What is ‘‘Object-Oriented Programming’’? (1991 revised version)

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

‘‘Object-Oriented Programming’’ and ‘‘Data Abstraction’’ have become very common terms. Unfortunately, few people agree on what they mean. I will offer informal definitions that appear to make sense in the context of languages like Ada, C++, Modula-2, Simula, and Smalltalk. The general idea is to equate ‘‘support for data abstraction’’ with the ability to define and use new types and equate ‘‘support for object-oriented programming’’ with the ability to express type hierarchies. Features necessary to support these programming styles in a general purpose programming language will be discussed. The presentation centers around C++ but is not limited to facilities provided by that language.

1 Introduction

Not all programming languages can be ‘‘object oriented’’. Yet claims have been made to the effect that APL, Ada, Clu, C++, CLOS, and Smalltalk are object-oriented programming languages. I have heard discussions of object-oriented design in C, Pascal, Modula-2, and CHILL. As predicted in the original version of this paper, proponents of object-oriented Fortran and Cobol programming are now appearing. ‘‘Object-oriented’’ has in many circles become a high-tech synonym for ‘‘good’’, and when you examine discussions in the trade press, you can find arguments that appear to boil down to syllogisms like:

Ada is good
Object oriented is good

Ada is object oriented

We simply *must* be more careful with our concepts and logic.

This paper presents one view of what ‘‘object oriented’’ ought to mean in the context of a general purpose programming language.

§2 Distinguishes ‘‘object-oriented programming’’ and ‘‘data abstraction’’ from each other and from other styles of programming and presents the mechanisms that are essential for supporting the various styles of programming.
§3 Presents features needed to make data abstraction effective.
§4 Discusses facilities needed to support object-oriented programming.
§5 Presents some limits imposed on data abstraction and object-oriented programming by traditional hardware architectures and operating systems.

The first version of this paper was presented at the Association of Simula Users’ meeting in Stockholm, August 1986. Later, a version was presented as an invited talk at the first European Conference on Object-Oriented Programming in Paris and published by Springer Verlag. It also appeared in the May 1988 issue of IEEE Software Magazine. This version has been revised to reflect the latest version of C++ as described in *The Annotated C++ Reference Manual*[5] approved by the ANSI C++ committee (X3J16) as the basis of formal standardization.

Examples will be presented in C++. The reason for this is partly to introduce C++ and partly because C++ is one of the few languages that supports both data abstraction and object-oriented programming in addition to traditional programming techniques. Issues of concurrency and of hardware support for specific higher-level language constructs are ignored in this paper.

2 Programming Paradigms

Object-oriented programming is a technique for programming – a paradigm for writing ‘‘good’’ programs for a set of problems. If the term ‘‘object-oriented programming language’’ means anything it must mean a programming language that provides mechanisms that support the object-oriented style of programming well.

There is an important distinction here. A language is said to *support* a style of programming if it provides facilities that makes 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; it merely *enables* the technique to be used. For example, you can write structured programs in Fortran, write type-secure programs in C, and use data abstraction in Modula-2, but it is unnecessarily hard to do because these languages do not support those techniques.

Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in the more subtle form of compile-time and/or run-time checks against unintentional deviation from the paradigm. Type checking is the most obvious example of this; ambiguity detection and run-time checks can be used to extend linguistic support for paradigms. Extra-linguistic facilities such as standard libraries and programming environments can also provide significant support for paradigms.

A language is not necessarily better than another because it possesses a feature the other does not. There are many example to the contrary. The important issue is not so much what features a language possesses but that the features it does possess are sufficient to support the desired programming styles in the desired application areas:

[1] All features must be cleanly and elegantly integrated into the language.
[2] It must be possible to use features in combination to achieve solutions that would otherwise have required extra separate features.
[3] There should be as few spurious and ‘‘special purpose’’ features as possible.
[4] A feature should be such that its implementation does not impose significant overheads on programs that do not require it.
[5] A user need only know about the subset of the language explicitly used 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 compilers or the literature.

I will now present some programming styles and the key language mechanisms necessary for supporting them. The presentation of language features is not intended to be exhaustive.

