Inside: SPECIAL ALL-MACINTOSH SUPPLEMENT

# BYTE

AUGUST 1988

A McGRAW-HILL PUBLICATION

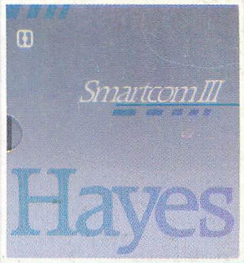
**REVIEWS**

Apple A/UX

PC Input Devices

Three 20-MHz 80386s

VersaCAD for the Mac




PRODUCT FOCUS
Script-Driven Communications

## The First of the 25-MHz Machines:
### Computing moves up another notch

**IN DEPTH**

## The C Language
with Kernighan and Ritchie, Stroustrup, and others

**PLUS**

Borland, Lotus, Norton on OS/2

Four New Columns

Short Takes:

- Dell System 220
- T-DebugPLUS
- Grammatik III
- Paradox OS/2

Netpro 386/25

Compaq Deskpro 386/25

Intel SYP302

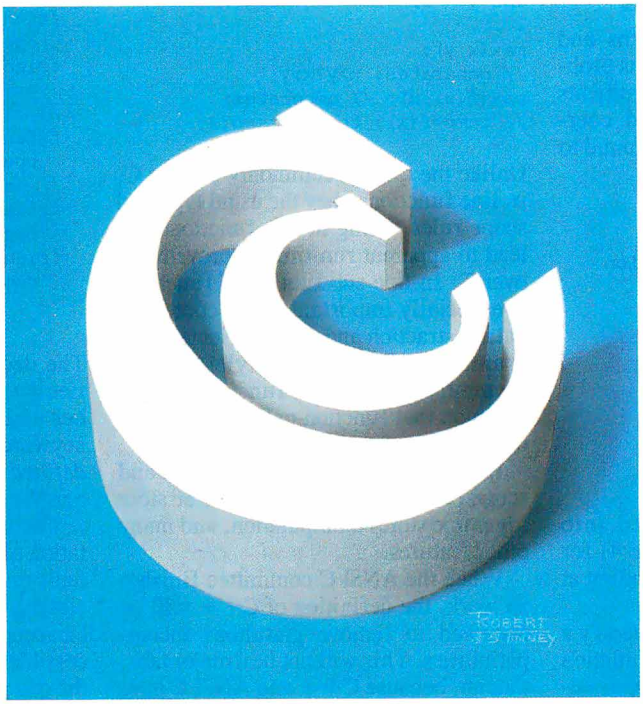
Everex Step 386/25


$3.50 U.S.A./$4.50 IN CANADA
0360-5280

# A Better C?

*This child of C goes its parent one better in compatibility and portability*

*Bjarne Stroustrup*



The C++ language is a general-purpose programming language that is, except for minor details, a superset of C. It improves on C through its support of data abstraction and object-oriented programming. The main influences on its design, in addition to C, were Simula-67 and Algol68 (see references 1 and 2).

C++ was first installed 5 years ago. Today, it has several independent implementations and many thousands of installations. It is being used for major university research projects and for large-scale software development in companies such as Apple, Apollo, AT&T, and Sun.

It has been applied to most branches of programming, including banking, CAD, compiler construction, database management, image processing, graphics, music synthesis, networking, programming environments, robotics, simulation, scientific computation, switching, and very-large-scale-integration design.

## A Better C

C++ improves the notational convenience of C and provides greater type safety. It compensates for C's weaknesses without compromising C's strengths. In particular, there is no program that can be written in C but not in C++, nor is there a program that can be written in C so that it achieves greater run-time efficiency than it does in C++ (see reference 3).

