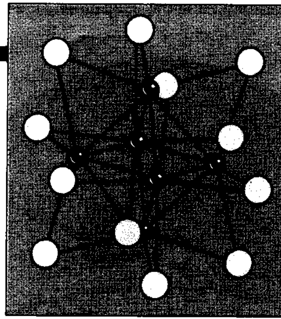


What is Object-Oriented Programming?

Bjarne Stroustrup, *AT&T Bell Laboratories*

Object-oriented has become a buzzword that implies "good" programming. But when it comes to really supporting this paradigm, not all languages are equal.

Not all programming languages can be object-oriented. Yet, claims have been made that APL, Ada, Clu, C++, Loops, and Smalltalk are object-oriented languages. I have heard discussions of object-oriented design in C, Pascal, Modula-2, and Chill. Could there somewhere be proponents of object-oriented programming in Fortran and Cobol? I think there must be.

"Object-oriented" has become a high-tech synonym for "good." Articles in the trade press contain arguments that appear to boil down to syllogisms like:

> *Ada is good; object-oriented is good; therefore, Ada is object-oriented.*

This article presents my view of what object-oriented means in the context of a general-purpose programming language. I present examples in C++, partly to introduce C++ and partly because C++ is one of the few languages that supports data abstraction, object-oriented programming, and traditional programming techniques. I do not cover issues of concurrency and hardware support for specific, higher level language constructs.

An earlier version of this article appeared in *Proc. First European Conf. on Object-Oriented Programming*, Springer-Verlag, New York, 1987, pp. 51-70.

Programming paradigms

Object-oriented programming is a technique — a paradigm for writing "good" programs for a set of problems. If the term "object-oriented language" means anything, it must mean a language that has mechanisms that support the object-oriented style of programming well.

There is an important distinction here: A language *supports* a programming style if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; in that case, the language merely

0740-7459/88/0500/0010/$01.00 ©1988 IEEE

enables programmers to use the technique. For example, you can write structured programs in Fortran and type-secure programs in C, and you can use data abstraction in Modula-2, but it is unnecessarily hard to do so because those languages do not *support* those techniques.

Support for a paradigm comes not only in the obvious form of language facilities that let you use the paradigm directly, but also in the more subtle forms of compile-time and runtime checks for unintentional deviations from the paradigm. Type checking, ambiguity detection, and runtime checks are an examples of linguistic support for paradigms. Extralinguistic facilities such as standard libraries and programming environments can also provide significant support for paradigms.

One language is not necessarily better than another because it has a feature the other does not — there are many examples to the contrary. The important issue is not how many features a language has, but that the features it does have are sufficient to support the desired programming styles in the desired application areas. Specifically, it is important that:

- All features are cleanly and elegantly integrated into the language.
- It is possible to combine features to achieve solutions that would have otherwise required extra, separate features.
- There are as few spurious and special-purpose features as possible.
- Implementing a feature does not impose significant overhead on programs that do not require it.
- A user need only know about the language subset used explicitly to write a program.

The last two principles can be summarized as "what you don't know won't hurt you." If there are any doubts about the usefulness of a feature, it is better left out. It is *much* easier to add a feature to a language than to remove or modify one that has found its way into the compiler or the literature.

Procedural. The original — and probably still the most common — programming paradigm is:

Decide which procedures you want; use the best algorithms you can find.

The focus is on procedure design — the algorithm needed to perform the desired computation. Languages support this paradigm with facilities for passing arguments to functions and returning values from functions. The literature about this paradigm is filled with discussions of how to pass arguments, how to distinguish different kinds of arguments and different kinds of functions (procedures, routines, macros, etc.), and so on.

A language does not support a techique if it takes exceptional effort or skill to write such programs.

Fortran is the original procedural language; Algol-60, Algol-68, C, and Pascal are later inventions in the same tradition.

An example of good procedural style is a square-root function. Given an argument, the function neatly produces a result. To do so, it performs a well-understood mathematical computation.

```
double sqrt(double arg)
{
   // the code for calculating a square root
}

void some_function()
{
   double root2 = sqrt(2);
   // ...
}
```

Procedural programming uses functions to create order in a maze of algorithms.

Data hiding. Over the years, the emphasis in the program design has shifted from procedure design to data organization. Among other things, this reflects an increase in program size. A set of related procedures and the data they manipulate is often called a module. The programming paradigm is:

Decide which modules you want; partition the program so that data is hidden in modules.

This paradigm is known as the data-hiding principle. When procedures do not need to be grouped with related data, the procedural style suffices. In fact, the techniques for designing good procedures are still applied, now to each procedure in a module.

The most common example of data hiding is a definition of a stack module. A good solution requires

- a user interface for the stack (for example, the functions push() and pop()),
- that the stack representation (for example, a vector of elements) can be accessed only through this user interface, and
- that the stack is initialized before its first use.

