C++: EVOLVING TOWARD A MORE POWERFUL LANGUAGE

James O. Coplien, Stephen C. Dewhurst, and Andrew R. Koenig

James O. Coplien *is a member of technical staff in the Advanced Software Technology Department at AT&T Bell Laboratories (Indian Hill Park) in Naperville, Illinois.*
Stephen C. Dewhurst *is a member of technical staff in the UNIX® Applications Environment Systems Engineering Department at AT&T Bell Laboratories in Summit, New Jersey.*
Andrew R. Koenig *is a distinguished member of technical staff in the Software Systems Department at AT&T Bell Laboratories in Liberty Corner, New Jersey. Mr. Coplien does applied research in system hardware architectures to support software productivity for large real-time systems. He joined the company in 1979 and has both a B.S. in electrical and computer engineering and an M.S. in computer science from the University of Wisconsin–Madison.*
(continued on page 32)

Programming languages don't appear by spontaneous generation. Instead, people develop them because they have new ways of thinking about the process of programming and have new types of problems to solve. When we generalize and codify these thought processes, they achieve the status of paradigms, and new languages are created to support those paradigms. The language enables programmers to use the paradigm's power to deliver software of improved quality and lower life-cycle cost. This paper shows how AT&T's C++ language supports several modern programming paradigms to improve software quality and boost productivity. The language is more expressive than C, leads to more maintainable programs, and employs constructs that encourage software reuse and software architecture techniques that deal with complexity in large systems.

Background

Programming languages are produced by shifts both in the way people think about the programming process and in the types of problems they attempt to solve. Generalized and codified, these thought processes achieve the status of paradigms,[1] and new languages are created to support them. This is the case with C++, a language that was developed at AT&T and is being widely used both inside and outside AT&T today.[2] In this paper, we examine how C++ provides linguistic support for the data-abstraction[3] and object-oriented[4] programming paradigms, in the context of the properties for which they were developed: expressiveness, evolution, reuse, and dealing with complexity.

When a programming language effectively supports a given problem area or paradigm, we can justify labeling it an X-language, where X is the problem domain or paradigm in question. So we say that COBOL is business-oriented, that Fortran deals with formulas for numeric calculations, and that the Smalltalk-80™ language[4] is object ori-

ented. (Smalltalk is a trademark of Xerox Corporation. Panel 1 defines acronyms used in this paper.) We are not saying that one cannot use the object paradigm with COBOL nor write business-related programs in Fortran. But such programs would not have easy, natural representations in those languages. One can argue that these languages do not *prevent* an object-oriented style, but cannot argue that they *support* it—and some languages make it downright difficult.

C++ *supports* object-oriented programming. But it also is a multiparadigm language that contributes to quality software and increased productivity for general-purpose applications. As a system programming language in the heritage of C, it applies to a broad spectrum of application domains and can be used to complement application-oriented languages (AOLs) or even to create the building blocks for an AOL environment.[5] Because it provides effective support for the use and melding of several modern programming paradigms described here, C++ extends the use of C into the future.

The next section looks at data abstraction as one important paradigm that underlies C++. Next, we look at a simple, but typical, software exercise (a string package) to show some commonly encountered programming pitfalls, and highlight opportunities for improving software quality. Then we show how to use C++ features for strings to address those problems. Finally, we introduce object-oriented programming and look at the application of C++ to increase productivity in large-scale developments.

Panel 1. Terms in This Paper

AOL	application-oriented language
COBOL	common business-oriented language
Fortran	formula translator (language)
ISDN	Integrated Services Digital Network
`malloc`	C library routine to allocate memory
`printf`	C library routine for printing messages
PODS	plain ordinary data service
POTS	plain old telephone service
`RD`	read
`RDWR`	read and write
Snobol	string-oriented symbolic language
`strcat`	C library routine to catenate to end of a string
`strcmp`	C library routine to compare two strings
`strcpy`	C library routine to copy a string
`strlen`	C library routine to find a string's length

Data Abstraction

An *abstract data type* is described completely by an explicit, user-defined set of operations. This user-defined data type is abstract in that irrelevant details of its implementation are abstracted away, and—as far as its users are concerned—its public interface entirely represents its properties.

This approach to software design reduces complexity, a major advantage. Only the provider can access the data type's implementation, so it can be changed without affecting client code. Because the public interface completely describes the data type's semantics, users of the type can ignore irrelevant complexities of its hidden implementation. This increases modularity and reduces total program complexity.

Equally important, however, is the use of abstract data types to bring programming language closer to the specific problem domain. For example, writing compact, clear string-manipulation programs in Snobol is easy, because of its significant linguistic support for string manipulation.[6] The ability to define an abstract string data type would confer similar advantages without requiring a built-in string type in a language.

In addition, it is not practical or possible to predefine all data types of interest for any general-purpose programming language. For example, a telephone data type has great applicability to switching applications but would be of little interest to most other users of a language. If users can define abstract data types effectively, they can customize the base language to support programming in a particular area when the expense of producing an application-oriented language cannot be justified.