C is clearly not the cleanest language ever designed nor the easiest to use, but it owes its current pervasiveness to several key strengths:

- *Flexibility*: You can apply C to almost every application area and use almost every programming technique with it. The language has no inherent limitations that preclude writing particular kinds of programs.
- *Efficiency*: C's semantics are "low-level." That is, its fundamental concepts mirror those of a traditional computer. Consequently, it's relatively easy, both for you and for a compiler, to efficiently use hardware resources for a C program.
- *Availability*: Given any computer, from the tiniest microcomputer to the largest supercomputer, chances are that there's an acceptable-quality C compiler available for it, and that such a compiler supports an acceptably complete and standard C language and library. There are also libraries and support tools available, so you rarely need to design a new system from scratch.
- *Portability*: While a C program may not be easily or automatically portable from one machine (or operating system) to another, such a port is usually possi-

*continued*

ILLUSTRATION: ROBERT TINNEY © 1988

# The Origin of C++

Rich Malloy

C++ (pronounced "C plus plus"), like many other languages, began life as a tool to solve a specific problem. Bjarne Stroustrup, a Bell Labs researcher, needed to write some simulation programs. Simula67, the first real object-oriented language, would have been ideal for these programs except for its comparatively slow execution speed. Dr. Stroustrup chose instead to write a new version of C, which he called "C with Classes." By 1983, the language had evolved considerably and the name was changed to C++.

After further evolution, Bell Labs' parent company, AT&T, began offering the language as a product in 1985.

The name C++, like the language itself, is terse but meaningful. The name, coined by an associate of Stroustrup's named Rick Mascitti, concisely describes the evolutionary nature of the language. The term "++" is, of course, the increment operator in C, suggesting that the language C++ is "a bit more than C." A possible alternative name, C+, is not only less inspired but also liable to generate a syntax error.

*Rich Malloy is an associate managing editor at BYTE. You can reach him on BIX as "rmalloy."*

ble. The level of difficulty is also usually low enough that even porting software that contains inherent machine dependencies is both technically and economically feasible.

C++ preserves these strengths and remedies some of C's most obvious problems. For example, function arguments are type-checked in C++, and coercions are applied where they are found to be appropriate:

```
extern double sqrt(doublt);
// declare square-root function
...
double d1 = sqrt(2);
// fine: 2 is converted to
// a double
...
double d2 = sqrt("two");
// error: sqrt() does not
// accept a string
```

The // notation was introduced into C++ from BCPL (see reference 4) for comments starting at the // and ending at the end of the line.

As shown, C++ makes you specify a function's argument types in a function declaration so that the standard type conversions (such as `int` to `double`) can be implicitly applied, and type errors (such as calling a function requiring a `double` with a `char*` argument) can be caught at compile time. With minor restrictions, the draft ANSI C standard accepts the C++ function-calling rules and the syntax for function declarations and function definitions (see reference 5).

C++ provides in-line substitution of functions:

```
inline int max(int a, int b)
  { return a>b?a:b; }
...
int x = 7;
int y = 9;
...
max(x,y);
// generates: x>y?x:y
max(f(x),x); // generates:
// temp=f(x); temp>x?temp:x
```

Unlike the macros commonly used in C, in-line functions obey the usual type and scope rules. Using in-line functions can lead to apparent run-time improvements over C. In-line substitution of functions is especially important in the context of data abstraction and object-oriented programming. With these styles of programming, very small functions are so common that function-call overhead can become a performance bottleneck.

In addition, C++ provides typed and scoped constants, operators for free store (dynamic store) manipulation, and many other features.

When the ANSI C committee finishes its work, the definition of C++ will be reviewed to remove gratuitous incompatibilities. This will not be a major task, though, because C++ and ANSI C have already absorbed most of the "new ANSI C" features from each other.

For example, the notion of a pointer to "raw storage," `void*`, was incorporated into C++ from ANSI C, as were notational conveniences such as the suffix `u` indicating an unsigned literal (e.g., `12u`) and hexadecimal character constants (e.g., `'\xfa'`). However, the most important features of C++ relate to the support of data abstraction and object-oriented programming and are thus outside the scope of ANSI C and unaffected by changes in the draft ANSI C standard.