A plausible external interface for a stack module is

```
// declaration of the interface of module
// stack of characters
char pop();
void push(char);
const stack_size = 100;
```

Assuming this interface is found in a file called stack.h, the internals can be defined like this:

```
#include "stack.h"
static char v[stack_size];   // "static" means
                             // local to this
                             // file/module
```

```
static char* p = v;   // the stack is initially
                      // empty
char pop()
{
  // check for underflow and pop
}
void push(char c)
{
  // check for overflow and push
}
```

It is quite feasible to change this stack representation to a linked list. The user does not have access to the representation anyway (because *v* and *p* were declared static — that is, local to the file or module in which they were declared). Such a stack can be used like this:

```
#include "stack.h"
void some_function()
{
  char c = pop(push('c'));
  if (c != 'c') error("impossible");
}
```

As originally defined, Pascal doesn't provide satisfactory facilities for such grouping: The only way to hide a name from the rest of the program is to make it local to a procedure. This leads to strange procedure nestings and overreliance on global data.

C fares somewhat better. As shown in the example, you can define a module by grouping related function and data definitions in a single source file. The programmer can then control which names are seen by the rest of the program (a name can be seen by the rest of the program *unless* it has been declared *static*). So in C you can achieve a degree of modularity. However, there is no generally accepted paradigm for using this facility, and relying on *static* declarations is rather low-level.

One of Pascal's successors, Modula-2, goes a bit further. It formalizes the module concept by making it a fundamental construct with well-defined module declarations, explicit control of the scope of names (import/export facilities), a module-initialization mechanism, and a set of generally known, accepted usage styles.

In other words, C enables the decomposition of a program into modules, while Modula-2 supports it.

Data abstraction. Programming with modules leads to the centralization of all data of a certain type under the control of a type-manager module. If you wanted two stacks, you would define a stack-manager module with an interface like this:

```
  // stack_id is a type; no details about
  // stacks or stack_ids are known here:
class stack_id;
  // make a stack and return its identifier:
stack_id create_stack(int size);
  // call when stack is no longer needed:
destroy_stack(stack_id);
void push(stack_id, char);
char pop(stack_id);
```

This is certainly a great improvement over the traditional unstructured mess, but "types" implemented this way are clearly very different from the types built into a language.

In most important aspects, a type created through a module mechanism is different from a built-in type and enjoys inferior support: Each type-manager module must define a separate mechanism for creating variables of its type, there is no established norm for assigning object identifiers, a variable of such a type has no name known to the compiler or programming environment, and such variables do not obey the usual scope and argument-passing rules.

For example:

```
void f()
{
  stack_id s1;
  stack_id s2;

  s1 = create_stack(200);
  // Oops: forgot to create s2

  char c1 = pop(s1,push(s1,'a'));
  if (c1 != 'c') error("impossible");

  char c2 = pop(s2,push(s2,'a'));
  if (c2 != 'c') error("impossible");

    destroy(s2);
  // Oops: forgot to destroy s1
}
```

In other words, the module concept that supports the data-hiding paradigm enables data abstraction, but does not support it.

Abstract data types. Languages such as Ada, Clu, and C++ attack this problem by letting the user define types that behave in (nearly) the same way as built-in types. Such a type is often called an abstract data type, although I prefer to call it a user-defined type.* The programming paradigm becomes:

> *Decide which types you want; provide a full set of operations for each type.*

When there is no need for more that one object of a type, the data-hiding programming style using modules suffices. Arithmetic types such as rational and complex numbers are common examples of user-defined types:

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

The declaration of the complex class (the user-defined type) specifies the representation of a complex number and the set of operations on a complex number. The representation is *private*; that is, *re* and *im* are accessible only to the functions specified in the declaration of class complex. Such functions can be defined like this:

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

and used like this:

```
complex a = 2.3;
complex b = 1/a;
complex c = a+b*complex(1,2.3);
// ...
c = -(a/b)+2;
```

Most, but not all, modules are better ex-

*As Doug McIlroy has said, "Those types are not 'abstract,' they are as real as *int* and *float*." Another definition of abstract data types would require a mathematical "abstract" specification of all types (both built-in and user-defined). What is referred to as types in this article would, given such a specification, be concrete specifications of such truly abstract entities.

pressed as user-defined types. When the programmer prefers to use a module representation, even when a proper facility for defining types is available, he can declare a type that has only a single object of that type. Alternatively, a language might provide a module concept in addition to and distinct from the class concept.

Problems. A user-defined type defines a sort of black box. Once it has been defined, it does not really interact with the rest of the program. The only way to adapt it to new uses is to modify its definition. This is often too inflexible.

Consider defining a type *shape* for use in a graphics system. Assume for the moment that the system has to support circles, triangles, and squares. Assume also that you have some classes:

```
class point{ /* ... */ };
class color{ /* ... */ };
```

You might define a shape like this:

```
enum kind { circle, triangle, square };

class shape {
  point center;
  color col;
  kind k;
  // representation of shape
public:
  point where()   { return center; }
  void move(point to) { center = to; draw(); }
}
  void draw();
  void rotate(int);
  // more operations
};
```

The type field, *k*, is used by operations such as draw() and rotate() to determine what shape they are dealing with (in a Pascal-like language, you might use a variant record with tag *k*). The function draw() might be defined like this:

```
void shape::draw()
{
  switch (k) {
  case circle:
    // draw a circle
    break;
  case triangle:
    // draw a triangle
    break;
  case square:
    // draw a square
  }
}
```

This is a mess. It requires that functions such as draw() know about all the kinds of shapes there are. Therefore, the code for any such function must by modified each time a new shape is added to the system.

