Crossroads
The ACM Student Magazine

Object-Orientation and C++

Part I of II

Objective Viewpoint

by G. Bowden Wise

C++ is just one of many programming languages in use today. Why are there so many languages? Why do new ones appear every few years? Programming languages have evolved to help programmers ease the transition from design to implementation.

The first programming languages were very dependent on the underlying machine architecture. Writing programs at this level of detail is very cumbersome. Just as hardware engineers learned how to build computer systems out of other components, language designers also realized that programs could be written at a much higher level, thereby shielding the programmer from the details of the underlying machine.

Why are there such a large number of high-level programming languages? There are languages for accessing large inventory databases, formatting financial reports, controlling robots on the factory floor, processing lists, controlling satellites in real time, simulating a nuclear reactor, predicting changing atmospheric conditions, playing chess, and drawing circuit boards. Each of these problems require different sets of data structures and algorithms. Programming languages are tools to help us solve problems. However, there is not one programming language that is best for every type of problem. New programming languages are often developed to provide better tools for solving a particular class of problems. Other languages are intended to be useful for a variety of problem domains and are more general purpose.

Each programming language imparts a particular programming style or design philosophy on its programmers. With the multitude of programming languages available today, a number of such design philosophies have emerged. These design philosophies, called programming paradigms, help us to think about problems and formulate solutions. In this issue, we begin the first part of a two-part series on the object-oriented paradigm and how C++ supports the paradigm. Before embarking on our journey into the object-oriented paradigm, we first take a look at how programming paradigms have shaped software design.

Software Design through Paradigms

When designing small computer programs or large software systems, we often have a mental model of the problem we are trying to solve. How do we devise a mental model of a software system? Programming paradigms offer many different ways of designing and thinking about software systems. A paradigm can be thought of as a mental model or as a framework for designing and describing a software system's structure. The model helps us think about and formulate solutions.

We can use the mental model of a paradigm independently from the programming language chosen for implementation. However, when the chosen language provides constructs and mechanisms that are similar to those that are found in the paradigm, the implementation will be more straightforward. Usually, there are several languages that belong to a paradigm. For this reason, a programming paradigm is also considered a class of languages.

A language does not have to fit into just one paradigm. More often, languages provide features or characteristics from several paradigms. Hybrid languages, such as C++, combine characteristics from two or more paradigms. C++ includes characteristics from the imperative and procedural paradigms -- just like its predecessor language, C -- and the object-oriented paradigm.

We will look briefly at the imperative, procedural, and object- oriented paradigms. Readers who wish to learn more about these paradigms and others may consult Sethi's survey of programming languages [5] or literature on multiparadigm computing [1] [7].

THE IMPERATIVE PARADIGM. The imperative paradigm is characterized by an abstract model of a computer with a large memory store. This is the classic von Neumann model of computer architecture. Computations, which consist of a sequence of commands, are stored as encodings within the store. Commands enable the machine to find solutions using assignment to modify the store, variables to read the store, arithmetic and logic to evaluate expressions, and conditional branching to control the flow of execution. Assembly languages and BASIC are example languages of this paradigm.

THE PROCEDURAL PARADIGM. The procedural paradigm includes the imperative paradigm, but extends it with an abstraction mechanism for generalizing commands and expressions into procedures. Parameters, which are essentially aliases for a portion of the store, were also introduced by this paradigm. Other features include iteration, recursion, and selection. The languages C, FORTRAN, and Pascal represent languages in this paradigm. Most mainstream programming today is done in a procedural language.

Designing systems in the procedural paradigm involves modeling the system as a set of algorithms (procedures) on a set of data values. Data values represent addresses in the store. Each procedure interacts with the store directly. Procedural languages represent an important evolution from imperative languages because they provide the capability and power of a von Neumann architecture independent from the underlying details of the machine.

The procedural paradigm was the first paradigm to introduce the notion of abstraction into program design. The purpose of abstraction in programming is to separate behavior from implementation. Procedures are a form of abstraction. The procedure performs some task or function. Other parts of the program call the procedure, knowing that it will perform the task correctly and efficiently, but without knowing exactly how the procedure is implemented.