Procedural Programming

The original (and probably still the most commonly used) programming paradigm is:

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

The focus is on the design of the processing, the algorithm needed to perform the desired computation. Languages support this paradigm by facilities for passing arguments to functions and returning values from functions. The literature related to this way of thinking is filled with discussion of ways of passing arguments, ways of distinguishing different kinds of arguments, different kinds of functions (procedures, routines, macros, ...), etc. Fortran is the original procedural language; Algol60, Algol68, C, and Pascal are later inventions in the same tradition.

A typical example of ‘‘good style’’ is a square root function. Given an argument, it produces a result.

To do this, 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);
    // ...
}
```

From a program organization point of view, functions are used to create order in a maze of algorithms.

Data Hiding

Over the years, the emphasis in the design of programs has shifted away from the design of procedures towards the organization of data. Among other things, this reflects an increase in program size. A set of related procedures with the data they manipulate is often called a *module*. The programming paradigm becomes:

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

This paradigm is also known as the ‘‘data hiding principle’’. Where there is no grouping of procedures with related data the procedural programming style suffices. In particular, the techniques for designing ‘‘good procedures’’ are now applied for each procedure in a module. The most common example is a definition of a stack module. The main problems that have to be solved are:

[1] Provide a user interface for the stack (for example, functions `push()` and `pop()`).
[2] Ensure that the representation of the stack (for example, a vector of elements) can only be accessed through this user interface.
[3] Ensure that the stack is initialized before its first use.

Here is a plausible external interface for a stack module:

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

Assuming that 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 would be quite feasible to change the representation of this stack to a linked list. A user does not have access to the representation anyway (since `v` and `p` were declared `static`, that is, local to the file/module in which they were declared). Such a stack can be used like this:

```
#include "stack.h"

void some_function()
{
    push('c');
    char c = pop();
    if (c != 'c') error("impossible");
}
```

Pascal (as originally defined) doesn't provide any satisfactory facilities for such grouping: the only mechanism for hiding a name from ‘‘the rest of the program’’ is to make it local to a procedure. This leads to strange procedure nestings and over-reliance on global data.

C fares somewhat better. As shown in the example above, you can define a ‘‘module’’ by grouping related function and data definitions together 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`). Consequently, in C you can achieve a degree of modularity. However, there is no generally accepted paradigm for using this facility and the technique of relying on `static` declarations is rather low level.

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

The differences between C and Modula-2 in this area can be summarized by saying that C only *enables* the decomposition of a program into modules, while Modula-2 *supports* that technique.

Data Abstraction

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

```
class stack_id;  // stack_id is a type
                 // no details about stacks or stack_ids are known here

stack_id create_stack(int size); // make a stack and return its identifier
destroy_stack(stack_id);         // call when stack is no longer needed

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 built-in types in a language. 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, nor do such ‘‘variables’’ obey the usual scope rules or argument passing rules.

A type created through a module mechanism is in most important aspects different from a built-in type and enjoys support inferior to the support provided for built-in types. For example:

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

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

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

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

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

In other words, the module concept that supports the data hiding paradigm enables this style of programming, but does not support it.

Languages such as Ada, Clu, and C++ attack this problem by allowing a user to define types that behave in (nearly) the same way as built-in types. Such a type is often called an *abstract data type*†. I prefer the term ‘‘user-defined type.’’ A way of defining types that are somewhat more abstract is demonstrated in the ‘‘Multiple Implementations’’ subsection of §3. The programming paradigm becomes:

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

Where 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; }
    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 (that is, user-defined type) `complex` 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 expressed as user defined types. For concepts where the ‘‘module representation’’ is desirable even when a proper facility for defining types is available, the programmer can

---

† ‘‘*Those types are not "abstract"; they are as real as* `int` *and* `float`.’’ – Doug McIlroy. An alternative 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 paper would, given such a specification, be concrete specifications of such truly abstract entities.

declare a type and 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 with Data Abstraction

An abstract data type defines a sort of black box. Once it has been defined, it does not really interact with the rest of the program. There is no way of adapting it to new uses except by modifying its definition. This can lead to severe inflexibility. 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 necessary to allow operations such as `draw()` and `rotate()` to determine what kind of shape they are dealing with (in a Pascal-like language, one 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. Functions such as `draw()` must ‘‘know about’’ all the kinds of shapes there are. Therefore the code for any such function grows 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 are not able to add a new shape to a system unless you have access to the source code for every operation. Since adding a new shape involves ‘‘touching’’ the code of every important operation on shapes, it requires great skill and potentially introduces bugs into the code handling other (older) shapes. The choice of representation of particular shapes can get severely cramped by the requirement that (at least some of) their representation must fit into the typically fixed sized framework presented by the definition of the general type `shape`.