Why Is C Programming So Hard?

When programming in C, much time and effort goes into dealing with little details and getting them right.

Panel 2. Setting the Value of a String Variable

These examples show the evolution of a program excerpt that defines a string variable whose value consists of the values of two other string variables.

A. Release 1, the straightforward approach:

```
/* pointers to characters, used to point
 * to a character vector (i.e. a string)
 */
char *machine, *service;
char dialstring[32];

 /* copy machine into dialstring*/
strcpy(dialstring, machine);
 /* catenate a period onto dialstring */
strcat(dialstring, ".");
 /* catenate service onto dialstring */
strcat(dialstring, service);
```

B. Release 2, string length increased and length check added:

```
char *machine, *service;
char dialstring[100];  /* bigger */

 /* error check */
if(strlen(machine)+strlen(service)
+1 > 100)
 error();
strcpy(dialstring, machine);
strcat(dialstring, ".");
strcat(dialstring, service);
```

C. Release 3, memory allocated dynamically for string:

```
char *machine, *service;
char dialstring;

/*malloc dynamically allocates memory*/
dialstring = malloc(strlen(machine)
+strlen (service) + 1);
strcpy(dialstring, machine);
strcat(dialstring, ".");
strcat(dialstring, service);
```

D. Program failed when the string was 100 characters long, so Release 4 increased the memory allocated by 1 byte:

```
char *machine, *service;
char *dialstring;

dialstring=malloc(strlen(machine)
+strlen(service)+2);
strcpy(dialstring, machine);
strcat(dialstring, ".");
strcat(dialstring, service);
```

E. What if program runs out of memory? Release 5 added an error check:

```
char *machine, *service;
char *dialstring;

dialstring=malloc(strlen(machine)
+strlen(service)+2);

 /* malloc failure check */
if (dialstring == NULL)
 error();
strcpy(dialstring, machine);
strcat(dialstring, ".");
strcat(dialstring, service);
```

F. Program never releases the string, so Release 6 provided the fix:

```
char *machine, *service;
char *dialstring;

dialstring=malloc(strlen(machine)
+strlen(service)+2);
if (dialstring == NULL)
 error();
strcpy(dialstring, machine);
strcat(dialstring, ".");
strcat(dialstring, service);

free(dialstring);
```

G. The same program fragment but written in C++, with a good string package:

```
String machine, service;
String dialstring;

dialstring = machine + "." + service;
```

As a hypothetical—but plausible—example, consider a code fragment to solve this simple problem.

Suppose we have two variables, `machine` and `service`, that contain the name of a destination (such as `nj/murray/hill`) and a network service (such as `mail`), respectively. (In our example, the destination is a machine name. It tells where the machine is located—the state, and the site and node, if the site has several nodes—and its UNIX® system name.) We want to set a variable, `dialstring`, to a single string that represents the full name of the network-service destination to call (in this example, `nj/murray/hill.mail`). Conceptually, this is a trivial problem—the kind surely solved thousands of times in thousands of programs. Let's look at one such solution (Panel 2).

In release 1, the developer took a straightforward approach (see A in Panel 2). Just as this problem is typical of those that occur routinely in writing C programs, so is this solution typical of those seen in actual use. And just as typically, it doesn't work. The program crashed when a user supplied a machine name that was longer than 32 characters.

To fix that bug, our developer increased `dialstring`'s length from 32 to 100 characters, and put in a check to ensure that the value wasn't too long. That change went out in release 2 (see B in Panel 2). But the developer thought that no machine name would ever exceed 100 characters, which proved just as wrong as the original thought that no name would ever exceed 32 characters. The resulting error was as devastating as a crash, but easier to find.

The developer wisely decided to avoid merely increasing the length again. Instead, the code in release 3 allocated memory dynamically to hold `dialstring` (see C in Panel 2). Now the amount of memory allocated was the length of `machine`, plus the length of `service`, plus one extra byte for the period that separates them.

Unfortunately, the developer forgot to allocate a byte for the null character that C puts at the end of the string. (More about that character later.) So release 3 clobbered one byte of dynamic memory when the dialing string's length was one less than a multiple of the machine's word size. (Lest you think this example too contrived, the `mail` command in UNIX System III once had exactly this bug. It took 16 hours to find.) Because of the problem's apparently random nature, it made it past system test to provoke random errors in the field.

It took a long, arduous debugging effort to locate this problem. No one noticed that it had existed in release 2 as well. There, it would have turned up only if `dialstring`'s value had been exactly 100 characters long. Once found, the problem was easily fixed for release 4 (see D in Panel 2).

With this one-character change, our program made it through system test and into the field again—where it still crashed. Because real applications use more memory than the test cases, our program occasionally ran out of memory. The `malloc` program (a C-library routine that allocates memory) failed gracefully enough; it returned a `NULL` pointer. But the pointer wasn't checked, so `strcpy` (which copies strings) caused mayhem by trying to copy characters into memory location zero. So release 5 incorporated an error check (see E in Panel 2).