If you define a new shape, every operation on a shape must be examined and (possibly) modified. You cannot add a new shape to a system unless you have access to the source code for every operation. Because adding a new shape involves touching the code of every important operation on shapes, it can require great skill and may introduce bugs into the code that handles the older shapes.

Also, your choice of how you represent particular shapes can be severely cramped by the requirement that at least some of their representation fit into the typically fixed-sized framework presented by the definition of the general type *shape*.

The problem is that there is no distinction between the general properties of any shape (a shape has a color, it can be drawn, etc.) and the properties of a specific shape (a circle is a shape that has a radius, is drawn by a circle-drawing function, etc.).

Object-oriented programming. The ability to express this distinction and take advantage of it defines object-oriented programming. A language with constructs that let you express and use this distinction supports object-oriented programming. Other languages don't.

The Simula inheritance mechanism provides a solution. 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);
  // ...
};
```

The functions marked virtual are those for which the calling interface can be defined, but the implementation cannot be defined except for a specific shape. ("Virtual" is the Simula and C++ term for "may be redefined later in a class derived from this one.") Given this definition, we can write general functions to manipulate 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);
}
```

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

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

In C++, the circle class is said to be *derived* from the shape class, and the shape class is said to be a *base* of the circle class. Another terminology calls circle a subclass and shape a superclass.

The programming paradigm is:

Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance.

Where there is no such commonality, data abstraction suffices. How much types have in common so that the commonality can be exploited using inheritance and virtual functions is the litmus test of the applicability of object-oriented programming.

In some areas, such as interactive graphics, there is clearly enormous opportunity for object-oriented programming. In other areas, such as classical arithmetic types and the computations based on them, there appears to be hardly any need for more than data abstraction.*

Finding commonality among types in a system is not a trivial process. How much commonality can be exploited depends on how the system is designed. Commonality must be actively sought when the system is designed, both by designing classes specifically as building blocks for other types and by examining classes to see if they have similarities that can be exploited in a common base class.

*However, more advanced mathematics may benefit from the use of inheritance: Fields are specializations of rings; vector spaces a special case of modules.

Nygaard[1] and Kerr[2] explain what object-oriented programming is without recourse to specific language constructs; Cargill has written a case study in object-oriented programming.[3]

Supporting data abstraction

Programming with data abstraction is supported with facilities both to define a set of operations for a type and to restrict access of objects of that type to that operation set. However, once that is done the programmer soon finds that language refinements are needed to define and use the new types conveniently.

Initialization and cleanup. When a type's representation is hidden, some mechanism must be provided for a user to initialize variables of that type. A simple solution is to require a user to call some function to initialize a variable before using it. For example,

```
class vector {
  int sz;
  int* v;
public:
  void init(int size);  // call init to initialize
                        // sz and v before the
                        // first use of a vector
  // ...
};
vector v;
// don't use v here
v.init(10);
// use v here
```

This is error-prone and inelegant. A better solution is to allow the designer of a type to provide a distinguished function to do the initialization. Such a function makes allocation and initialization of a variable a single operation (often called instantiation) instead of two operations. Such an instantiation function is often called a constructor.

In cases where constructing object types is nontrivial, it is often necessary to provide a complementary operation to clean up objects after their last use. In C++, a cleanup function is called a destructor. Consider a *vector* type:

```
class vector {
  int sz;       // number of elements
  int* v;       // pointer to integers
public:
  vector(int); // constructor
  ~vector();  // destructor
  int& operator[](int index);  // subscript
                               // operator
};
```

The *vector* constructor can be defined to allocate space like this:

```
vector::vector(int s)
{
  if (s<=0) error("bad vector size");
  sz = s;
  v = new int[s]; // allocate an array
                  // of "s" integers
}
```

The *vector* destructor frees storage:

```
vector::~vector()
{
  delete v;  // deallocate the memory
             // pointed to by v
}
```

C++ does not support garbage collection. It compensates for this by letting a type maintain its own storage management without user intervention. While this is a common use for the constructor/destructor mechanism, many uses of this mechanism are unrelated to storage management.

Assignment and initialization. Controlling the construction and destruction of objects is sufficient for many types, but not for all. Sometimes, it is also necessary to control copy operations. Consider the vector class:

```
vector v1(100);
vector v2 = v1; // make a new vector v2
                // initialized to v1
v1 = v2;        // assign v2 to v1
```

It must be possible to define the meaning of the initialization of *v2* and its assignment to *v1*. It should also be possible to prohibit such copy operations; preferably both alternatives should be available. For example:

```
class vector {
  int* v;
  int sz;
public:
  // ...
  void operator=(vector&); // assignment
  vector(vector&);          // initialization
};
```

specifies that user-defined operations should be used to interpret *vector* assignment and initialization. Assignment might be defined like this:

```
vector::operator=(vector& a)
  // check size and copy elements
{
  if (sz != a.sz)
    error("bad vector size for =");
    for (int i = 0; i<sz; i++) v[i] = a.v[i];
}
```

Since the assignment operation relies on the old value of the vector being assigned to, the initialization operation *must* be different. For example:

```
vector::vector(vector& a)
  // initialize a vector from another vector
{
  sz = a.sz;           // same size
  v = new int[sz];  // allocate element array
    // copy elements:
    for (int i = 0; i<sz; i++) v[i] = a.v[i];
}
```

In C++, a constructor *X(X&)* defines all initialization of objects of type *X* with another object of type *X*. In addition to explicit initialization, constructors of the form *X(X&)* are used to handle arguments passed by value and function-return values.