Procedures were soon recognized as a powerful mechanism and paved the way for more forms of abstraction. One disadvantage of procedural abstraction is its loose coupling between the procedure and the data it manipulates. Language designers developed new forms of abstraction to provide better abstraction mechanisms for data.

THE PROCEDURAL PARADIGM WITH ADTs. *DATA ABSTRACTION* is concerned with separating the behavior of a data object from its representation or implementation. For example, a stack contains the operations Push, Pop, and IsEmpty. A stack object provides users with these operations, but does not reveal how the stack is actually implemented. The stack could be implemented using an array or a list. Users of the stack object do not care how the stack is implemented, only that it performs the above operations correctly and efficiently. Because the underlying implementation of the data object is hidden from its users, the implementation can easily be changed without affecting the programs that use it.

When we design algorithms, we often need a particular data type to use in order to carry out the algorithm's operations. The design of an algorithm is easier if we simply specify the data types of the variables, without worrying about how the actual data type is implemented. We describe the data type by its properties and operations and assume that whatever implementation is chosen, the operations will work correctly and efficiently. Types defined in this way are called ABSTRACT DATA TYPES (ADTs).

For example, suppose we are developing a simulation system to study an ecosystem. We need a data structure to hold events of interest and sort them by priority. We need the ability to insert events into the data structure and remove them in priority order. This particular abstract data type is called a priority queue. For the purpose of designing our simulation algorithm, we don't have to specify the types of the items in the queue, nor do we have to specify the data types of the priorities. We need only to assume that the priorities will be ordered.

The use of abstract data types makes the design of the algorithm more general, and allows us to concentrate on the algorithm at hand without getting bogged down in implementation details. After the algorithms have been designed, the actual data types will need to be implemented, along with the algorithms. Recently, procedural languages have been extended to support the definition of new data types and provide facilities for data abstraction. The languages CLU, Ada, and Modula-2 are examples of languages that provide abstract data type facilities.

THE OBJECT-ORIENTED PARADIGM. The object- oriented paradigm retains much of the characteristics of the procedural paradigm, since procedures are still the primary form for composing computations. However, rather than operate on abstract values, programs in the object-oriented paradigm operate on objects. An object is very similar to an abstract data type and contains data as well as procedures.

There are three primary characteristics of the object-oriented paradigm. We have already described the first, *ENCAPSULATION*, the mechanism for enforcing data abstraction. The second characteristic is *INHERITANCE*. Inheritance allows new objects to be created from existing, more general ones. The new object becomes a specialized version of the general object. New objects need only provide the methods or data that differ because of the specialization. When an object is created (or derived) from another object, it is said to inherit the methods and data of the parent object, and includes any new representations and new or revised methods added to it.

The third and final characteristic of object-oriented programming is *POLYMORPHISM*. Polymorphism allows many different types of objects to perform the same operation by responding to the same message. For example, we may have a collection of objects which can all perform a sort operation. However, we do not know what types of objects will be created until run-time. Object-oriented languages contain mechanisms for ensuring that each sort message is sent to the right object.

Encapsulation, inheritance, and polymorphism are considered the fundamental characteristics of object-oriented programming and all object-oriented languages must provide these characteristics in some way. Not surprisingly, languages support these characteristics in very different ways. Smalltalk, C++, Objective-C, and Lisp with CLOS (the Common Lisp Object System) are all examples of object-oriented languages, and each provide support for encapsulation, inheritance, and polymorphism.

Constructing an object-oriented program involves determining the objects that are needed to solve the problem. The objects are then used to construct computations that define the behavior of the software system. Message passing is the fundamental interaction mechanism among objects. Messages (from other objects or programs) are sent to objects to inform them to perform one of their operations.

Objects are responsible for maintaining the state of their data. Only the object may modify its internal data. Objects may themselves be implemented via other sub-objects. Implementing an object involves a recursive process of breaking it into sub-objects until at some level the objects and methods defined on them are primitives. At this point, the methods and data consist of elements that can be implemented using the basic constructs pro- vided by the programming language.

