course is based [3] is built around a class library of approximately 90 base classes, with over a thousand methods, distributed into about 40 compilation units. This library was developed specifically for this course to emphasize the pedagogical objectives. I have begun the development of similar materials in Modula-3.

## 4. COURSE CONTENT

The data structures course must deal first with the ideas of data abstraction and second with the design and implementation of several common algorithms. Data abstraction provides the theoretical basis on which the data structures and algorithms stand. Typically, the data structures presented are stacks, queues and other linear structures as well as binary trees and some types of general trees. Some, but not all courses, will deal with various numeric representations (large integers, complex numbers, etc.). More ambitious courses will deal with simpler aspects of graphs and their representations. For all of these, various implementations are presented, and there is a discussion of the tradeoffs inherent in the various implementations. Algorithms typically taught are searching and sorting, including tree walks and binary search trees of various kinds.

In object-oriented programming the programmer normally has a large software library in source form at his or her disposal. Program development proceeds by examining the library (class hierarchy) for classes providing functionality similar to what is needed in the current application and either using them directly, or specializing the existing classes to create new ones using inheritance and compositional mechanisms. Much of the functionality is inherited from the classes already in place and does not need to be rewritten. A typical application consists of more than 75% existing code. A typical class library contains two rather different kinds of components. The first of these is "interface" components that implement the application's user interface. Generally this code consists of classes to create windows and menus, handle the keyboard and mouse, and handle common functions like printing and filing. The course does not consider the interface level of component.

The second major kind of functionality in the class library contains low level or "data level" constructs that are used across a wide range of

applications. These fall primarily into two categories, the collections, and the magnitudes. Collections consist of lists, stacks, queues, etc., and are responsible for managing aggregates of objects. Magnitudes include things that can be compared by "size," such as the numeric classes as well as associations, which are used as elements of dictionaries. The data structures course focuses on this latter level of functionality.

The application programmer using a class hierarchy will write "bridge" or "glue" code, to connect the "interface level" components to the "data level" components, and which implements the specific features of the application. While the professional programmer must certainly be an expert in the use of the interface level code to create an application, the content of this code is too rich in detail and too poor in intellectual content to spend much time with in a foundation course. (Note: The *design* of the user interfaces themselves does, of course, require a great deal of "intellectual content." But that is another issue for other courses.)

The key idea of the course is to examine the elements in the class hierarchy of Figure 1 (inheritance is shown using indentation), and to use objects of these classes in several moderately large applications of varying difficulty. Students must also implement key methods in most of these classes. For example, they must build the important insertion and deletion methods of lists, and the searching and sorting methods of lists and arrays.

Most of the source code for the class library of Figure 1 is made available to students. All of the interfaces of all of the units are made available in machine readable source form. All methods that are trivial to implement but needed for the correct operation of interrelationships in the library are also provided. Other methods that we expect students to build are given to students in "skeleton" form such as:

```
void SList::insert (PObject o){
        error("SList::insert. Student must implement");
};
```

The entire library is also available to the students in linkable-executable form so they may use the entire library before they have built parts of it. This makes it possible to do sophisticated examples early in the course that depend functionally on material developed later. In fact, it may be useful to present all of the library in source form, with the idea that here we have one implementation, and some assignments will be to improve it, and others will be to replace major parts of it with better implementations. Some instructors will prefer the latter approach. In giving hundreds of procedures to students in source form, students will