In C++, assignment of an object of class X can be prohibited by declaring the assignment operation private:

```
class X {
  void operator=(X&); // only members
                      // of X can
  X(X&);              // copy an X
  ...
public:
  ...
};
```

Ada does not support constructors, destructors, assignment overloading, or user-defined control of argument passing and function return. This lack of support severely limits the class of types that can be defined and forces the programmer back to data-hiding techniques: The user must design and use type-manager modules instead of proper types.

Parameterized types. Why would you want to define a vector of integers anyway? Typically, a user needs a vector of elements of some type unknown to the writer of the type *vector*. Consequently the *vector* type ought to be expressed so it takes the element type as an argument:

```
class vector<class T> {
    // vector of elements of type T
    T* v;
    int sz;
public:
    vector(int s)
    {
      if (s <= 0) error("bad vector size");
      v = new T[sz = s]; // allocate an array
                         // of "s" "T"s
    }
    T& operator[] (int i);
    int size() { return sz; }
    // ...
};
```

Vectors of specific types can now be defined and used:

```
vector<int> v1(100);  // v1 is a vector
                      // of 100 integers
vector<complex> v2(200); // v2 is a vector
                         // of 200 complex
                         // numbers
v2[i] = complex(v1[x],v1[y]);
```

Ada and Clu support parameterized types. Unfortunately, C++ does not; the notation used here is still experimental. When they are needed, parameterized classes are faked with macros. There need not be any runtime overheads compared with a class where all types involved are specified directly.

Typically, a parameterized type will have to depend on at least some aspect of a type parameter. For example, some of the vector operations must assume that assignment is defined for objects of the parameter type. How can you ensure that? One way is to require that the designer of the parameterized class state the dependency. For example, "T must be a type for which = is defined." A better way is not to require this — or to take a specification of an argument type as a partial specification. A compiler can detect if a a missing operation has been applied and give an error message such as

```
cannot define
vector<non_copy>::operator[](non_copy&):
    type non_copy does not have operator=
```

This technique lets you define types where the dependency on attributes of a parameter type is handled at the level of the individual operation of the type. For example, you might define a vector with a sort operation. The sort operation might use <, ==, and = on objects of the parameter type. It would still be possible to define vectors of a type for which < was not defined, as long as the vector-sorting operation was not actually invoked.

A problem with parameterized types is that each instantiation creates an independent type. For example, the type *vector<char>* is unrelated to the type *vector<complex>*. Ideally you would like to be able to express and use the commonality of types generated from the same parameterized type. For example, both *vector<char>* and *vector<complex>* have a size() function that is independent of the parameter type. It is possible, but not easy, to deduce this from the definition of class vector and then let size() be applied to any vector. An interpreted language or a language that supports both parameterized types and inheritance has an advantage here.

Exception handling. As programs grow, and especially when libraries are used extensively, standards for handling errors (or "exceptional circumstances") become important.

Ada, Algol-68, and Clu each support a standard way to handle exceptions. Unfortunately, C++ does not. When necessary, exceptions are faked using pointers to functions, exception objects, error states, and the C library's signal and longjmp facilities. This is not satisfactory because it fails to provide even a standard framework for error handling.

Consider the vector example again. What should be done when an out-of-range index value is passed to the subscript operator? The designer of the vector class should be able to provide a default behavior for this:

```
class vector {
    ...
    except vector_range {
        // define an exception called
        // vector_range and specify default
        // code for handling it
        error("global: vector range error");
        exit(99);
    }
}
```

Instead of calling an error function, *vector::operator[]()* can invoke the exception-handling code:

```
int& vector::operator[](int i)
{
    if (0<i || sz<=i) raise vector_range;
    return v[i];
}
```

This will cause the call stack to be unraveled until an exception handler for *vector_range* is found and executed.

An exception handler may be defined for a specific block:

```
void f() {
  vector v(10);
  try {    // errors here are handled
           // by the local exception
           // handler defined below
  // ...
  int i = g(); // g might cause a range error
               // using some vector
  v[i] = 7;    // potential range error
  }
  except {
    vector::vector_range:
      error("f(): vector range error");
      return;
  }
        // errors here are handled by the
        // global exception handler
        // defined in vector
  int i = g();  // g might cause a range
                // error using some vector
  v[i] = 7;     // potential range error
}
```

There are many ways to define exceptions and the behavior of exception handlers. The facility sketched here resembles the ones found in Modula-2+. This style of exception handling can be implemented so that code is not executed unless an exception is raised (except possibly for some initialization code). It can also be ported across most C implementations by using setjmp() and longjmp() (see the C library manual for your system).

Could exceptions, as defined above, be completely faked in a language such as C++? Unfortunately, no. The snag is that when an exception occurs, the runtime stack must be unraveled up to a point where an exception handler is defined. To do this properly in C++ involves invoking destructors defined in the scopes involved. This is not done by a C longjmp() and cannot in general be done by the user.