## Data Abstraction

Data abstraction is a programming technique in which you define general-purpose and special-purpose types as the basis for applications (see reference 6). These user-defined types are convenient for application programmers since they provide local referencing and data hiding. The result is easier debugging and maintenance and improved program organization.

In C++, you can define types that you then can use as conveniently as, and in a manner similar to, built-in types. Common examples are arithmetic types such as rational and complex numbers.

```
class complex {
  double re, im;
public:
  complex(double r, double i)
    { re=r; im=i; }
  complex(double r)
    { re=r; im=0 }
  // float->complex conversion
  friend complex
    operator+(complex, complex);
  friend complex
    operator-(complex, complex);
  // binary minus
  friend complex
    operator-(complex);
  // unary minus
  friend complex
    operator*(complex, complex);
  friend complex
    operator/(complex, complex);
};
```

The declaration of class `complex` specifies the representation of a complex number and the set of operations on it. The keyword `class` is C++'s term for *user-defined type*. The declaration of class `complex` has two parts.

The initial part specifies the representation of a `complex` number and is by default `private`. This representation (consisting of the two double-precision floating-point numbers `re` and `im`) is accessible only to the functions defined in the declaration of class `complex`.

The second part of the declaration specifies how a user can create and manipulate complex numbers. It is called the `public` part of the declaration because it provides an interface to the general public. It consists of two constructors and the usual arithmetic operations. A constructor is a function that constructs a value of a given type. The first constructor for `complex` creates a com-

*continued*

plex number given a coordinate pair; the second creates a complex number given a single floating-point number (using the obvious mapping of the real line into the complex plane). Together they provide the two obvious ways of initializing a complex variable. For example:

```
complex a = complex(1.2);
// a becomes (1.2,0)
complex b = complex(3.4,5.6);
```

The arithmetic operations are defined by `friend` functions: Specifically, these functions are completely ordinary except that they are granted access to the otherwise inaccessible representation of complex numbers by the `friend` declarations. The notation `operator+` is used to name a function defining the addition operator, +. The number of arguments determines whether an operator function implements a binary or a unary operator. For example, `operator-(complex,complex)` defines subtraction of complex numbers, whereas `operator-(complex)` defines unary minus.

Such functions can be defined as

```
complex operator+(complex a1,
  complex a2)
{
  return complex(a1.re+a2.re,
    a1.im+a2.im);
}
```

and used like this:

```
main()
{
  complex a = 2.3;
  complex b = complex(1/a,7);
  complex c = a+b+complex(1,4.5);
}
```

Here, `a` receives the value `(2.3,0)` by implicit application of the constructor `complex(double)`; `b` receives the value `(1/2.3,7)`; and `c` becomes the value `(2.3+1/2.3+1,7+4.5)`—that is, about `(3.7,11.5)`.

The constructors and the operator functions let you use complex numbers just as if they were built into the language. In-line functions let the run-time efficiency of a user-defined type come close to an equivalent built-in type.

Hiding the representation is the key to modularity. It allows the representation of a class to be changed without affecting users. For example, you might decide to change the Cartesian representation of `complex` used above to a polar one. Such a change would affect *only* the functions listed in the class definition. User code, such as `main()`, is unaffected. Debugging can also be greatly simplified by proper use of such data hiding.

Programming with classes shifts the emphasis from the design of algorithms to the design of classes (user-defined types). Each class is a direct representation of a concept in the program; each object the program manipulates is of some specific class that defines its behavior. In other words, every object in a program is of some class that defines the set of legal operations on that object. This lets you program in a language with a set of types, or concepts, appropriate to the application. An engineer might use complex numbers, matrices, and fast Fourier transforms, while the telephony-software designer might prefer types such as switch, line, trunk, handset, and digit buffer.

In C++, this style of programming is supported by a general and flexible set of mechanisms for data hiding, by constructors providing optional guaranteed initialization, by destructors providing optional guaranteed cleanup (termination), and by operator overloading and user-defined coercions providing a convenient and conventional notation for many kinds of applications. All these features are cleanly integrated into the language, and all uses are checked for type violations and ambiguities at compile time to catch errors as early as possible and to avoid unnecessary run-time overheads.