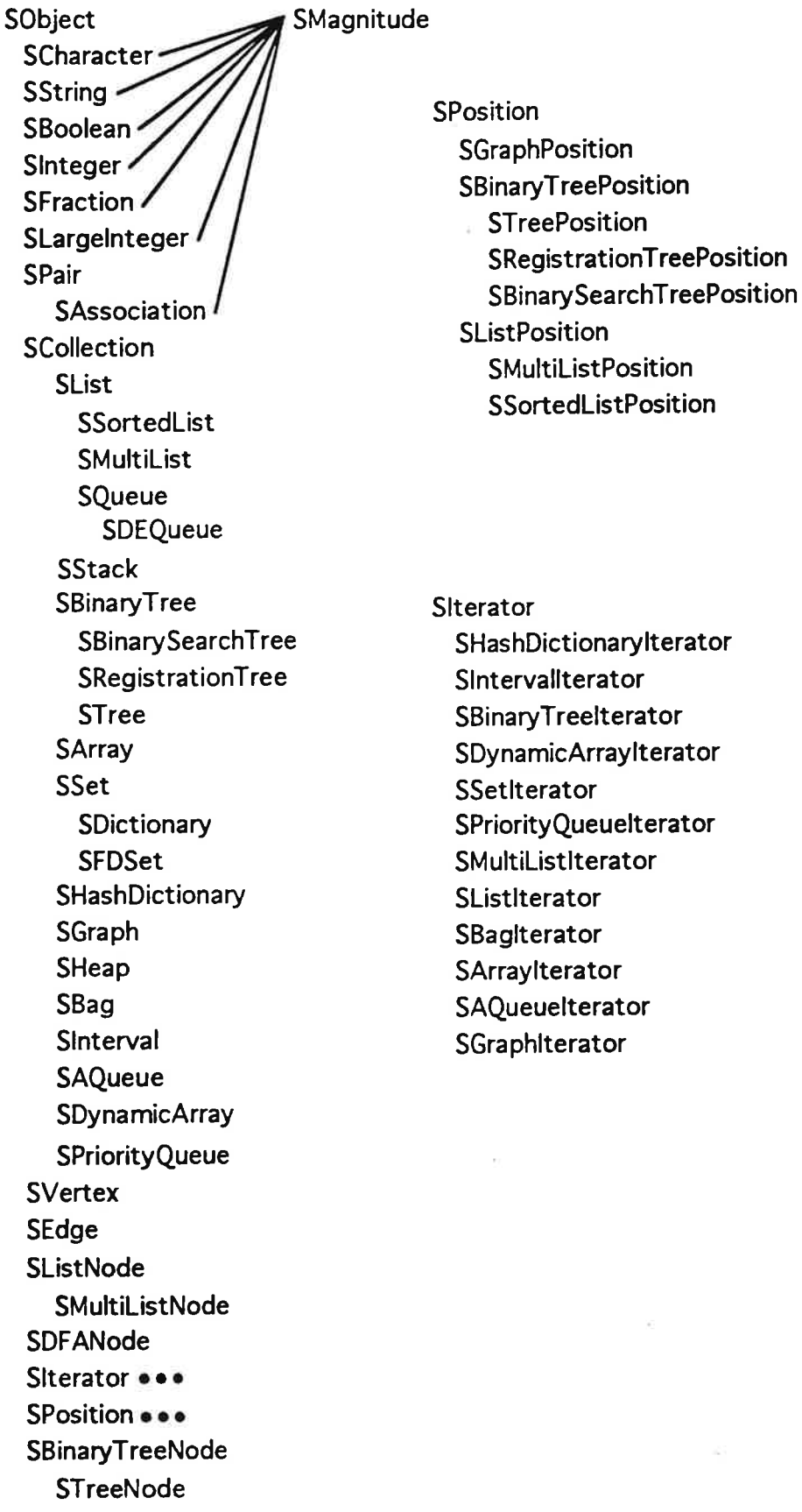


**Figure 1.** The Major Classes

see and use a larger quantity of software than they can be expected to write. The quantity is sufficient that structuring mechanisms are necessary for its understanding. The components of the library are not all handled to the same level of detail. Instead we focus on the linear and tree structures within collections as this is the area richest in algorithms. It is also necessary to deal with associations and some of the numeric classes within magnitudes to give balance (not everything is a collection) and to present fundamental ideas about internal data representation (of integers, Booleans . . .) and implementation needed at this level in the curriculum.