Coercions. User-defined coercions, such as the one from floating-point numbers to complex numbers implied by the constructor *complex(double)*, have proven unexpectedly useful in C++. Such coercions can be applied explicitly or the pro-

grammer can rely on the compiler to add them implicitly where necessary and unambiguous:

```
complex a = complex(1);
complex b = 1;  // implicit:
                // 1 -> complex(1)
a = b+complex(2);
a = b+2;  // implicit:
          // 2 -> complex(2)
```

Coercions were introduced into C++ because mixed-mode arithmetic is the norm in languages for numerical work and because most user-defined types used for calculation (of matrices, character strings, and machine addresses) have natural mappings to and from other types.

One use of coercions has proven especially useful in organizing programs:

```
complex a = 2;
complex b = a+2;  // interpreted as
                  //operator+
                  //(a,complex(2))
b = 2+a;          //interpreted as
                  //operator+
                  //(complex(2),a)
```

Only one function is needed to interpret + operations, and the two operands are handled identically by the type system. Furthermore, class complex is written without any need to modify the concept of integers to enable the smooth and natural integration of the two concepts.

This is in contrast to a "pure" object-oriented system, where the operations would be interpreted like this:

```
a+2; // a.operator+(2)
2+a; // 2.operator+(a)
```

making it necessary to modify the integer class to make *2+a* legal.

You should avoid modifying existing code as much as possible when adding new facilities to a system. Typically, object-oriented programming offers superior facilities for adding to a system without modifying existing code. In this case, however, data-abstraction facilities provide a better solution.

Iterators. A language that supports data abstraction must provide a way to define control structures.[4] In particular, users need a mechanism to define loops over the elements contained in an object of some user-defined type, without forcing them to depend on implementation details of the user-defined type. Given a sufficiently powerful mechanism for defining new types and the ability to overload operators, this can be handled without a separate mechanism for defining control structures.

For a vector, defining an iterator is not necessary because an ordering is available to a user through the indices. I'll define one anyway, to demonstrate the technique.

There are several iterator styles. My favorite relies on overloading the function application operator ():*

```
class vector_iterator {
  vector& v;
  int i;
public:
  vector_iterator(vector& r) { i = 0; v = r; }
  int operator() ()
    { return i<v.size() ? v.elem(i++) : 0;}
};
```

A *vector_iterator* type can now be declared and used for a vector:

```
vector v(sz);
vector_iterator next(v);
int i;
while (i=next()) print(i);
```

More than one iterator can be active for a single object at one time, and a type may have several different iterator types defined for it so different kinds of iteration can be performed. An iterator is a rather simple control structure. More general mechanisms can also be defined. For example, the C++ standard library provides a coroutine class.[5]

For many container types, such as *vector*, you can avoid introducing a separate iterator type by defining an iteration mechanism as part of the type itself. A *vector* type might be defined to have a "current element":

```
class vector {
  int* v;
  int sz;
  int current;
public:
  // ...
  int next()
  { return (current++<sz) ? v[current] : 0; }
  int prev()
  { return (0<--current) ? v[current] : 0; }
};
```

*This style also relies on the existence of a distinct value to represent end of iteration. Often, in particular for C++ pointer types, 0 can be used.

Then the iteration can be performed like this:

```
vector v(sz);
int i;
while (i=v.next()) print(i);
```

This solution is not as general as the iterator solution, but it avoids overhead in the important special case where only one kind of iteration is needed and where only one iteration at a time is needed for a vector.

If necessary, you can apply a more general solution in addition to this simple one. The simple solution requires more foresight from the designer of the container class than does the iterator solution. The iterator-type technique can also be used to define iterators that can be bound to several different container types, thus providing a mechanism for iterating over different container types with a single iterator type.

Implementation issues. Support for data abstraction is primarily provided in the form of language features implemented by a compiler. However, parameterized types are best implemented with support from a linker with some knowledge of the language semantics, and exception handling requires support from the runtime environment. Both can be implemented to meet the strictest criteria for both compile-time speed and efficiency without compromising generality or programmer convenience.

As the power to define types increases, programs will depend increasingly on types from libraries (and not just those described in the language manual). This naturally puts a greater demand on facilities to express what is inserted into or retrieved from a library, for finding out what a library contains, for determining what parts of a library are actually used by a program, and so on.

For a compiled language, facilities to calculate the minimal compilation necessary after a change are important. It is essential that the linker/loader can bring a program into memory for execution without also bringing in a lot of related but unused code. In particular, a library/linker/loader system that includes the code for every operation on a type in the executable pro-

gram just because the programmer used one or two operations on the type is worse than useless.

Supporting object-oriented programming

The basic support functions a programmer needs to write object-oriented programs are a class mechanism with inheritance and a mechanism that lets calls of member functions depend on the actual object type (when the actual type is unknown at compile time).

The design of the member-function calling mechanism is critical. In addition, facilities that support data-abstraction techniques are important because the arguments for data abstraction and for its refinements to use types elegantly are equally valid where support for object-oriented programming is available.

The success of both techniques hinges on the design of types and on the ease, flexibility, and efficiency of such types. Object-oriented programming simply lets user-defined types be far more flexible and general than the ones designed using only data-abstraction techniques.

Calling mechanisms. The key language facility to support object-oriented programming is the mechanism by which a member function is invoked for an object. For example, given pointer *p*, how is a call *p→f(arg)* handled? There is a range of choices.