Finally this program fragment no longer crashed (but it still fails). As the application program became more reliable, it started running for longer periods between crashes. It was now discovered that the program was gradually eating up memory—`dialstring` was never being freed. This was easy to fix in release 6 (see F in Panel 2).

Look what has happened so far. A simple code fragment took six tries to get right. If we account for calendar time spent to distribute releases, process bug reports, and so on, it is realistic to expect this entire process to take months or years to complete. And if we account for required changes in functionality, it never ends.

Consider the expense and wasted time in all this, and then consider that our developer, if programming in C++ with a good string package, would have had a hard time not doing it right the first time (see G in Panel 2).

Defining the Semantics of an Abstract Type

Being able to define a `String` class that is as easy to use as if it were built into the language is a powerful thing. But it is far from obvious how a language might allow itself to be extended that way. To see how to do it, first let's examine how C programs typically handle

character strings.

C has no direct support for character strings; its closest notion is that of an array of characters. By convention, C programmers normally represent a string as such an array of characters, with a *null character* (represented in C programs as `'\0'`) after the last character. To support this convention, when a programmer writes some sequence of characters enclosed in double quotes, C language automatically adds a null character at the end of the sequence. Thus, a C programmer can write

```
printf("Hello world\n");
  /* \n is newline character */
```

instead of

```
printf("Hello world\n\0");
```

In both examples, the `printf` function sees a pointer to the string's initial character. The act of passing a string to a function does not copy the string's characters; only the string's starting address is passed.

In C, there is no built-in way to copy a string! For instance, if we say

```
char *p;
char *q;
p = "hello";
q = p;
```

then `p` and `q` refer to the same memory. So if we change a character of `p`, we also change that character of `q`. In fact, C does not offer a type `S` that allows a programmer to say

```
S foo;
foo = "Hello world\n";
```

and have the value of the variable `foo` be a copy of the string. Instead, a programmer has several choices.

One possibility is the `strcpy` function from the C library:

```
char hello[13];
strcpy (hello, "Hello world\n");
```

This simple function copies characters from memory addressed by its second argument into memory addressed by its first argument until it has copied a null character. It is the programmer's responsibility to ensure that enough memory exists to receive a copy of the value.

In this example, we counted the characters and allocated just the right amount of memory. (The count is 13 instead of 12 because the null character takes up room, too.) Many programmers do just this. They allocate some arbitrary amount of memory and hope it's enough.

A cleaner way to do it is to use `malloc`, the C library routine for memory allocation:

```
char *hello;
hello = malloc(strlen("Hello world\n")
+ 1);
strcpy(hello, "Hello world\n") + 1);
```

Besides the obvious inconvenience of writing all that code and the danger of changing one copy of the string but not the other, this practice places two more burdens on a programmer:

- Check `malloc`'s result to ensure that memory has not been exhausted.
- Free the memory after it is no longer needed. Otherwise, a program that runs long enough will eventually consume all available memory.

Thus, a programmer who wants to write a C program using character strings must remember several conventions:

1. Assigning a "string" variable to another one does not copy the characters of the string.
2. Before copying a string or creating a new string, allocate enough memory to hold it.
3. When allocating memory for a string, don't forget to count the null character at the end.
4. Check for unsuccessful memory allocation.
5. Free the memory that a string uses when done with it.

C does not help programmers remember these things. The price of failing to remember one of these conventions is a program that fails wildly and unpredictably.

Strings in C++. We can greatly simplify string manipulation in C++ by incorporating these conventions into a program. Although this program will be more complicated than the C library's corresponding string-manipulation functions, it need only be written once and is easier and safer to use.

Panel 3. A Class Declaration

```
class String{
public:
   String();        // default constructor
   String(char*);   // constructor to make a String from a char*
   String(String&);// constructor for copying Strings (automat-
                    // ically called for argument passing etc.)
   ~String();       // destructor to clean up a String

   String& operator=(String&);  // handles String assignment

/*
 * relational operators: not part of the class per se;
 * defined elsewhere, but declared as friends here so
 * they have complete access to String internals. They each
 * take a pair of references to Strings as arguments.
 */
   friend int operator==(String&, String&);
   friend int operator!=(String&, String&);
   friend int operator<(String&, String&);
   friend int operator>(String&, String&);
   friend int operator<=(String&, String&);
   friend int operator>=(String&, String&);

   operator char *(); // for conversion (casting) back to
                      // a C string
private:
     :              // internal stuff private to String
};
```

We will create a *class* as the framework for managing strings in C++. A C++ class is a user-defined type whose representation and behavior can be defined completely by a programmer. To follow the convention of many C++ programmers, we will start the name of our string class with an upper case letter and call it a `String`. For illustration, we will define a class that is much simpler than one might want in practice. Nevertheless, it is rich enough to use.

Classes in C++. We first define the interface to our `String` class. Clearly, we must be able to create a `String`, perhaps giving it an initial value from a C "string." We must be able to pass a `String` as an argument to a function and assign one `String` to another. We would also like to be able to compare two `Strings` and to convert a `String` back to a C "string" for output purposes.