It is important in presenting this material that the instructor do more (much more) than present various implementations. It is necessary at all times to reinforce the ideas of software specification and the idea that if we can trust that software meets the specification it can be used without regard to the implementation. This encourages a decoupling of the ideas of a data structure from its implementation in the minds of the students. Therefore the course (and the book) proceeds from:

1. a general discussion of object-oriented programming, to
2. a general description of a specific class hierarchy, to
3. a specific discussion and description of the specifications of the classes, to
4. applications which use the above classes in important ways, and finally to
5. implementations of major elements of the classes.

Examples of applications that can exercise the class hierarchy are: finite automata simulators (using lists, stacks, and dictionaries), symbolic calculators (lists, trees), infix to postfix translators (stacks, strings, queues), and simple database or spreadsheet programs (lists, sets, numeric, and string classes). Applications such as these have the additional advantage of showing students some of the intellectual background of computer science, such as the Turing machine.

Two more complex applications are program parsers and verifiers (using trees) and database design normalizers (strings, sets). A program verifier is an especially rich area, in that it teaches much about several fundamental ideas of computer science as well as requiring solid data structuring mechanisms for its organization.

It is important to impart a notion early in the course that the software may be used once its specification is understood. This can be achieved through a discussion of the interfaces of several important classes. Figure 2 is an (unannotated) version of the interface for the list class and Figure 3 presents its associated position and iterator classes. Pre- and post-conditions should be developed in class for many of the methods, especially those like SList::insertFirst and SList::remove which modify

```
//REQUIREMENTS  For Use
//      initLists();

class    SList;
class    SListIterator;
class    SListPosition;

typedef SList                   * PList;
typedef SListIterator           * PListIterator;
typedef SListPosition           * PListPosition;

enum WhereFound {IsHere, IsNext, NotFound};

class SList: public SCollection {
    public:
                                SList(void);
        virtual                     ~SList(void);
        virtual PObject         first(void);
        virtual void            insert(PObject);
        virtual void            remove(PObject);
        virtual void            removeFirst(void);
        virtual PIterator           newIterator(void);
        virtual PListPosition newPosition(void);
        virtual char            empty(void);
        virtual char            element(PObject);
        virtual void            writeIt(void);

    protected:
        class SListNode;
        typedef SListNode *PListNode;

  // nested protected class SListNode
  class SListNode : public SObject {
    public:
                SListNode(PObject);
        virtual PObject         value(void);
        virtual void            writeIt(void);
        virtual PListNode           next(void);
  }; // end list node class

}; // end list class
```

**Figure 2.** The List Class

```
class SListPosition : public SPosition {
    public:
                        SListPosition (PList L);
        virtual void            next(void);
        virtual void            insertFirst(PObject);
        virtual void            insertAfter(PObject);
        virtual PObject         at(void);
        virtual void            atPut(PObject);
        virtual char            last(void);
        virtual void            toFirst();
        virtual void            deleteNext(void);
        virtual WhereFound              search(PObject);
};

class SListIterator: public SIterator {
    public:
                        SListIterator(PList L);
        virtual char            nextItem(PObject&);
        virtual void            reset(void);
};

void initLists(void);
```

**Figure 3.** The classes associated with lists (positions and iterators)

the storage structures. (Most of the nonpublic members have been omitted from the listing as have all of the comments.)

Only a small part of the interface needs to be presented initially. Note, however, that an attempt has been made to separate functionality into four classes: nodes hold references to data, a list is a linked list of nodes, positions refer abstractly to locations in the list (physically, to nodes), and iterators implement ways of applying some function to all elements of a list. The nodes, however, are implemented as a hidden nested class, which means that, to a client program, they are invisible. Thus, to a client, the list consists of a list of values, not nodes. Ideally, the implementation variables of the classes (not shown here) are not presented in the first cycle, so that the fact that we have a linked implementation is not obvious at first.

