David Jordan

Implementation Benefits of C++ Language Mechanisms

C++ was designed by Bjarne Stroustrup at AT&T Bell Laboratories in the early 1980s as an extension to the C language, providing data abstraction and object-oriented programming facilities. C++ provides a natural syntactic extension to C, incorporating the *class* construct from Simula. A design principle was to remain compatible and comparable with C in terms of syntax, performance and portability. Another goal was to define an object-oriented language that significantly increased the amount of static type checking provided, with user-defined types (classes) and built-in types being part of a single unified type system obeying identical scope, allocation and naming rules. These aims have been achieved, providing some underlying reasons why C++ has become so prevalent in the industry. The approach has allowed a straightforward evolution from existing C-based applications to the new facilities offered by C++, providing an easy transition for both software systems and programmers. The facilities described are based on Release 2.0 of the language, the version on which the ANSI and ISO standardization of C++ is being based.

Paradigms

C++ supports the object paradigm but does not enforce it; it is a multi-paradigm language. Programmers who have primarily used procedural paradigm decomposition techniques can migrate to the language at a comfortable pace and still achieve many benefits of the language. Many can use predefined object-oriented C++ libraries, incurring only a minimal learning curve. Because of the syntactical similarity to C, this is natural and does not result in software appearing as a mixed collection of programming styles. Thus C++ has become a major vehicle for the migration from traditional "procedure-oriented" programming and design techniques to data abstraction and object-oriented programming.

C++ As A Better C

As an extension to C, C++ supports the basic data types in C; not all data types are of an "object type." C++ is often initially used "as a better C," taking advantage of the strong type checking facilities (some of which were later incorporated into the ANSI C standard).

The name and argument types of a function are included in its *signature*, allowing function name overloading. This allows a mnemonic name to be reused in all appropriate contexts, reducing namespace clutter and the need to devise unique and often cryptic names. The likelihood of name collisions is reduced when separately designed software is integrated into an application. A function prototype specification must precede the use of a function so the compiler can do argument type checking.

C++ also allows programmers to define functions as *inline*, allowing the code of the function to be expanded where it is called, eliminating procedure call overhead. Inline functions provide the benefits of macros in C, but go further by providing argument type checking. Sometimes inline functions can result in both a reduction of code size and an increase in execution speed. But inlining is not a panacea and can be overused, sometimes resulting in significant code expansion.

The *reference* type in C++ provides call by value syntax, but with the efficiency and operational characteristics of passing an argument by address. A reference must be initialized when declared and cannot be changed, providing some protection against improper uses of pointers. A reference is an *lvalue*, allowing functions that return a reference to be used on the left-hand side of an assignment.

The *const* specifier in C++ allows data/objects to be defined as read-only. This allows programmers to provide constants of any data type and ensure that parame-

ters passed to a function by address or reference are not changed by the called function.

Defining New Types

The *class* construct in C++ allows programmers to define new data types completely, so that they operate as if they were directly supported in the language. Objects are instances of these classes; members of a class can be functions or data. One can fully specify the functionality of the type mechanisms in the language, allowing new types to have all the expressive capabilities possessed by the built-in types borrowed from C. This includes specifying operators to handle assignment, initialization and type conversions, for example.

Nearly all the built-in operators of C can be associated with functions whose operands include user-defined types; arithmetic, comparison, logical, dereferencing and subscripting operators, for example. As a syntactical convenience, operator symbols like +, <, &, ==, ->, and [] are treated as functions whose names happen to be the standard operators found in many programming languages. Operator overloading should be used where the semantics normally attributed to the operator apply intuitively for the types of the operands. C has predefined semantics for use of operators with pointer operands. References can be used with operator overloading to pass the addresses of objects to the function implementing an operator, which is often more efficient than passing objects by value.

Object Instantiation

Constructors are functions that can be defined for a class to ensure proper initialization of an object when it is instantiated. A constructor called the *copy constructor* is used to make a copy of an object. This constructor is used when an object is passed by value as a function argument or return value. Some object-oriented languages do not pass objects by value, but always use a reference. Specifying function arguments as references or pointers in C++ is recommended to reduce the often unnecessary overhead of instantiating a new object. Constructors are also used in type conversions. A *destructor* is a function that can be defined to provide necessary cleanup when an object is deleted.

Objects in C++ can be instantiated either implicitly or explicitly. C++, like C, is a block-structured language and objects local to a block are implicitly instantiated/deleted when a program enters/exits a block. The extent of an object is either *static, automatic* or *dynamic.* Static objects have their constructors called automatically when a process starts and the destructors are then implicitly called when the process terminates; these objects reside in the data segment of a program. Automatic objects reside on the stack and are the local variables within a function. As with static objects, the constructors and destructors are called implicitly for automatic objects on entry and exit from a function (or a block within a function).