One of the most important aspects of the object-oriented paradigm is how it changes our way of thinking about software systems. Systems are thought of as consisting of individual entities that are responsible for carrying out their own operations. Each object is conceived and implemented as self-contained. Such a model facilitates software design (and later implementation) because objects often model conceptual real-world entities. Designing systems using the object-oriented paradigm results in software systems that behave and appear more like their real-life counterparts.

Object-Oriented Design in Brief

Programming with objects has resulted in a whole new way of thinking about and designing software systems. New techniques for designing systems within the object-oriented paradigm have surfaced. The basics of object-oriented design are presented below. You may wish to design your next project using these techniques. Just as the procedural paradigm resulted in many different ways to design systems using structured programming techniques, there are many different design philosophies for object-oriented design as well. Interested readers may wish to consult books by Coad and Yourdon [3,6], Booch [2], or Rumbaugh [4].

To effectively design a software system, we must have a deep understanding of the software and its inner-workings. This is true no matter which design philosophy we choose. This intricate knowledge is even more important when designing an object-oriented system. Well designed objects and class hierarchies requires that the relationships among different objects be well understood.

The development process of object-oriented design is very different from the traditional top-down or structured design techniques. There is more emphasis on the actual real-world entities that the software system intends to emulate. In addition, object-oriented techniques enable software to be developed in an iterative cycle allowing prototype systems to be developed and tested early on. This is a big advantage over traditional techniques, where the first prototype does not originate until much later in the development process.

The subject of object-oriented design deserves an entire issue devoted to its intricate details, or even an entire book for that matter! The following list serves as a brief overview of the activities of object-oriented design:

- Identify the real-world entities in the application domain.
- Identify barriers of the entities.
- Identify relationships between entities.
- Create a C++ design structure and class hierarchy from the entities.

- Implement the classes and the system.
- Fine-tune the code for performance.
- Test the system.

The above activities do not have to be done in a sequential fashion, and most often they are not. The activities are performed in a back-and-forth manner, sometimes in parallel, and often in a number of iterations. The key to the object-orientation is to iterate, create prototypical implementations, test, and remove mistakes. Many times, testing identifies areas that need to be redesigned. Because classes are self-contained entities it is much easier to make a change to an implementation with minimal impact to the rest of the system.

The Object-Oriented Characteristics of C++

Our focus for the remainder of this and the next issue of the column will be to look at how C++ provides support for the fundamental characteristics of the object-oriented paradigm: encapsulation, inheritance, and polymorphism. C++ introduces the class mechanism which not only provides a facility for designing and implementing abstract data types, but also provides the basis for object-orientation.

ENCAPSULATION in C++. C++ extends C with a facility for defining new data types. A class is like a C struct, but contains data as well as methods. In addition, C++ provides different levels of access to the members of a class in order to control how the members of a class can be manipulated from outside the class.

Recall that the importance of data abstraction is to hide the implementation details of a data object from the user. The user only accesses the object through its PUBLIC INTERFACE. A C++ class consists of a public and private part. The public part provides the interface to the users of the class, while the private part can only be used by the functions that make up the class.

C++ provides keywords to indicate which members of a class are hidden and which are part of its public interface. The members of the hidden implementation are marked in sections beginning with the keyword private. The public interface part of the class follows the keyword public. By default, the declarations within a class are private, meaning that only the member functions (and friends) of the class have access to them.

Let's take a look at an actual C++ class definition. **Figure 1** shows the class definition for the IArray class. We see that the data members DefaultSize, size, and the member function Init() are private and are therefore inaccessible to user pro- grams. There are only member functions within the public sec- tion of the IArray class definition. These functions consist of a default constructor; a copy constructor; a destructor; assign- ment, subscript, and output operators; and a function that returns the size of the array.

Figure 2 contains the implementation of the IArray class and the program in **Figure 3** illustrates the use of the IArray class. **Figure 4** shows the output from the program.

A class definition does not allocate any memory. Memory is allocated when an array object is created through a variable declaration. Constructors and destructors provide the initialization and clean up of an object. When an object is declared, the constructor is called to initialize the memory used by the object. The destructor performs any clean-up for the object when the object goes out of scope and is destroyed. Looking at the implementation of the IArray class in **Figure 2**, we see that the constructor allocates memory for the data array, while the destructor frees that memory.