Object-Oriented Programming

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.). Expressing this distinction and taking advantage of it defines object-oriented programming. A language with constructs that allows this distinction to be expressed and used supports object-oriented programming. Other languages don’t.

The Simula inheritance mechanism provides a solution. First, 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 for which the calling interface can be defined, but where the implementation cannot be defined except for a specific shape, have been marked ‘‘virtual’’ (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 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);
}
```

To define a particular shape, we must say that it is a shape and specify its particular properties (including the virtual functions).

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

In C++, class `circle` is said to be *derived* from class `shape`, and class `shape` is said to be a *base* of class `circle`. An alternative terminology calls `circle` and `shape` subclass and superclass, respectively.

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. The amount of commonality between types that can be exploited by using inheritance and virtual functions is the litmus test of the applicability of object-oriented programming to an application area. In some areas, such as interactive graphics, there is clearly enormous scope for object-oriented programming. For other areas, such as classical arithmetic types and computations based on them, there appears to be hardly any scope for more than data abstraction and the facilities needed for the support of object-oriented programming seem unnecessary†.

---

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

Finding commonality among types in a system is not a trivial process. The amount of commonality to be exploited is affected by the way the system is designed. When designing a system, commonality must be actively sought, both by designing classes specifically as building blocks for other types, and by examining classes to see if they exhibit similarities that can be exploited in a common base class.

For attempts to explain what object-oriented programming is without recourse to specific programming language constructs see Nygaard[14] and Kerr[10]. For a case study in object-oriented programming see Cargill[3].

Having examined the minimum support needed for procedural programming, data hiding, data abstraction, and object-oriented programming we will go into some detail describing features that – while not essential – can make data abstraction and object-oriented more effective.

3 Support for Data Abstraction

The basic support for programming with data abstraction consists of facilities for defining a set of operations (functions and operators) for a type and for restricting the access to objects of the type to that set of operations. Once that is done, however, the programmer soon finds that language refinements are needed for convenient definition and use of the new types. Operator overloading is a good example of this.

Initialization and Cleanup

When the representation of a type 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. Given such a function, allocation and initialization of a variable becomes a single operation (often called instantiation or construction) instead of two separate operations. Such an initialization function is often called a constructor. In cases where construction of objects of a type is non-trivial, one often needs a complementary operation to clean up objects after their last use. In C++, such 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 the storage used:

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

C++ does not support garbage collection. This is compensated for, however, by enabling a type to maintain its own storage management without requiring intervention by a user. This is a common use for the constructor/destructor mechanism, but many uses of this mechanism are unrelated to storage management.

Assignment and Initialization

Controlling construction and destruction of objects is sufficient for many types, but not for all. It can also be necessary to control all copy operations. Consider class `vector`:

```
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 the assignment to `v1`. Alternatively it should be possible to prohibit such copy operations; preferably both alternatives should be available. For example:

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

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

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

In C++, a copy constructor, for example `X(const X&)` defines all initialization of objects of type `X` with another object of type `X`. In addition to explicit initialization copy constructors 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 assignment private:

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

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

Parameterized Types

Why would you want to define a vector of integers anyway? A user typically needs a vector of elements of some type unknown to the writer of the `vector` type. Consequently the vector type ought to be expressed in such a way that it takes the element type as an argument:

```
template<class T> class vector {     // 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; }
    // ...
};
```

A `template` specifies a family of types generated by specifying the the templats argument(s).

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, Clu, ML, and C++ support parameterized types†. There need not be any run-time overheads compared with a class where all types involved are specified directly.

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 one would like to be able to express and utilize 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 trivial, to deduce this from the definition of class `vector` and then allow `size()` to be applied to any `vector`. An interpreted or dynamically compiled language (such as Smalltalk) or a language supporting both parameterized types and inheritance (such as C++) has an advantage here.