Dynamic objects are explicitly instantiated by invoking the *new* operator and are destroyed by invoking the *delete* operator. As a default, the standard memory allocation and deallocation facilities of the operating system environment are used. Programmers can also provide their own memory allocation primitives by overriding operator new and delete; this can be used to place objects in shared memory, for example. It also provides a means of transparently taking control of the allocation of objects of a known size and using techniques far more efficient than possible with the system's standard memory allocation facilities that must handle objects of any size.

C++ does not provide implicit dynamic allocation of objects, nor is there mandated support for garbage collection to free memory for objects no longer accessible. Many of the languages that use garbage collection by default incur performance penalties. If garbage collection mechanisms are needed, they can still be developed and used on a per-class basis and kept transparent to users of the class.

Inheritance

C++ supports both single and multiple inheritance. All classes do not have to be derived from a single root class; there can be as many independent class lattices as required. C++ supports class inheritance; it does not support object-level inheritance.

Multiple inheritance can result in a base class occurring more than once in a derivation. A single occurrence of a base class can be obtained by declaring it as *virtual.* Multiple inheritance can also result in a member function with a given signature being inherited from more than one base class. These clashes are detected at compile time and the ambiguity can be resolved by redefining the member function in the derived class.

Polymorphism

Polymorphism is a key benefit offered by object-oriented languages. It provides software a generic interface defined by a base class so objects can be manipulated uniformly though they may be instances of either the base class or any derived class. Dynamic binding is a mechanism used to support this. New derived classes can be defined and easily incorporated into a system; objects of those classes are transparently manipulated by software interfacing at the generic base class level. *Virtual* functions provide this capability in C++. The function actually called depends on the class of the object used when invoking the virtual function.

While some object-oriented languages are implemented such that all functions are virtual, in C++ the programmer can make the choice. If the designer of a class does not want to permit a function to be redefined in a derived class, the function can be specified without the virtual keyword.

One can define an *abstract base class* in C++ by defining a set of virtual functions as having null values; the antithesis of an abstract class is a concrete class in which all functions have been specified. Abstract classes can exist

at multiple levels in a class hierarchy. The compiler only allows instantiations of concrete classes. Abstract classes provide a uniform and transparent interface to a set of semantically related derived classes, ensuring the derived classes provide a base set of functionality.

Member Access Control

C++ provides several levels of access control to the members of a class; this includes both functions and data. Members can be *private,* such that they are only accessible by functions that are members of the class. Members of a class can be declared as *public,* allowing all functions access to the members. Members can be specified as being *protected,* only allowing access by member functions of the class and any derived classes. This is useful for providing access to members needed by derived classes, but preventing access by other functions. The accessibility of base classes can be specified as either public or private.

A class can also specify *friend* functions that are allowed access to private and protected members. It is also possible to specify that all the member functions of another class are friends by declaring the class as a friend. This is useful when several classes are tightly related, each providing the definition of a component used in the facility being designed; the nodes of a linked-list class, for example.

Control of member accessibility can also be used to prevent use of some of the operators that provide the type mechanics in the language. For example, the copy constructor and the assignment operator could be restricted to a subset of functions by declaring them as private or protected. Defining operator new as a private member of a class could be used to restrict the dynamic allocation of objects of the class to those functions that are either friends or members.

Other Class Facilities

It has been mentioned that C++ supports constant data; this applies also to class instances. Class member functions that do not change the value of the object can be specified as *const.* These are the only member functions that the compiler will permit to be invoked with constant objects of the class. The compiler also ensures that those functions do not change the value of the object. This is useful information for users of a class, clearly indicating those member functions that can or cannot change the value of an object. The const specifier should be used for both data and functions wherever possible; otherwise it precludes users from having constant data.

A class can have members that are specific to the class by declaring them as *static.* Static data members have class scope and only one occurrence is shared by all instances of the class. While C++ does not provide meta-class facilities, static data members can be used to provide some of the information commonly associated with a meta-class or class object. One can also define static member functions that can be invoked with or without an instance of the class.

Development Environment

The C++ compilation environment results in C++ source being compiled into the object code of a particular machine. The current language implementation distributed by AT&T is a translator; the C++ source is translated into C and then compiled by a C compiler. The translator, in a sense, treats C as a universal assembly language, which is in many ways true. C runs on virtually every machine in the industry and this translation approach has allowed C++ to be quickly bootstrapped onto many machine architectures. Another benefit of this approach is the ease of using C cross-compilers to build software for target machines architecturally different from the development machine. There is an industry trend toward RISC processors to increase performance and since most RISC processors have instruction sets specifically designed to efficiently execute C, the C++ translator is able to cost-effectively take advantage of this industry trend with minimal development cost.

C++ is link-compatible with C and any other language with which C code can be linked: Fortran, for example. A linkage specification mechanism has been defined so that C++ implementations can provide link compatibility with other languages. This permits C++ software to be readily incorporated into existing software environments without requiring rewriting millions of lines of tested, working software. Even if a software development organization wants to rewrite its software to take advantage of the advanced features, this linking capability allows an incremental migration path. This allows reuse not only of new code designed specifically using the object-oriented facilities provided by C++, but also reuse of the massive amount of useful code available in C, Fortran, etc.