This leads us to the class declaration in Panel 3. The `String` class members declared there parallel the C routines discussed above. The first three are *constructors*; they say how to build a `String`. Next is a *destructor*, which tells how to clean up a `String`, and so on. C++ arranges to call the appropriate constructor every time a `String` is created and to call the destructor when the

Panel 4. Defining Constructors

```
  /* body of default constructor for String */
String::String()
{
   s = new char[1]; // dynamically allocate char
                    // array of size 1
   *s = '\0';
}
  /* body of constructor to build a String from a
   * C character vector
   */
String::String(char *p)
{
s = new char[strlen(p) + 1];
   strcpy(s, p);
}
  /* body of constructor to build one String from an
   * existing one,
   * Called automatically when a String is passed as a
   * parameter, returned as a value from a function, etc.
   */
String::String(String& str)
{
   s = new char[strlen(str.s) + 1];
   strcpy(s, str.s);
}
```


`String` is destroyed. The *relational operators* are defined as `friend`s because they need to access "private" data; a `friend` has all the privileges of members but is not a member. Because all these declarations appear after a `public` label, they represent the *public members* or *interface* to the `String` class. In general, the keyword `operator` and the operator symbol (e.g., `==`) that follows it represent the name of a function that will be called whenever the operator symbol is used. Thus, if `s` and `t` are `String`s, then `s==t` will call `operator==`.

Implementing a String. We are now ready to decide how to represent a `String`. For this example, we do it about the simplest way possible: as a pointer to a dynamically allocated array that contains the characters that form the `String` followed by a null character to mark the end.

This pointer is part of the *implementation* of the `String` class. We do not want users getting at the pointer, because we might want to use a different representation later. So, we make it part of the private data of the `String` class by putting it after the `private` label:

```
class String {
        ⋮
private:
   char *s;
};
```

For convenience and safety, we will adopt the convention that *every* `String` has a value. If a `String` is not other-

Panel 5. Assigning a String Value

```
String&
String::operator=(String& x)
{
   char *save = s;
   s = new char[strlen(x.s) + 1];
   strcpy(s, x.s);
   delete save;
   return *this;
}
```

wise initialized, it will have a null value.

Defining operations. We are now ready to start defining the `String` operations. The constructors are pretty trivial (see Panel 4). In each case, to make room for the null character at the end, we allocated one more character than we needed for the data. Thus, an uninitialized `String`'s constructor allocates a single character of memory and puts a null character there, and the other two constructors allocate enough to hold a copy of the data being used for initialization.

The destructor is similarly simple:

```
String::~String()
{
     delete s; //delete frees memory
               //allocated by new
}
```

Because an object cannot be destroyed without having first been constructed, we know that it is safe to delete `s`. Besides, every `String` constructor stores the address of allocated memory in `s`!

Assigning one `String` to another is a little tricky (see Panel 5). The obvious thing to do is to free the memory that corresponds to the old value, allocate memory for a copy of the new value, and copy the old value into that memory. However, that will fail if the user assigns a `String` to itself. To forestall that, we save a pointer to the old value, and free it only after we have copied the new value. The statement "`return *this;`" is a common C++ idiom. It means: the value of this operator is the object being operated on [e.g., so `a = (b = c)` works].

The relational operators are straightforward but somewhat tedious because there are six of them. We show only the == operator; the other five are analogous:

```
int
operator==(String& p, String& q)
{
     return strcmp(p.s, q.s)==0;
}
```

Converting a String. Finally, we must decide how to convert a `String` back to a `char*`: Do we just return a pointer to the actual data and trust the user? Or, do we return a pointer to a copy of the data and make the user responsible for freeing the copy? Here, we do the latter:

```
String::operator char *()
{
     char *r = new char[strlen(s) + 1];
     strcpy(r, s);
     return r;
}
```

Advantages of Using C++. We have seen how a small amount of programming in C++ can greatly simplify a common, error-prone task in a way that would have been completely impossible in C. Several aspects of C++ combine to make this possible.

First is the ability to define a *class* with *public* and *private* data. The class definition limits the operations that can be performed on a class. When the structure of the string is defined as private data, programs that use strings are protected from damage if the implementation changes.

Next are the notions of a *constructor* and *destructor*. Together, they make it possible to guarantee that an object will always have a meaningful value and that the memory this value occupies will always be released when the object is freed.

Finally, the C++ programmer can explicitly define the semantics of assignment and argument passing. This makes it possible to ensure that the implementor can control *every* reference to an object.

In fact, our example's only trouble spot is in converting a `String` to a C "string," and that trouble stems unavoidably from the C notion of strings.

Objects, Inheritance, and Object-Oriented Programming

In the string example above, for the most part, `friend` functions provide the public interface to the `String` type. These functions have special access permission to the implementation parts of the type. In effect, the union of these functions with the private representation is what makes up the *abstract data type*.