In languages such as C++ and Simula, where static type checking is used extensively, the type system can select between different calling mechanisms. In C++, there are two alternatives:

1. Normal function call: The member function to call is determined at compile time (through a lookup in the compiler's symbol tables) and called with the standard function-call mechanism, with an argument added to identify the object for which the function is called. When the standard function call is not efficient enough, the programmer can declare a function to be in-line and the compiler will try to expand its body in-line. This lets you achieve the efficiency of a macro expansion without compromising the standard function semantics. This optimization is equally valuable as a support for data abstraction.

2. Virtual function call: The function called depends on the object type, which usually cannot be determined until runtime. Typically, the pointer *p* will be of some base class *B* and the object will be an object of some derived class *D*. The call mechanism must look into the object and find some information placed there by the compiler to determine which function *f* is to be called. Once that function, say *D::f*, is found, it is called with the mechanism described above. At compile time, the name *f* is converted into an index to a table containing pointers to functions. This virtual-call mechanism can essentially be made as efficient as the normal function-call mechanism. In the standard C++ implementation, only five additional memory references are used. In cases where the actual type can be deduced at compile time, even this overhead is eliminated and in-lining can be used. Such cases are quite common and important.

In languages with weak static type checking, a third, more elaborate alternative must be used. In a language like Smalltalk, a list of the names of all member functions (called methods) of a class are stored so they can be found at runtime:

3. Method invocation: The appropriate table of method names is first found by examining the object that *p* points to. In this table (or set of tables), the string *f* is looked up to see if the object has an *f*(). If an *f*() is found, it is called; otherwise, some error handling takes place. This lookup differs from the lookup done at compile time in a statically checked language because the method invocation uses a method table for the actual object.

A method invocation is inefficient compared with a virtual function call, but it is more flexible. Since static type checking of arguments typically cannot be done for a method invocation, the use of methods must be supported by dynamic type checking.

Type checking. The shape example earlier showed the power of virtual functions. What else does a method-invocation mechanism do for you? It lets you invoke any method for any object.

This ability lets the designer of a general-purpose library push the responsibility for handling types onto the user. Naturally this simplifies library design. For example:

```
class stack {   // assume class any has a
                // member next
    any* v;
    void push(any* p)
    {
        p->next = v;
        v = p;
    }
    any* pop()
    {
        if (v == 0) return error_obj;
        any* r = v;
        v = v->next;
        return r;
    }
};
```

It becomes the user's responsibility to avoid type mismatches like this:

```
stack<any*> cs;
cs.push(new Saab900);
cs.push(new Saab37B);
plane* p = (plane*)cs.pop();
p->takeoff(); p = (plane*)cs.pop();
p->takeoff();
    // Oops! Runtime error:
    //A Saab 900 is a car.
    // A car does not have a takeoff method.
```

An attempt to use a car as a plane will be detected by the message handler and an appropriate error handler will be called. However, that is only a consolation when the user is also the programmer. The absence of static type checking makes it difficult to guarantee that errors of this class are not present in systems delivered to end users. Naturally, a language designed with methods and without static types can express this example with fewer keystrokes.

The combination of parameterized classes and virtual functions approaches the flexibility, ease of design, and ease of use that characterizes libraries designed with method lookup, without relaxing static type checking or incurring measurable runtime overhead (in time or space). For example:

```
stack<plane*> cs;
cs.push(new Saab900);
    //Compile time error: type mismatch:
    //car* passed, plane* expected
cs.push(new Saab37B);
plane* p = cs.pop();
p->takeoff(); //fine: a Saab 37B
              // is a plane
p = cs.pop();
p->takeoff();
```

The use of static type checking and virtual function calls leads to a somewhat different style of programming than does dynamic type checking and method invocation. For example, a Simula or C++ class specifies a fixed interface to a set of objects (of any derived class), while a Smalltalk class specifies an initial set of operations for objects (of any subclass). In other words, a Smalltalk class is a minimal specification and the user is free to try operations not specified, while a C++ class is an exact specification and only operations specified in the class declaration are guaranteed to be accepted by the compiler.

Inheritance. Consider a language that has some form of method lookup without an inheritance mechanism. Does that language support object-oriented programming? I think not.

Clearly, you could do interesting things with the method table to adapt the objects' behavior to suit conditions. However, to avoid chaos there must be some systematic way to associate methods and the data structures they assume for their object representation. To let a object's user know what kind of behavior to expect, there would also have to be some standard way to express what is common to the different behaviors the object might adopt. This systematic, standard way is an inheritance mechanism.

Consider a language that has an inheritance mechanism without virtual functions or methods. Does that language support object-oriented programming? I think not: The shape example does not have a good solution in such a language.

However, such a language would be more powerful than a plain data-abstraction language. This contention is supported by the observation that many Simula and C++ programs are structured using class hierarchies without virtual functions. The ability to express commonality (factoring) is an extremely powerful tool. For example, the problems associated with the need to have a common representation of all shapes could be solved; no union would be needed.

However, in the absence of virtual functions, the programmer would have to resort to using type fields to determine actual types of objects, so the problems with the code's lack of modularity would remain.*

This implies that class derivation (subclassing) is an important programming tool in its own right. While it can be used to support object-oriented programming, it has wider uses. This is particularly true if you associate inheritance in object-oriented programming with the idea that a base class expresses a general concept of which all derived classes are specializations. This idea captures only part of the expressive power of inheritance, but it is strongly encouraged by languages where every member function is virtual (or a method).