Once the specifications of the classes in the hierarchy have been discussed in some detail students may proceed to examine applications that use these classes. Figure 4 is a complete declaration of a deterministic finite automaton simulator class. The implementation uses the string,

```
PDFANode newDFANode(char isFinal);

class SDFANode: public SObject{
        public:
            SDFANode(char final = FALSE);
            virtual                 ~SDFANode(void);
            virtual char    member(classtype c);
            virtual void    writeIt(void);
            virtual void    addNeighbor(PDFANode nbr, PCharacter transition);
            virtual void    run(PString tape);
            virtual PObject Clone(void);
        protected:
            PDictionary fNeighbors;
            char fIsFinal;
            virtual int sizeOf(void);
};

//REQUIREMENTS  For Use
//      initLists();
//      initCharacters();
```

**Figure 4.** The Deterministic Finite Automaton Class

character, association, and dictionary classes. Implicitly, it also uses lists and sets.

This class implements a DFA as a distributed set of nodes. Each node maintains a dictionary containing the names and references to its neighbors; in effect, a projection of the transition function onto a node. A dictionary is a set of associations (pairs) and a set is built from a list. Importantly, we may use list and association abstractions before examining their implementations. The *run* method of a DFA object proceeds by examining the string it is sent; if it is not empty it strips the first character, looks in its (the object's) dictionary for a match, which, if found, causes the node to send a *run* message to the corresponding neighbor node, passing the remainder of the string.

Following development of a few such applications using the class library, students begin to examine the implementations of the classes and to develop parts of it themselves. Emphasis is on the development of general purpose, reusable, well-designed components, carefully implemented. It is here that one emphasizes abstraction, encapsulation, information hiding, representation, and the separation of specification and implementation. Emphasis is, of course, on those methods exhibiting interesting algorithms such as searching and sorting.

This is the most important segment of the course, dealing as it does

with the fundamental data structures and their implementation, and the algorithms necessary for their manipulation. It is here that object-oriented programming techniques have their greatest benefit. To use C++, it is necessary to design and write down a specification for a class before you write down an implementation. The specification is nicely encapsulated in a class declaration and made public by a compilation unit using a header file. One can then develop various implementations of the class and easily integrate them into software designed to use them. In the above example, a dictionary is a specialization of a set, which has a list member variable, which currently has a linked representation. As a class exercise, students might change the implementation of a dictionary or a set, link the changed classes to the above DFA unit and explore the differences in efficiency.

It is also necessary to teach the various logical relationships between abstractions and how they translate to relationships between classes. Students should learn about "isA", "isLike", "hasA," and "collaborates with" relationships. They should also learn that inheritance may be used to implement all but the last of these relationships, but with different degrees of success, especially in large projects.

## 5. CHALLENGES FOR THE OBJECT-ORIENTED COURSE

There are a number of difficulties in presenting these materials that must be creatively overcome. The first, and most important difficulty is that object-oriented programming is a different paradigm than students may be familiar with. It is important to specifically provide a mental model of programming that is useful in OOP. The model of autonomous actors interacting via messages, able to send and receive messages, and able to save and modify a purely local state has been very useful here. I have augmented this model with the idea of a processor being passed along with a message and the receiver using the processor to execute a method in fulfillment of the message, returning the processor along with any returned results to the sender. The instructor must be specific in defining programming as doing data decomposition of a system rather than procedural decomposition. In other words, the data decomposition comes first, with the procedural decomposition following.

The challenge comes from the relative complexity of C++. I address this complexity issue by building a library using only a subset of available features, thus reducing the learning requirements of the language allowing students to focus on the ideas of data abstraction and design. The library does not use reference variables (except reference parameters), operator overloading, tricky syntactical forms, or many of the operators of C++. It also focuses entirely on indirect (pointer to) objects. On the other hand, it makes extensive use of inheritance and

virtual functions, as well as a limited amount of class nesting and multiple inheritance (mixin classes). The library was developed with pedagogical rather than commercial goals as the primary consideration. It has been used successfully, however, in a number of applications.

The third issue concerns the overall methodology of creating a general purpose library of low level reusable service modules, which does not lend itself well to programming with arrays. The data structures course, however, traditionally discusses sorting and searching of arrays. In this library, the array class, which treats arrays as objects, overcomes this difficulty. The array class also shows how to treat arrays as expandable data structures. We may therefore treat all of the standard topics in an object-oriented framework.