To make this binding of data and operations even more explicit, one can use functions that are members of the type. For example, an abstract data type that represents a telephone could define its public interface as a set of these member functions. Because functions that define the type's external properties are also part of the type and not merely functions with special access permission, we can say that instances of the type define their own behavior. Instance variables that are created using a class type as a template are called *objects*. They are autonomous units that have a state (data) and behavior (as defined by functions and operators).

Data abstraction is a useful mechanism when one can identify a single, clearly-defined concept, such as the `String` type of the previous section. However, many problem domains are not easily modeled as distinct types. Instead, they are more naturally represented as collections of related types, or modifications to or combinations of existing types. For example, a switching application may have to deal with several kinds of telephones. These telephones will share a common core of properties (the properties that make them telephones), but each kind of telephone will have additional properties that distinguish it from the others. When only data abstraction is used to represent all the telephone types, one has the choice of creating either a single, general type that incorporates the complexities of all the others, or a set of distinct types that

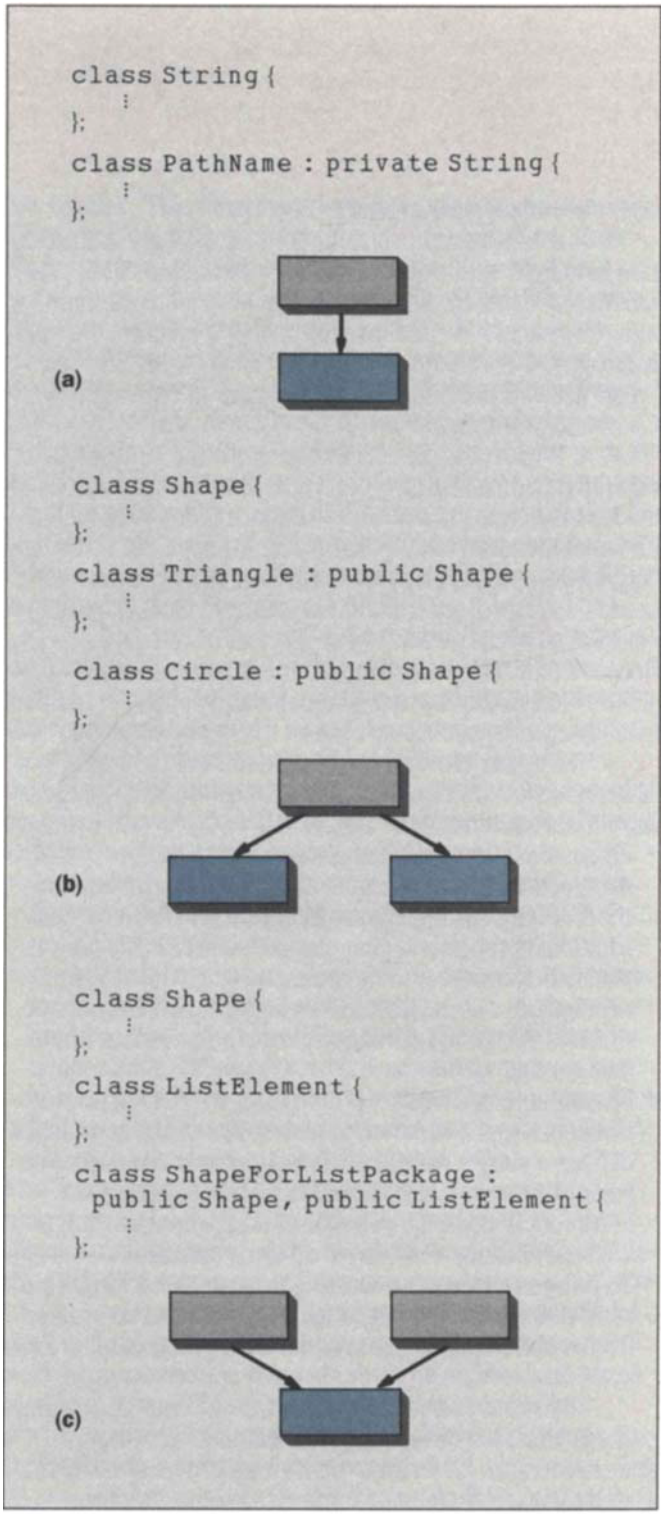

Figure 1. A simple taxonomy of class inheritance. In general, each class has its own properties but some "borrow" other properties. (a) Derivation; here, the subclass borrows general behavior from its parent. (b) Multiple derivations; a family of subclasses shares general behaviors from a common parent. (c) Data abstraction—specifically, multiple inheritance; a subclass borrows its general behavior from several parent classes.

does not reflect their commonality.

A better approach is to extend the data abstraction paradigm with the concept of *type inheritance.* (Type inheritance goes by different names in various programming languages. In the Simula67™ and Smalltalk languages, it is called "subclassing"; in C++, it is called "class derivation." Simula is a trademark of Norwegian Computing Center.)