Given suitable controls over what is inherited,[6,7] class derivation can be a powerful tool for creating new types. Given a class, derivation can be used to add and subtract features. The relation of the resulting class to its base cannot always be completely described in terms of specialization; factoring is a better term.

Derivation is another programmer's tool and there is no foolproof way to predict how it is going to be used — and it is too early (even after 20 years of Simula) to tell which uses are simply misuses.

Multiple inheritance. When class *A* is a base of class *B*, *B* inherits the attributes of *A*; that is, *B* is an *A* in addition to whatever else it might be. Given this explanation, it seems obvious that it might be useful to have class *B* inherit from two base classes, *A1* and *A2*. This is called multiple inheritance.[8]

An example of multiple inheritance are two library classes, the displayed class and the task class, that respectively represent objects under the control of a display manager and coroutines under the control of a scheduler. A programmer could then create classes such as

```
class my_displayed_task
: public displayed, public task {
  // my stuff
};

class my_task
: public task { // not displayed
  // my stuff
};

class my_displayed
: public displayed { // not a task
  // my stuff
};
```

*This is the problem with Simula's Inspect statement and the reason it does not have a counterpart in C++.

With single inheritance, only two of these three choices are open to the programmer. This leads to code replication or loss of flexibility — and typically both. In C++, this example can be handled with no significant overhead (in time or space), compared to single inheritance, and without sacrificing static type checking.[9]

Ambiguities are detected at compile time:

```
class A { public: f(); ... };
class B { public: f(); ... };
class C : public A, public B { ... };

void g() {
    C* p;
    p->f(); // error: ambiguous
}
```

In this capability, C++ differs from the object-oriented Lisp dialects that support multiple inheritance. In these Lisp dialects, ambiguities are resolved by considering the order of declarations significant, by considering objects of the same name in different base classes identical, or by combining methods of the same name in base classes into a more complex method of the highest class.

In C++, you would typically resolve the ambiguity by adding a function:

```
class C : public A, public B {
public:
  f()
  {
    // C's own stuff
    A::f();
    B::f();
  }
  ...
}
```

In addition to this fairly straightforward concept of independent, multiple inheritance, there appears to be a need for a more general mechanism to express dependencies between classes in a multiple-inheritance lattice. In C++, the requirement that a subobject be shared in a class object is expressed through the mechanism of a virtual base class:

```
class W { ... };
class Bwindow   // window with border
```

```
    : public virtual W
    { ... };

class Mwindow   // window with menu
    : public virtual W
    { ... };

class BMW       // window with border
                // and menu
    : public Bwindow, public Mwindow
    { ... };
```

Here, the single window subobject is shared by the Bwindow and Bwindow subobjects of a BMW. The Lisp dialects use method combinations to ease programming using such complicated class hierarchies. C++ does not.

Encapsulation. Consider a class member (data or function) that must be protected from unauthorized access. What choices are reasonable to delimit the set of functions that may access that member?

The obvious answer for a language supporting object-oriented programming is "all operations defined for this object," or all member functions. A hidden implication of this answer is that there cannot be a complete and final list of all functions that may access the protected member since you can always add another by deriving a new class from the protected member's class and then defining a member function of that derived class. This approach combines a large degree of protection from accidents (it's not easy to define a new derived class by accident) with the flexibility needed for tool building using class hierarchies (you can grant yourself access to protected members by deriving a class).

Unfortunately, the obvious answer for a language supporting data abstraction is different: "List the functions that need access in the class declaration." There is nothing special about these functions; they need not be member functions.

A nonmember function with access to private class members is called a *friend* in C++. Class Complex, above, was defined using friend functions. It is sometimes important that a function may be specified as a friend in more than one class. Having the full list of members and friends available is a great advantage when you are trying to understand the behavior of a type and especially when you want to modify it.

Here is an example that demonstrates part of the range of choices for encapsulation in C++:

```
class B {
        // class members are
        // default private
    int i1;
    void f1();
protected:
    int i2;
    void f2();
public:
    int i3;
    void f3();
friend void g(B*);  // any function can be
                    // designated as a friend
};
```

Private and protected members are not generally accessible:

```
void h(B* p)
{
    p->f1();  // error: B::f1 is private
    p->f2();  // error: B::f2 is protected
    p->f3();  // fine: B::f1 is public
}
```

Protected, but not private, members are accessible to members of a derived class:

```
class D : public B {
public:
    void g()
    {
        f1();  // error: B::f1 is private
        f2();  // fine: B::f2 is protected,
               // but D is derived from B
        f3();  // fine: B::f1 is public
    }
};
```

Friend functions have access to private and protected members just like member functions:

```
void g(B* p)
{
    p->f1();  // fine: B::f1 is private,
              // but g() is a friend of B
    p->f2();  // fine: B::f2 is protected,
              // but g() is a friend of B
    p->f3();  // fine: B::f1 is public
}
```

The importance of encapsulation issues increases dramatically as program size increases, and as the number and geographical dispersion of its users expands. For a detailed discussion of encapsulation issues, see Snyder[6] and Stroustrup.[7]

Implementation issues. Support for object-oriented programming is provided primarily by the runtime system and the programming environment. Part of the reason is that object-oriented programming builds on the language improvements already provided to support data abstraction, so relatively few additions are needed.