Exception Handling

As programs grow, and especially when libraries are used extensively, standards for handling errors (or more generally: ‘‘exceptional circumstances’’) become important. Ada, Algol68, Clu, and C++ each support a standard way of handling exceptions††.

---

† The ANSI C++ committee, X3J16, only accepted templates into C++ in July 1990 so only a few C++ implementations support templates at the time of writing.

†† The ANSI C++ committee, X3J16, only accepted exception handling into C++ in November 1990 so C++ implementations that sup-

Consider again the `vector` example:

```
class vector {
        // ...
        class range { }; // type to be used for exceptions
};

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

Instead of calling an error function, `vector::operator[]()` can invoke the exception handling code, ‘‘throw the range exception.’’ This will cause the call stack to be unraveled until an exception handler for `vector::range` is found; this handler will than be executed.

An exception handler may be defined for a specific block:

```
void f(int i) {
    try {               // exceptions in this try block are handled by the
                        // exception handler defined below
        vector v(i);
        // ...
        v[i] = 7;       //  causes vector::range exception
        // ...
        int i = g();    // might cause a vector::range exception
    }
    catch (vector::range) {
            error("f(): vector range error");
            return;
    }
}
```

There are many ways of defining exceptions and the behavior of exception handlers. The facility sketched here resembles the ones found in Clu and ML.

A poor implementation of exception handling can be a serious drain on run-time efficiency and the portability of language implementations. The C++ exception handling can be implemented so that code is not executed unless an exception is thrown or portably across C implementations by (implicitly) using the C standard library functions `setjmp()` and `longjmp()`.

Type conversions

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

User-defined type conversions 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’’ (for example, matrices, character strings, and machine addresses) have natural mappings to and/or from other types.

One use of coercions has proven especially useful from a program organization point of view:

---

port exception handling are rare at the time of writing.

```
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 class `integer` to make `2+a` legal. Modifying existing code should be avoided as far 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

It has been claimed that a language supporting data abstraction must provide a way of defining control structures[12]. In particular, a mechanism that allows a user to define a loop over the elements of some type containing elements is often needed. This must be achieved without forcing a user to depend on details of the implementation 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 since an ordering is available to a user through the indices. I’ll define one anyway to demonstrate the technique. There are several possible styles of iterators. 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` can now be declared and used for a `vector` like this:

```
void f(vector& v)
{
        vector_iterator next(v);
        int i;
        while (i=next()) print(i);  // maybe too ‘cute’
}
```

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 that different kinds of iteration may 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 co-routine class[16].

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

---

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

```
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; }
};
```

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 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, a more general solution can be applied in addition to this simple one. Note that the ‘‘simple’’ solution requires more foresight from the designer of the container class than the iterator solution does. 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.

Multiple Implementations

The basic mechanism for supporting object-oriented programming, derived classes, and virtual functions can be used to support data abstraction by allowing several different implementations for a given type. Consider again the stack example:

```
template<class T>
        class stack {
        public:
                virtual void push(T) = 0; // pure virtual function
                virtual T pop() = 0;      // pure virtual function
        };
```

The `=0` notation specifies that no definition is required for the virtual function and that the class is abstract, that is, the class can only be used as a base class. This allows stacks to be used, but not created:

```
stack<cat> s; // error: stack is abstract

void some_function(stack<cat>& s, cat kitty) // ok
{
        s.push(kitty);
        cat c2 = s.pop();
        // ...
}
```

Since no representation is specified in the stack interface, its users are totally insulated from implementation details.

We can now provide several distinct implementations of stacks. For example, we can provide a stack implemented with an array

```
template<class T>
        class astack : public stack<T> {
                // actual representation of a stack object
                // in this case an array
                // ...
        public:
                astack(int size);
                ˜astack();

                void push(T);
                T pop();
        };
```

and elsewhere a stack implemented using a linked list:

```
template<class T>
        class lstack : public stack<T> {
                // ...
        };
```

We can now create and use stacks:

```
void g()
{
        lstack<cat> s1(100);
        astack<cat> s2(100);

        cat ginger;
        cat snowball;

        some_function(s1,ginger);
        some_function(s2,snowball);
}
```

Only the creator of `stack`s, `g()`, needs to worry about different kinds of `stack`s, the user `some_function()` is totally insulated from such details. The price of this flexibility is that each operation on such a type must be a virtual function.