What is the idea of inheritance? When a new type is created from an existing type, it inherits all the properties of the existing type and new properties are added or the inherited ones are altered. So in our switching application, we can define a generic `Telephone` abstract data type that encapsulates properties common to all telephones, and use inheritance to create a family of specialized telephone types. Because they inherit the properties of the generic telephone, each of the new types remains a telephone in addition to being a more specialized type.

Besides providing collections of related types, inheritance has other uses. Figure 1 shows three basic ways to use inheritance:

- To customize an existing abstract data type for a particular application, as in Figure 1a. Here, a pathname in the UNIX operating system is a `String` with certain additional properties. One can use an arbitrarily long chain of derivations to create increasingly particular refinements, each class borrowing general behaviors from its parent but defining specific behaviors of its own.
- For multiple derivations (Figure 1b), a family of subclasses shares a common parent. None of the subclasses defines a subset of the behaviors or implementations of any of the others, but all the subclasses share some behaviors in common. The parent class contains the implementations common to all the subclasses.
- To merge two or more existing types by inheriting from all of them. For example, a list of shapes can be created from a list-element type and the `Shape` type, as in Figure 1c.

Another important effect of C++ type inheritance is that it permits efficient dynamic binding. That is, generic code can be written for a group of types that are related by inheritance. At run time, the object's type determines what operations are to be performed, as we show in the next section.

This ability to express commonality has obvious benefits. It helps a system designer abstract a problem and encourages code sharing. (We discuss these benefits in the next section.) Another benefit of C++ dynamic binding is its negligible run-time cost. Code that the compiler generates automatically is about as efficient as a programmer's code.

The use of objects with inheritance and dynamic binding forms the basis of the object paradigm, and is called *object-oriented programming.*[7]

C++ and Productivity in Large-Scale Development

Problems of cost and maintainability loom disproportionately large on large projects. When a system grows larger than a small group of people can grasp, traditional abstraction mechanisms break down in architecture and design, as does the way the development is organized. Our experience with C++ verifies that it offers a rich set of notations and conventions (or paradigms) that work well to nurture large projects.

To handle complexity—especially the high level found in large, highly reliable, real-time systems—system architects and designers use several paradigms, most of which are forms of abstraction. Abstraction helps in two ways:

- It *hides* complexity that really is there.
- It *reduces* artificial complexity that might arise from the application of a less well-suited approach.

We want the solution's total complexity to fit the problem's complexity. And, where appropriate, we want to push details into "black boxes," so they aren't visible on the surface.

With C++ , architects and programmers have much latitude in choosing paradigms to fit their own style and the project's needs. Many methodologies that accompanied the rise of structured programming in the 1970s tended to be dogmatic and exclusionary. "Top-down" advocates showed disdain for "bottom-up" programming, and vice versa. In the same vein, some contemporary languages limit the architect to constructs that are all high level, all low level, or just for specific custom applications.

Panel 6. A Program Fragment

This is part of a function for copying an existing file into a temporary one:

```
char *mktmp(String original_file){
    char *tmp = tmpnam( NULL );
    FILE *to_fp = open(tmp, O_RDWR);
    FILE *from_fp = open(original_file,
    O_RD);
      :
}
```

C++ does not fall into these traps. It is not only a general-purpose language, but also supports a range of paradigms. These paradigms can be used either separately on separate projects or to complement each other within a single development. For example, C++ does not dictate bottom-up or top-down programming, but allows either and suggests an approach that combines both.

Much of C++'s architectural support results from the object paradigm, and C++ is often used in a way suggested by Booch[8] and others. It supports separation of concerns; implementors need to worry about the innards of only the classes that they modify. It allows architect and coder to work separately or together, or even be the same person. In any case, the class interface can serve as a behavioral specification to be handed off from architecture into development. Because C++ does not limit low-level access, programmers can write code close to the machine level for high-performance algorithms, custom-device control, context-switching management, etc. Also, such low-level data structures and functions can easily be hidden inside a high-level language envelope like a class.

The original programming abstraction was the function or procedure, and C functions can still be used in C++. But in addition, name overloading adds more abstraction power. (*Name overloading* is using the same name to mean different things in different contexts.) Consider the program fragment in Panel 6, part of a function used to copy an existing file into a temporary file. Notice that the programmer can freely use the function `open` to deal with any of the string representations that might be in the system. This function is an abstraction mechanism that hides implementation differences in different parts of the whole system. Operators such as the arithmetic + or –, and even operators such as –>, can be similarly overloaded to abstract nuances of behavior.

A system designer can use these abstraction tools along with other C++ abstractions, such as data aggregation (similar to C structures). These serve primarily as conveniences to system coders, and can contribute to productivity gains on small projects. But on large projects, the focus for productivity gains is in architecture and design, because coding is a small fraction of life-cycle cost. (Although the actual cost may show up in testing, the fix is in architecture and design; see Reference 9.)