Class implementations are usually placed in libraries. A program using a class will only link in those object files that are explicitly specified or are implicitly linked because they are needed by an object file that is included in the program. This would include all the object files containing the virtual function definitions for those classes used by an application (this is a result of the technique commonly used for implementing the virtual function mechanism). Operating system environments supporting dynamically linked libraries would only link in object files when they are needed. Functions can be separated into as many object files as necessary to minimize the inclusion of unneeded functions. These aspects of the compilation environment permit C++ programs to fit easily into small machine architectures.

Initially there were no debuggers that provided C++-level debugging; programmers used C-level debuggers. With Release 2.0 of the translator, routines are provided that understand the name encoding algorithms used by the translator. These routines are being incorporated into existing C debuggers, permitting the debugger to provide a more complete C++ debugging environment. There also exists a utility that can be run against a program to rebuild its symbol table, replacing

the encoded C names with their C++ representation. C++ compilers directly provide C++-specific symbol table representations.

C++ can be obtained from multiple sources and several companies are marketing both C++ translators and compilers. Some of the compiler implementations include syntax-directed editors, incremental compilation and integrated debugging facilities. Interpreters for C++ are also being developed, though none are commercially available yet.

Performance Advantages

C++ allows generation of very efficient code when invoking member functions. Non-virtual function calls are completely resolved at compilation. Invoking them at runtime is just as efficient as a function call in C, with none of the lookup overhead characteristic of many object-oriented languages.

C++ implementations also have very efficient mechanisms for calling virtual functions. Each object of a class with virtual functions has a pointer to a jump vector that exists for each class. The jump vector contains the addresses of all the virtual functions. Each virtual function is assigned an entry in the jump vector during compilation. When a virtual function is called, the jump vector pointer in the object is used with the virtual function's index in the jump vector to determine the appropriate function to call.

Some object-oriented languages delay the computations needed to determine the proper function to call until runtime. While this approach does provide a higher degree of flexibility, it results in slower execution and delays the detection of many errors until runtime (the lack of a function being specified, for example). C++ does not have this characteristic and thus provides excellent performance and a guarantee of the construction of complete programs.

The class definition includes the data members used in the implementation of the class. One side effect is that during development of a class, changes to its declaration require recompilation of all files using the class. But because the memory layout of an object is known at compile time, significant optimizations can be made. The compiler can make effective use of the stack for automatic variables and accessible data members can be directly addressed without incurring any overhead. Class implementors can provide inline member functions that insulate the class user from the internal representation and still provide direct memory access to the data without incurring any function call overhead. The C++ language has been defined so that software can be written to run very efficiently. Programmers do not have to leave the context of the language to obtain needed efficiencies.

Features Not Part Of The Language

C++ does not have a large virtual machine environment directly providing features like garbage collection and graphics capabilities by default. These are not required in all application environments and can be provided effectively through libraries. Their absence in the language allows C++ to be used in a wider set of system environments than some object-oriented languages. Companies are producing development environments around C++ that do not intrude into the language itself, but do provide many of the interactive development facilities characteristic of languages with built-in environments.

C++ does not have a built-in meta-class facility. In addition, no generally available C++ environment yet supports the run-time creation and integration of new types into running processes, though this is possible and has been done. A new class can be derived from a given base class and be incrementally compiled and linked into a running process. Software interfacing at the base class level can then manipulate instances of the new derived class via virtual functions.

Another feature some object-oriented languages/environments provide is *persistence,* allowing objects to be placed on secondary storage so they can exist across processes. Some languages also support mechanisms to handle *concurrency,* sometimes used with persistence. C++ does not directly support persistence or concurrency. Several companies have developed C++ object-oriented databases that provide these capabilities. Persistence is an example of a language feature that may be more appropriately provided by libraries. By not incorporating the feature as a built-in language mechanism, applications not requiring the feature are not constrained by its presence.

Parameterized types and *exception handling* are currently not defined in the language and thus are not yet available in commercial C++ implementations. Techniques exist within the current language definition to approximate these features, but the techniques have drawbacks. Parameterized types and exception handling are likely to be added to C++ in the future.

Summary

The C++ language provides the key capabilities and benefits offered by object-oriented programming, without including features that would constrain its use to a limited set of application domains and environments. The mechanisms are defined to allow very efficient implementations and an easy migration path for the large amount of existing C software and programmers. Features that would result in performance penalties have not been included in the language. Instead, the language provides base functionality permitting developers to provide needed mechanisms efficiently. This provides an example of the efficiency and versatility offered by the language. ■

David Jordan is a Distinguished Member of Technical Staff at AT&T Bell Laboratories. He has been a lead developer on several large projects building systems in C++ since 1985. He has given courses and presentations on C++, object-oriented design and the project management and methodology implications of using object technology. He is currently evaluating object-oriented database technology.