Another difficulty, to some, will be the use of explicit pointer variables. Students will be programming extensively with pointers so as to obtain the benefits of polymorphism through virtual function. In standard imperative languages there is almost a prohibition against aliasing. This has arisen because of the difficulty of writing correct programs that contain aliases. In object-oriented programming aliasing is a fundamental technique. It is the basis of the collection classes. Two different "sets" can contain references to the same objects, rather than different objects with the same bit representation. The problems with aliasing don't go away however, as C++ does not have garbage collection. It is necessary, therefore to deal conceptually with the idea of object lifetimes, and the creation/disposal problem.

Some of the classes discussed are an attempt to overcome the fact that C++ is a hybrid language. It has well-designed objects, but it also has simpler constructs like integers, characters, and structs. We want to have a generic "stack" class in which we may hold data of any type. Therefore we declare stacks to be stacks of "objects" and implement a class of integers and one of characters so that we may stack these as well as more complex things. But then there are conceptual issues relating to the identity of the instances of such classes. For example, in the integer class there should be only a single object representing "six." Different references to an object "five" should all refer to the same object. This requires careful implementation and introduces a natural use of hash tables within the system itself. As such, the difficulty has been turned to an advantage as it shows an important use of one of the structures developed in the course. Another possible solution to this would be to use templates, but I have not done so to avoid a level of complexity in the class structure that is only needed in a few cases. This solution also makes heterogeneous collections natural.

Another difficulty that arises in the use of C++ is that the language does not give full support to interrelated data abstractions. If "friendship" (which opens the implementation of one class to another) were

inherited then this could be easily overcome, but, as it is, it is occasionally necessary to chose to implement a class as a subclass of another class to obtain visibility of certain implementation details, when it would be more natural to use an instance variable instead. An example occurs when trying to build a "bag" abstraction using a list. If bag doesn't inherit from list (logically it should not) but uses an instance variable of type list, then you have no access to the nodes out of which the list is built. Even if bag does inherit from list it is difficult to build a more suitable node class for bags and still get full access to it through the existing list code. Solutions to this in C++ are hard to find. One option is to make nodes public, but then clients may manipulate them. One option is to rewrite the list classes so that bags are friends. Another is to inherit from list and use protected functions of the parent classes for access. Finally, one may rebuild much of the list infrastructure to gain the added functionality. All of these have drawbacks. Unfortunately, the cleanest way is to rebuild, which entails redoing much of the work of positions and iterators as well as lists and nodes.

Finally, instructors should be aware that this course contains a large amount of material to cover. In fact, however, the time spent discussing the concepts of object-oriented programming and the structure of the library can be made up later due to the increased efficiency with which one can teach when students have a structure in which they may see the overall relationships, and also have a growing library of modular units on which they may build.

## 6. BENEFITS OF THE OBJECT-ORIENTED PARADIGM

The largest benefit of using object-oriented programming in the data structures course is that an object-oriented language like C++ naturally uses data encapsulation (into classes), which makes it easier to emphasize data abstraction. Such a language also permits separation of functionality into related classes. This was shown in the list class, in which the ideas of nodes, list, positions and list iterators are built in different classes, and their relationship emphasized by inclusion in the same unit.

The basic structure of this object-oriented approach can be realized in many other object-oriented languages, including Smalltalk, Simula, Object Pascal, Modula-3, and Eiffel. The structure of the class hierarchy makes clear the relationships between the collection types, so that they are not perceived as being unrelated to each other. It also reinforces the difference between design and implementation and the choices that must be made in each domain.

I have had a chance to examine the later work of students who have learned this approach. My experience is that students will use this

methodology later, even when programming in languages that do not support the object-oriented paradigm at all (e. g., Pascal). They seem to do a better job than others in decomposing their programs rationally. Their data structures are single purpose, with clearly defined interfaces, manipulated only through associated procedures. They partition a problem logically into compilation units that implement a single abstraction or set of related abstractions. They also tend to re-use more of their previous work than other students do. To achieve this, however, requires that the student truly understand the concepts of object-oriented programming. Those who see it as a mechanical approach to reuse, or as a simple addendum to "regular" programming, do not do so well.