Implementation Issues

The support needed 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 run-time 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 to a larger degree depend on types from libraries (and not just those described in the language manual). This naturally puts greater demands on facilities to express what is inserted into or retrieved from a library, facilities for finding out what a library contains, facilities for determining what parts of a library are actually used by a program, etc.

For a compiled language facilities for calculating the minimal compilation necessary after a change become important. It is essential that the linker/loader – with suitable help from the compiler – is capable of bringing a program into memory for execution without also bringing in large amounts of related, but unused, code. In particular, a library/linker/loader system that brings the code for every operation on a type into core just because the programmer used one or two operations on the type is worse than useless.

4 Support for Object-Oriented programming

The basic support a programmer needs to write object-oriented programs consists of a class mechanism with inheritance and a mechanism that allows calls of member functions to depend on the actual type of an object (in cases where the actual type is unknown at compile time). The design of the member function calling mechanism is critical. In addition, facilities supporting data abstraction techniques (as described above) are important because the arguments for data abstraction and for its refinements to support elegant use of types 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 allows user-defined types to be far more flexible and general than the ones designed using only data abstraction techniques.

Calling Mechanisms

The key language facility supporting object-oriented programming is the mechanism by which a member function is invoked for a given object. For example, given a 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 extensively used, the type system can be employed to select between different calling mechanisms. In C++, two alternatives are available:

[1] A normal function call: the member function to be called is determined at compile time (through a lookup in the compiler's symbol tables) and called using the standard function call mechanism with an argument added to identify the object for which the function is called. Where the ‘‘standard function call’’ is not considered efficient enough, the programmer can declare a function `inline` and the compiler will attempt to inline expand its body. In this way, one can 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] A virtual function call: The function to be called depends on the type of the object for which it is called. This type cannot in general be determined until run time. Typically, the pointer `p` will be of some base class `B` and the object will be an object of some derived class `D` (as was the case with the base class `shape` and the derived class `circle` above). 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 is found, say `D::f`, it can be called using the mechanism described above. The name `f` is at compile time converted into an index into a table of pointers to functions. This virtual call mechanism can be made essentially as efficient as the ‘‘normal function call’’ mechanism. In the standard C++ implementation, only five additional memory references are used.

In languages with weak static type checking a more elaborate mechanism must be employed. What is done in a language like Smalltalk is to store a list of the names of all member functions (methods) of a class so that they can be found at run time:

[3] A method invocation: First the appropriate table of method names is found by examining the object pointed to by `p`. 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 in that the method invocation uses a method table for the actual object.

A method invocation is inefficient compared with a virtual function call, but 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 showed the power of virtual functions. What, in addition to this, does a method invocation mechanism do for you? You can attempt to invoke *any* method for *any* object.

The ability to invoke any method for any object enables the designer of general purpose libraries to push the responsibility for handling types onto the user. Naturally this simplifies the design of libraries. However, it then becomes the responsibility of the user to avoid type mismatches like this:

```
// assume dynamic type checking.
// *** NOT C++ ***

Stack s;  // Stack can hold pointers to objects of any type

cs.push(new Saab900);
cs.push(new Saab37B);

cs.pop()->takeoff();   // fine: a Saab37B is a plane

cs.pop()->takeoff();   // Oops! Run time 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.

Combinations of parameterized classes and the use of virtual functions can approach the flexibility, ease of design, and ease of use of libraries designed with method lookup without relaxing the static type checking or incurring significant run time overheads (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);

cs.pop()->takeoff();            // fine: a Saab 37B is a plane
cs.pop()->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) whereas 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 whereas a C++ class is an exact specification and the user is guaranteed that only operations specified in the class declaration will be accepted by the compiler.

Inheritance

Consider a language having some form of method lookup without having an inheritance mechanism. Could that language be said to 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 of associating methods and the data structures they assume for their object representation. To enable a user of an object to know what kind of behavior to expect, there would also have to be some standard way of expressing what is common to the different behaviors the object might adopt. This ‘‘systematic and standard way’’ would be an inheritance mechanism.

Consider a language having an inheritance mechanism without virtual functions or methods. Could that language be said to 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 noticeably 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 the use of ‘‘type fields’’ to determine actual types of objects, so the problems with the lack of modularity of the code would remain†.

This implies that class derivation (subclassing) is an important programming tool in its own right. It

---

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

can be used to support object-oriented programming, but it has wider uses. This is particularly true if one identifies the use of 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 of what is inherited (see Snyder[18] and Stroustrup[19]), class derivation can be a powerful tool for creating new types. Given a class, derivation can be used to add and/or subtract features. The relation of the resulting class to its base cannot always be completely described in terms of specialization; factoring may be a better term.

Derivation is another tool in the hands of a programmer and there is no foolproof way of predicting how it is going to be used – and it is too early (even after almost 25 years of Simula) to tell which uses are simply mis-uses.

Multiple Inheritance

When a class `A` is a base of class `B`, a `B` inherits the attributes of an `A`; that is, a `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 a class `B` inherit from two base classes `A1` and `A2`. This is called multiple inheritance[23].

A fairly standard example of the use of multiple inheritance would be to provide two library classes `displayed` and `task` for representing objects under the control of a display manager and co-routines under the control of a scheduler, respectively. 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
};
```