## Object-Oriented Programming

Concepts do not usually come as self-contained entities. On the contrary, most concepts relate to other concepts in a variety of ways. For example, the concepts of airplane and car relate to those of vehicle and transport; the concepts of mammal and bird relate to each other through the more general concept of vertebrate animal, through the concept of food, and so forth; and the concepts of a circle, rectangle, and polygon involve the general concept of a shape.

Therefore, representing concepts directly as types in a program also requires ways of expressing the relations between types. C++ lets you specify hierarchically organized classes. This is the key feature supporting object-oriented programming. Hierarchical organization is an extremely important way of coping with complex issues in many fields and has, not surprisingly, also proven to be a good way of organizing programs in a wide variety of application areas.

Consider defining a type `shape` for use in a graphics system. The system has to support circles, triangles, squares, and many other shapes. First, you specify a class that defines the general properties of all shapes:

```
class shape {
  point center;
  color col;
  // ...
public:
  point where() { return center; }
  void move(point to)
          { center = to; draw(); }
  virtual void draw();
  virtual void rotate(int);
  // ...
};
```

You can define the calling interfaces for `draw()` and `rotate()`, but you cannot yet define their implementation. They are, therefore, declared `virtual` (the Simula67 and C++ term for "to be defined later in a class derived from this one"). They will be defined for each specific shape. Given this definition of class `shape`, you can write general functions manipulating shapes:

```
void rotate_all(shape* v[],
  int size, int angle)
// rotate all members of
// vector "v" of size "size"
// "angle" degrees
{
  for (int i = 0; i < size; i++)
    v[i]->rotate(angle);
}
```

For each shape `v[i]`, the proper `rotate()` function for the actual type of the object will be called. That "actual type" is not known at compile time.

To define a particular shape, you must say that it is a shape and specify its particular properties:

```
class circle : public shape {
// a circle is a shape
  int radius;
public:
  void draw() { /* ... */ };
  void rotate(int) {}
  // yes, the null function
};
```

A class is said to be *derived* from another class, which is then called its *base* class. Here, `circle` is derived from `shape`, and `shape` has a base class of `circle`. A derived class is said to inherit the properties of its base. In addition to such inherited properties, a derived class has its own specific properties. For example, class `circle` has the member `radius` in addition to the members `col` and `center` that it inherited from class `shape`.

Note that the new shape `center` was

*continued*

added without modifying "old code," such as the `rotate_all()` function and other shapes. The ability to extend a program by adding new variations of a basic concept (i.e., adding new derived classes given a base class) without touching old code is a major boon. Using traditional techniques, such additions require access to the source code of the system you want to extend, require understanding of the key implementation details of the old code, and carry the risk of introducing errors in the already-tested old code. Furthermore, using derived classes, improvements and bug fixes done to a base class are automatically "inherited" by every class derived from it.

I chose the "shape" example because everyone understands about shapes, not because object-oriented programming has anything particular to do with graphics. Graphics is a good area for object-oriented techniques, but most uses of such techniques in C++ have nothing to do with graphics. Other examples are compilers, operating-system kernels and device drivers, switching software, and network simulations.

In many contexts, it is important that the C++ virtual-function mechanism be nearly as efficient as a "normal" function call. The additional run-time overhead is about five memory references (depending on the machine architecture and the compiler), and the memory overhead is one word per object plus one word per virtual function per class.

C++ provides multiple inheritance (see reference 7), or the ability to derive a class from more than one direct base class. For example, if you have a class `task` representing the concept of a concurrent activity, and a class `displayed` representing the concept of something displayed on the screen, you might write:

```
class displayed_task
  : public displayed, public task
  { ... }
```

Now a `displayed_task` is really both a `displayed` and a `task`, so a `displayed_task` can be used wherever a `displayed` or a `task` is required:

```
void wait(task*,int);
// do something to a task
void update(displayed*);
// do something to a displayed

f()
{
  // make a displayed_task:
  displayed_task* dtp =
    new displayed_task(
    /* appropriate arguments */ );
  wait(ctp,10);
  // use displayed_task as a task
  update(ctp);
  // displayed_task as displayed
}
```

Naturally, the usual type-checking rules, ambiguity rules, and encapsulation mechanisms are applied to multiple inheritance to ensure the usual degree of safety and efficiency.