The real power of C++ is its ability to express abstractions that map onto things that are found in the problem domain, bringing together a resource's behavior and representation into a single abstract data type, or class. The architect can create new classes that capture the behavior of real-world resources, and that also hide the details of their representation and implementation. This achieves two ends for the system implementor:

- It encodes architectural components as building blocks that have full first-class standing as programming language types.
- It allows the implementor to work in terms of the *behaviors* of system components—not their implementations—something the implementor already understands. In other words, it accurately mirrors the application's complexity in the architecture—no more, no less—while encapsulating the lowest levels of complexity so they are not globally visible.

This approach can apply to abstractions as simple as strings or as involved as telephones (see Panel 7). The object paradigm has recently received much well-deserved attention as a way to create architectures.[10]

Inheritance. A large system may have thousands of classes, so classes alone are not enough to reduce complexity to manageable proportions. The architect needs to be able to group classes into families whose members exhibit similar behavior, perhaps even allowing shared implementations among such classes.

Inheritance can be used to group like classes of

things into conceptual trees—just as phyla and their subtrees, down to genuses and species, are organized—from the root of the tree as the most general class, to the leaves as the most detailed identifiable entities. But inheritance is *not* stepwise refinement, where the collection of units at one level of abstraction composes an element at the next higher level. With inheritance, each level increases in specialization and detail, but each level is itself a completely defined and operational entity. As Figure 1 shows, we can think about inheritance in three ways: derivation, multiple derivations, and multiple inheritance.

The style of inheritance depicted in Figure 1a is called *derivation*. A newly formed type is based on the functionality of another but builds on it, modifies it, and augments it to make a more refined or specialized class. As class `PathName` can be derived from `String` in Figure 1a, so any derived class reuses the implementation of its base class.

In both Figures 1a and 1b, the architect can choose to arrange for powerful dynamic run-time behavior. Let's say that a programmer is writing some telephony code, and the code has been passed a pointer (a memory address) to an object that controls a telephone. The pointer may be a reference to *any* kind of telephone object—one that controls an analog phone, a digital one, or perhaps a video phone. Our programmer wants to make some general requests of the object. The requests are common to all telephones but might be implemented differently for each kind (e.g., to ask if a telephone is hung up or not).

Suppose the programmer knows that all these objects come from classes that form a family that has `Telephone` as a parent class in common. Then, he or she can write the code as though dealing only with the parent class `Telephone`. To make this work, our programmer declares the parent's member functions to be *virtual*. That is, they serve as placeholders for the functions that are associated with the subclasses and are identified and selected at run time.

This is a very powerful abstraction mechanism. Whole families of classes can be treated as if they have the behavior of one class, the family's parent. Also, a member of a family of types need only be concerned with the specific behaviors of its ancestors, never with its descendants or cousins. For example, all `Shapes` in Figure 1 can be treated as if they were of type `Shape`. An object that manipulates shapes in some generic way does not need to know what shape it is handling.

Panel 7. A C++ Description of a Telephone

```
class Telephone{
public:
   void ring();
   void lightMessageLamp();
   void enableTalking();
   void disableTalking();
   Boolean isBusy();
private:
   int extension;
   int lastNumberDialed;
   Boolean forwardingOn;
   int forwardTo;
};
```

Similarly, one might declare both `ISDNPhone` (Integrated Services Digital Network) and `POTSPhone` (plain old telephone service) to be subclasses of `Telephone`. The implementor of `ISDNPhone` can reuse all existing code in `Telephone` that implements the generic "phoneness" of all telephones.

For example, suppose someone wants to write a function that takes an array of pointers to different kinds of phones and rings each phone. That function doesn't have to worry about what *kinds* of phones are configured, because the right "ring" function is selected at run time (Figure 2). Furthermore, someone can still add a new class `PODSPhone` (plain ordinary data service) *without changing any existing code* in `Telephone`, in any other related classes, or in classes that use general `Telephone`-class operations. Adding such a new class can not only leave existing source code unmodified, but also does not require regenerating any existing object code. Productivity is enhanced through reuse and the resulting ease of evolution.

Data abstraction is the creation of a type—a kind of abstraction—from data and procedures. In Figure 1c,