Using (only) single inheritance only two of these three choices would be open to the programmer. This leads to either code replication or loss of flexibility – and typically both. In C++ this example can be handled as shown above with to no significant overheads (in time or space) compared to single inheritance and without sacrificing static type checking[20].

Ambiguities are handled at compile time:

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

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

In this, 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++, one would typically resolve the ambiguity by adding a function:

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

void f()
{
        // C’s own stuff
        A::f();
        B::f();
}
```

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

```
class W { /* ... */ }; // window

class Bwindow: public virtual W { // window with border
        // ...
};

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

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

Here the (single) `window` sub-object is shared by the `Bwindow` and `Bwindow` sub-objects of a `BMW`. The Lisp dialects provide concepts of method combination to ease programming using such complicated class hierarchies. C++ does not.

Encapsulation

Consider a class member (either a data member or a function member) that needs to be protected from ‘‘unauthorized access.’’ What choices can be reasonable for delimiting 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,’’ that is, all member functions. A non-obvious implication of this answer is that there cannot be a complete and final list of all functions that may access the protected member since one can always add another by deriving a new class from the protected member’s class and define a member function of that derived class. This approach combines a large degree of protection from accident (since you do not easily define a new derived class ‘‘by accident’’) with the flexibility needed for ‘‘tool building’’ using class hierarchies (since you can ‘‘grant yourself access’’ to protected members by deriving a class). For example:

```
class Window {
        // ...
protected:
        Rectangle inside;
        // ...
public:
        // ...
};
```

```
class Dumb_terminal : Window {
        // ...
public:
        void prompt();
        // ...
};
```

Here `Window` specifies `inside` as protected so that derived classes such as `Dumb_terminal` can read it and figure out what part of the `Window`'s area it may manipulate.

Unfortunately, the ‘‘obvious’’ answer for a language oriented towards data abstraction is different: ‘‘list the functions that need access in the class declaration.’’ There is nothing special about these functions. In particular, they need not be member functions. A non-member 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.

Encapsulation issues increase dramatically in importance with the size of the program and with the number and geographical dispersion of its users. See Snyder[18] and Stroustrup[19] for more detailed discussions of language support for encapsulation.

Implementation Issues

The support needed for object-oriented programming is primarily provided by the run-time system and by the programming environment. Part of the reason is that object-oriented programming builds on the language improvements already pushed to their limit to support for data abstraction so that relatively few additions are needed†.

The use of object-oriented programming blurs the distinction between a programming language and its environment further. Since more powerful special- and general-purpose user-defined types can be defined their use pervades user programs. This requires further development of both the run-time system, library facilities, debuggers, performance measuring, monitoring tools, etc. Ideally these are integrated into a unified programming environment. Smalltalk is the best example of this.

5 Limits to Perfection

A major problem with a language defined to exploit the techniques of data hiding, data abstraction, and object-oriented programming is that to claim to be a general purpose programming language it must

[1] Run on traditional machines.
[2] Coexist with traditional operating systems.
[3] Compete with traditional programming languages in terms of run time efficiency.
[4] Cope with every major application area.