This assumes that an object-oriented language does indeed support data abstraction. However, the support for data abstraction is often deficient in such languages. Conversely, languages that support data abstraction are typically deficient in their support of object-oriented programming.

Object-oriented programming further blurs the distinction between a programming language and its environment. Because more powerful special- and general-purpose user-defined types can be defined, they pervade user programs. This requires further development of the runtime system, library facilities, debuggers, performance measuring, monitoring tools, and so on. Ideally, these are integrated into a unified programming environment, of which Smalltalk is the best example.

Limits to perfection

To claim to be general-purpose, a language that is designed to exploit the techniques of data hiding, data abstraction, and object-oriented programming must also

- run on traditional machines,
- coexist with traditional operating systems,
- compete with traditional programming languages in runtime efficiency, and
- cope with every major application area.

This means that facilities must be available for effective numerical work (floating-point arithmetic without overhead that would make Fortran attractive), and memory must be accessible so that device drivers can be written. It must also be possible to write calls that conform to the (often rather strange) standards required for operating-system interfaces. In addition, it should be possible to call functions written in other languages from a object-oriented language and for functions written in the object-oriented language to be called from a program written in another language.

It also means that an object-oriented language cannot rely completely on mechanisms that cannot be efficiently implemented on a traditional architecture and still expect to be used as a general-purpose language. A very general implementation of method invocation can be a liability unless there are alternative ways of requesting a service.

Similarly, garbage collection can become a performance and portability bottleneck. Most object-oriented programming languages use garbage collection to simplify the programmer's task and to reduce the complexity of the language and its compiler. However, it ought to be possible to use garbage collection in non-critical areas while retaining control of storage use in areas where it matters. As an alternative, it is feasible to have a language without garbage collection and then provide sufficient expressive power to enable the design of types that maintain their own storage. C++ is an example of this.

Exception handling and concurrency are other potential problems. Any feature that is best implemented with help from a linker is likely to become a portability problem.

The alternative to having low-level features in a language is to handle major application areas using separate low-level languages.

Object-oriented programming is programming using inheritance. Data abstraction is programming using user-defined types. With few exceptions, object-oriented programming can and ought to be a superset of data abstraction.

These techniques need proper language support to be effective. Data abstraction needs support primarily in language features; object-oriented programming needs more support in the programming environment. To be considered general-purpose, a language must let you use traditional hardware effectively. ❖

Acknowledgments

An earlier version of this article was presented to the Association of Simula Users meeting in Stockholm in August 1986. The discussions there caused many improvements in both style and content. Brian Kernighan and Ravi Sethi made many constructive comments. Also, thanks to all who helped shape C++.

AN OUTSTANDING SEMINAR BY THE INTERNATIONALLY RECOGNIZED AUTHORITY ON...

Software Cost Estimation Using

COCOMO

(COnstructive COst MOdel)

Special Feature: A Full Description of the Recently Developed Ada COCOMO Model

- COCOMO vs. Other Cost Models/Techniques
- Basic, Intermediate, Detailed COCOMO—Features/Applications
- Tailoring COCOMO
- COCOMO Extensions—Incremental Development, Acquisition management

PRESENTED BY:

DR. BARRY W. BOEHM

LOS ANGELES June 9-10, 1988

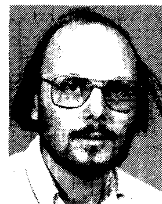
WASHINGTON, DC June 13-14, 1988

For Information Call: (213) 534-4871

Reader Service Number 4

References

1. K. Nygaard, "Basic Concepts in Object-Oriented Programming," *SIGPlan Notices*, Oct. 1986, pp. 128-132.
2. R. Kerr, "Object-Based Programming: A Foundation for Reliable Software," *Proc. 14th Simula Users' Conf.*, Simula Information, Oslo, Norway, 1986, pp. 159-165; a short version is "A Materialistic View of the Software 'Engineering' Analogy, *SIGPlan Notices*, March 1987, pp 123-125.
3. T. Cargill, "PI: A Case Study in Object-Oriented Programming," *SIGPlan Notices*, Nov. 1986, pp. 350-360.
4. B. Liskov et al., "Abstraction Mechanisms in Clu," *Comm. ACM*, Aug. 1977, pp. 564-576.
5. J. Shopiro, "Extending the C++ Task System for Real-Time Applications," *Proc. Usenix C++ Workshop*, Usenix, Santa Monica, Calif., 1987, pp. 77-94.
6. A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *SIGPlan Notices*, Nov. 1986, pp. 38-45.
7. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.
8. D. Weinreb and D. Moon, *Lisp Machine Manual*, Symbolics, Cambridge, Mass., 1981.
9. B. Stroustrup, "Multiple Inheritance for C++," *Proc. Spring European Unix Users Group Conf.*, EEUG, London, 1987.



Bjarne Stroustrup is the designer and original implementer of C++. His research interests include distributed systems, operating systems, simulation, programming methodology, and programming languages.

Stroustrup received an MS in mathematics and computer science from the University of Aarhus and a PhD in computer science from Cambridge University. He is a distinguisehd member of the Computer Science Research Center and is a member of IEEE and ACM.

Address questions about this article to the author at AT&T Bell Laboratories, Rm. 2C-324, 600 Mountain Ave., Murray Hill, NJ 07974.