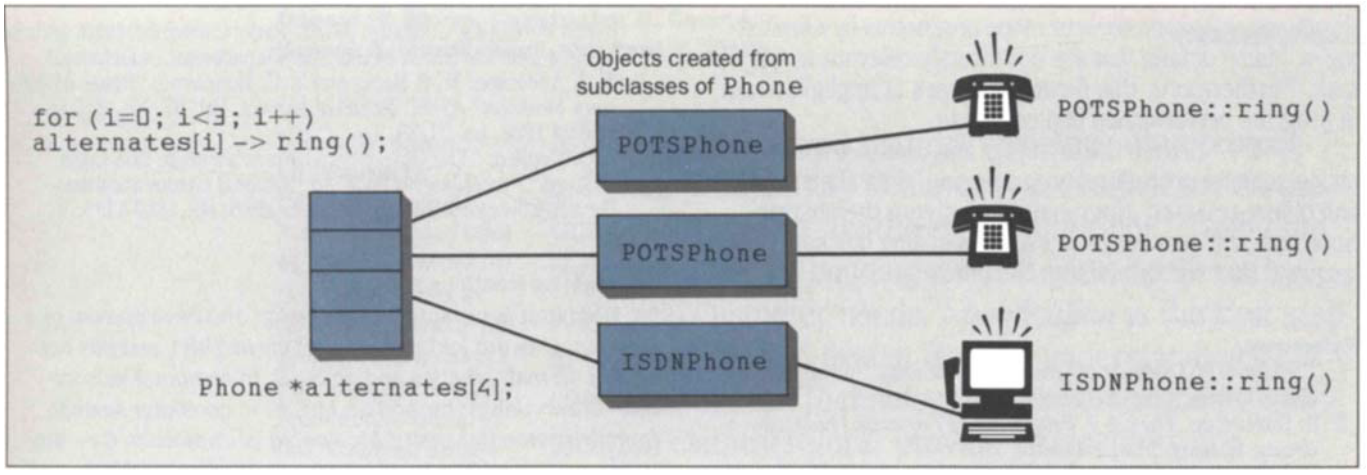

Figure 2. Automatic selection of right `ring` operator at run time. The function at the left uses the pointers in the array to identify the telephone to ring.

the behaviors of several existing classes are abstracted into a single class that has first-class standing as a C++ type; this is called *multiple inheritance*. Multiple inheritance is a higher order abstraction: the creation of a type from other types. It is sometimes needed to reflect accurately the semantics of the problem domain (as with Figure 1c). In any case, multiple inheritance is a powerful software-reuse mechanism that makes it easier to combine the behaviors of multiple parent classes.

Other Advantages. In addition, our own experience at AT&T with C++ in large systems bears out other advantages. Perhaps most striking is that its use, compared to conventional approaches, greatly reduced software integration times. This results in part from C++'s static type checking, and partly because, from the beginning, the project worked with a "common ground" of C++ class specifications to build against. It reduces the need for communication among implementors late in the project and throughout the maintenance phase. (Such communication is greatest early in development, when interfaces are being specified and coded.) This is partly because the object paradigm is used, but is motivated by the language itself, which supports and encourages the use of the abstractions it supports at the language level. However, this increased startup cost pays off in an overall productivity increase and lower life-cycle cost.

Another major lesson was: After programmers cultivated an understanding of the object paradigm and of how to use C++ features to complement each other, no code efficiency was lost while they took advantage of the language's power to increase productivity. While the C++ language constructs gave them a more expressive and higher level programming interface than C, they produced code as efficient, and almost as compact, as the corresponding C code would have been. Code that used only the C subset of C++ produced the same code as an ordinary C compiler would have, preserving the programmer's ability to get close to the machine to write real-time code.

Conclusion

C++ provides a solid foundation for software reuse by facilitating safe and convenient software packaging that can be used to build libraries of programming services. These services may be simple and common, like strings, or internally complex and application-specific, like a telephone object.

To reuse an existing module's design and implementation, one can derive a new module from it, customizing and elaborating where necessary. Powerful run-time type support and storage management make

designers and programmers more productive by eliminating or hiding details that are not directly relevant to the task. Furthermore, this flexibility comes at negligible cost in program performance and code size.

C++ leaves the architect and designer free to choose among procedural programming, data abstraction, and object-oriented programming to create the abstractions that will best serve as system building blocks. It is a language that will take C into the future.

References

1. T. Kuhn, *The Structure of Scientific Revolutions*, University of Chicago Press, Chicago, Illinois, 1970.
2. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 1986.
3. B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*, MIT Press, Cambridge, Massachusetts, 1986.
4. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.
5. D. W. Brown et al., "Software Specification and Prototyping Technologies," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 33-45.
6. R. E. Griswold, J. F. Poago, and J. P. Polonsky, *The Snobol4 Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.
7. B. Stroustrup, "What is Object-Oriented Programming?," *IEEE Software*, Vol. 5, No. 3, May 1988, pp. 10-20.
8. G. Booch, *Software Engineering with Ada®*, The Benjamin/Cummings Publishing Company, Menlo Park, California, 1983. (Ada is a registered trademark of the U.S. Department of Defense.)
9. K. J. Anderson, R. P. Beck, and T. E. Buonanno, "Reuse of Software Modules," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 71-76.
10. J. O. Coplien, "The Object Paradigm as a Future Life Cycle Method," *Proceedings of NCF/86*, National Communications Forum, Chicago, Illinois, November 1986, pp. 1110-1115.

Biographies (continued)

Mr. Dewhurst is working on the design and development of a C++ compiler. He joined the company in 1981 and has both an A.B. in mathematics and an Sc.B. in computer science from Brown University, and an M.S.E. in computer science from Princeton University. Mr. Koenig is involved in C++ education and class library development. His book, C Traps and Pitfalls, *will be published in September. He joined the company in 1977, and has a B.A. in mathematics from Columbia College and an M.S. in computer science from the Columbia School of Engineering and Applied Science.*

(Manuscript received April 13, 1988)