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

Another implication is that an object-oriented programming language cannot completely rely 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-

---

† 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.

oriented programming languages employ garbage collection to simplify the task of the programmer 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 features are other potential problem areas. Any feature that is best implemented with help from a linker can 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.

6 Conclusions

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 support to be effective. Data abstraction primarily needs support in the form of language features and object-oriented programming needs further support from a programming environment. To be general purpose, a language supporting data abstraction or object-oriented programming must enable effective use of traditional hardware.

7 Acknowledgements

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

8 References

[1] Birtwistle, Graham et.al.: *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden. 1971. Chartwell-Bratt Ltd, UK. 1980.

[2] Dahl, O-J. and Hoare, C.A.R.: *Hierarchical Program Structures*. In *Structured Programming*. Academic Press 1972.

[3] Cargill, Tom A.: *PI: A Case Study in Object-Oriented Programming*. SIGPLAN Notices, November 1986, pp 350-360.

[4] C.C.I.T.T Study Group XI: *CHILL User’s Manual*. CHILL Bulletin no 1. vol 4. March 1984.

[5] Ellis, M.A and Stroustrup, B. *The Annotated C++ Reference Manual*. Addison-Wesley 1990.

[6] Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and its Implementation*. Addison-Wesley 1983.

[7] Ichbiah, J.D. et.al.: *Rationale for the Design of the Ada Programming Language*. SIGPLAN Notices, June 1979.

[8] Kernighan, B.W. and Ritchie, D.M.: *The C Programming Language*. Prentice-Hall 1978. 2nd Edition 1988.

[9] Keene, Sonya E.: *Object-Oriented Programming in COMMON LISP* Addison-Wesley 1988.

[10] Kerr, Ron: *Object-Based Programming: A Foundation for Reliable Software*. Proceedings of the 14th SIMULA Users’ Conference. August 1986, pp 159-165. An abbreviated version of this paper can be found under the title *A Materialistic View of the Software ‘‘Engineering’’ Analogy* in SIGPLAN Notices, March 1987, pp 123-125.

[11] Liskov, Barbara et. al.: *Clu Reference Manual*. MIT/LCS/TR-225, October 1979.

[12] Liskov, Barbara et. al.: *Abstraction Mechanisms in Clu*. CACM vol 20, no 8, August 1977, pp 564-576.

[13] Milner, Robert: *A Proposal for Standard ML*. ACM Symposium on Lisp and Functional Programming. 1984, pp 184-197.

[14] Nygaard, Kristen: *Basic Concepts in Object Oriented Programming*. SIGPLAN Notices, October 1986, pp 128-132.

[15] Rovner, Paul: *Extending Modula-2 to Build Large, Integrated Systems*. IEEE Software, Vol. 3. No. 6. November 1986, pp 46-57.

[16] Shopiro, Jonathan: *Extending the C++ Task System for Real-Time Applications*. Proc. USENIX C++ Workshop, Santa Fe, November 1987.

[17] SIMULA Standards Group, 1984: *SIMULA Standard*. ASU Secretariat, Simula a.s. Post Box 150 Refstad, 0513 Oslo 5, Norway.

[18] Snyder, Alan: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. SIGPLAN Notices, November 1986, pp 38-45.

[19] Stroustrup, Bjarne: *The C++ Programming Language*. Addison-Wesley, 1986. 2nd Edition 1991.

[20] Stroustrup, Bjarne: *Multiple Inheritance for C++*. Proceedings of the Spring'87 EUUG Conference. Helsinki, May 1987.

[21] Stroustrup, Bjarne: *The Evolution of C++: 1985-1989*. USENIX Computer Systems, Vol 2 No 3, Summer 1989.

[22] Stroustrup, Bjarne: *Possible Directions for C++: 1985-1987*. Proc. USENIX C++ Workshop, Santa Fe, November 1987.

[23] Weinreb, D. and Moon, D.: *Lisp Machine Manual*. Symbolics, Inc. 1981.

[24] Wirth, Niklaus: *Programming in Modula-2*. Springer-Verlag, 1982.

[25] Woodward, P.M. and Bond, S.G.: *Algol 68-R Users Guide*. Her Majesty's Stationery Office, London. 1974.