## Why C++?

What distinguishes C++ from other programming languages? C++ was designed under severe constraints of compatibility, internal consistency, and efficiency. No feature was included that would cause a serious incompatibility with C at the source or linker levels; would cause run-time or space overheads for a program that did not use it; would increase run time or requirements for a C program; would significantly increase the compile time compared with C; or could only be implemented by making more demands than in a traditional programming environment.

Traditional languages such as C, FORTRAN, Pascal, and Modula-2 don't provide anything comparable to C++'s features for data abstraction and object-oriented programming. This gives the C++ programmer a strong advantage when it comes to understanding, writing, and maintaining programs. It's often important that the improved structure of C++ programs be achieved without sacrificing efficiency or restricting the range of areas for which the language is suitable.

Ada provides facilities for data abstraction that may not be as elegant as C++'s but should be about as effective in actual use. But Ada doesn't provide an inheritance mechanism to support object-oriented programming, so C++ has greater expressive power in this area.

C++ is distinguished among languages that support object-oriented programming, such as Smalltalk, by a variety of factors: its emphasis on program structure; the flexibility of encapsulation mechanisms; its smooth support of a range of programming paradigms; the portability of C++ implementations; the run-time efficiency (in both time and space) of C++ code; and its ability to run without a large run-time system.

C++ is a programming language in the traditional sense and is not a complete program development system or a complete execution environment. It can be installed easily into an existing C program development or execution environment, and C++-specific tools can then be added as needed. In addition, several C++-specific environments are being built to suit specific needs (see references 8 and 9).

The emphasis on explicit static structure (as opposed to a weak type-checking, as in C, or purely dynamic type-checking, as in Smalltalk) is particularly important for projects involving many programmers and for individual programmers using large libraries written by others. C++'s strong type-checking and encapsulation mechanisms have repeatedly proven themselves by dramatically reducing integration time for larger projects. Similarly, C++ provides a good base for designing libraries with precisely defined, elegant, and statically checked interfaces.

C++ has a single, very flexible, type system. This makes it possible to use hybrid programming styles without violating the C++ type system. It also lets you choose a style of programming closely matching individual application areas. ■

REFERENCES

1. Birtwistle, Graham, et al. *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden, 1971. Chartwell-Bratt Ltd., U.K., 1980.
2. Woodward, P. M., and S. G. Bond. *Algol 68-R Users Guide*. London: Her Majesty's Stationery Office, 1974.
3. Stroustrup, Bjarne. *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.
4. Richards, Martin, and Colin Whitby-Strevens. *BCPL—The Language and its Compiler*. New York: Cambridge University Press, 1980.
5. Prosser, David F. Draft Proposed American National Standard for Information Systems—Programming Language C X3 Secretariat, CBEMA, Washington.
6. Stroustrup, Bjarne. "What Is 'Object-Oriented Programming'?" *IEEE Software Magazine*, May 1988.
7. Stroustrup, Bjarne. *The Evolution of C++: 1985-1987*. Santa Fe, NM: Proc. USENIX C++ Workshop, November 1987.
8. Linton, Mark A. "Distributed Management of a Software Database." *IEEE Software*, November 1987, pp. 70-76.
9. Stroustrup, Bjarne. *Possible Directions for C++: 1985-1987*. Santa Fe, NM: Proc. USENIX C++ Workshop, November 1987.

*Bjarne Stroustrup is the designer and original implementor of C++. He works at AT&T Bell Labs, Murray Hill, New Jersey. You can reach him on BIX as "bstroustrup."*