Note that we didn't really hide the implementation details from the user. C++ does not provide a way to completely exclude all of the details of the underlying implementation, since the private part of the class must be included with the class definition it is useful to relax the access to variables within a class, particularly under inheritance. Often derived classes need easy access to the private members of their parent classes. C++ defines the keyword protected for this purpose. Protected members can be accessed by the member functions of a class as well as by member functions of derived classes. However, like private members, protected members cannot be accessed by user programs.

One final note about objects. Recall that message passing is the fundamental means for communication among objects. When we write i < a2.Size() we are effectively sending a message to the a2 array object to determine the size of the array and return it. In actuality, no message is really sent. C++ emulates message passing through the use of function calls. The compiler ensures us that the correct function will be called for the desired object. So, in C++ you can think of message passing as function calls.

Closing Remarks

Object-orientation is a truly powerful and elegant way of developing software systems. Languages such as C++ make it possible to develop systems using the object-oriented paradigm, making it possible to design our software while maintaining a real world view of the system. Those of you who are looking for some good books on C++ may wish to take a look at the side-bar entitled To Probe Further: A Guide to C++ Books. In the next issue we will look at how C++ provides support for the two remaining characteristics of the object-oriented paradigm: inheritance and polymorphism.

Send In Your Questions!

I would like to start a question and answer grab bag beginning with the next issue of ObjectiveViewPoint. Have you encountered any surprising or baffling results when developing your own C++ programs? Are there any topics related to C++ that you wish to see discussed here? Please send your questions and comments to wiseb@cs.rpi.edu. I will do my best to provide feedback to all submissions.

References

1.
Ambler, A. L., Burnett, M. M., and Zimmerman, B. A. Operational versus Definitional: A Perspective on Programming Paradigms. IEEE Computer, 25, 9 (Sep. 1992) 28-42.

2.
Booch, G. Object-Oriented Analysis and Design with Applications. Benjamin/Cummings Pub. Co. 1994.

3.
Coad, P. and Yordon, E. Object-Oriented Design., Yourdon Press. 1991.

4.
Rumbaugh, J. Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, N.J. 1991.

5.
Sethi, R. Programing Languages: Concepts and Constructs. Addison-Wesley, Reading, Mass. 1989.

6.
Yourdon, E. Object-Oriented System Design: An Integrated Approach. Yourdon Press. 1994.

7.
Zave, P. Compositional Approach to Multiparadigm Programming. IEEE Software (Sep. 1989), 15-25. Object-Orientation and C++

To Probe Further: A Guide to C++ Books

As you learn C++ you will undoubtedly begin to create a collection of books on C++. Below are some books that you may wish to add to your library.

Primers

The following are good books for learning C++ or as general reference books:

- Graham, N. Learning C++. McGraw Hill, Inc., New York, N.Y., 1991.
- Lippman, S.B. A C++ Primer. Second Edition. Addison-Wesley, Reading, Mass., 1992.

Advanced Books

C++ is a very powerful and large language. Once you have learned the basics and have written a few programs you may want to learn more about the many subtle features of the language. The following books will take you on the path to becoming a C ++ guru:

- Coplien, J.O. Advanced C++ Programming Styles and Idioms. Addison-Wesley, Reading, Mass., 1992.
- Murray, R.B. C++ Strategies and Tactics. Addison-Wesley, Reading, Mass., 1993.
- Myers, B. Effective C++. Addison-Wesley, Reading, Mass., 1992.

Language Reference

C++ is still a relatively new language and is still undergoing developments, particularly in the areas of run-time type checking (RTTI) and templates. Most compilers have not implemented RTTI and templates completely. The ANSI/ISO committee responsible for the C++ language standard has published an excellent book on the many implementation details for C++ compiler developers. The ARM, as it is commonly called, provides a perspective of the history of C++:

- Ellis, M.A. and Stroustrup, B. The Annotated C++ Reference Manual. Addison-Wesley, Reading, Mass., 1990.