## 7. CONCLUSION

It is time that we brought more powerful paradigms into the computer science curriculum. Object-oriented programming is a powerful method with advantages for learning as well as for efficient software construction. To be successful, it should be introduced at an early point where it will have the greatest benefit. And it must be completely adopted, and not treated as an "add on" to a traditional course. To be object-oriented, it must deal with issues beyond language, such as the class hierarchy. As an interim step, the data structures course seems to be a good choice for the introduction.

## REFERENCES

[1] W. Atchison, S. Conte, J. Hamblen, T. Hull, T. Keenan, W. Kehl, E. McCluskey, S. Navarro, W. Rheinboldt, E. Schweppe, W. Viavant, D. Young, "Curriculum '68," *ACM Curricula Recommendations for Computer Science*, Vol. 1, 1983.

[2] R. Austing, B. Barnes, D. Bonnette, G. Engel, G. Stokes, "Curriculum '78," *ACM Curricula Recommendations for Computer Science*, Vol. 1, 1983.

[3] J. Bergin, *Data Abstraction: The Object-Oriented Approach using C++*, to be published by McGraw-Hill in 1993.

[4] P. Denning, D. Comer, D. Gries, M. Mulder, A. Tucker, A. Turner, P. Young, "Computing as a discipline," *Commun. ACM* V32 (1), January 1989.

[5] M. Ellis, and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[6] D. Gries, *The Science of Programming*, Springer-Verlag, 1981.

[7] R. Kruse, *Data Structures & Program Design*, 2nd Ed. Prentice-Hall, 1987.

[8] L. Tesler, *"Object Pascal Report,"* Apple Computer, 1985.

**Drive by Wire**

Gas pedal/brake pedal/steering wheel are not directly connected to engine/brakes/ wheels, but to a computer. They are, in effect instructions to be interpreted by the program in the computer which in a few microseconds decides from all of its inputs, including yours, what is required and issues the commands it thinks best to the brakes or engine or steering mechanism, all of which are driven by simple electric motors connected by wires to the computer. The programs are written by expert programmers who create a few millions of instructions intended to anticipate your every driving need. If some aspect of the program doesn't appeal to you you are free of course to reprogram the computer (using C++, of course). Even your teen-age son will be able to completely reprogram the system to maximize performance and driving enjoyment.

Traditional (expensive and bulky) steering wheel can be replaced by a cheap and simple joy stick easily replaceable at your hobby center

Brakes no complex hydraulics, simple plugs/wires. If the brakes cease to function you can easily replace the actuator motors or the connecting wires yourself without going to an expensive mechanic.

The computer itself will be easily replaceable in case of the inevitable system crash. Computer will of course be designed to have a comfortable mean time between failures of 10000 hours.

The car itself will be moving through a sea of radar waves emitted by itself and other similarly equipped automobiles. This permits a positioning system that will keep your car separated from others to front, rear, and to the sides. These radar signals (potentially hundreds of separate simultaneous signals) are just one more input to the very complex on board computer system. They also explain why the computer must occasionally override the instructions from the human operator (driver). The computer may sense some hazard of which the driver is unaware and take evasive action on its own.

Extensive Windows™ displays/Flip a switch and you can control/ select options with joystick or mouse

Computer will run day and night, making it possible to program the heating/cooling system so that the interior will always be at a comfortable temperature.

Turning too tightly/ stopping too quickly. Computer may override your instructions and do what it thinks best to protect you.

For safety, of course, the kids won't be able to play their walkman's or those annoying video games, as the stray electro-magnetic fields from those devices could disrupt the on board electronics. However, the passenger compartment can easily be equipped with a multi-channel entertainment system individually programmed by the passengers and controlled by the CPU.

Note: If you find this appalling note two things. There really is a proposal to build such cars. Second, there already exist airliners (the Airbus and the next Boeing airliner) that are built on this principle. Ah, the arrogance of engineers--and others.

Joe Bergin