WG14/N298 X3J11/93-044
November 16, 1993

Classes in C

Robert Jervis
Sun Microsystems
robert.jervis@sun.com

This paper outlines a set of features that allow for useful object-oriented programming. I have made every attempt to restrict the features to those that are simple to use and allow for efficient implementation without excessive hidden machinery. Compatibility with existing C syntax and semantics is extremely important, as well as compatibility with C++ where possible. I have deliberately limited this proposal to a *necessary* set of features, without which C does not have meaningful object-oriented capability.

C's type system has evolved as our understanding of type has evolved. C itself arose by adding a type system to a typeless language (B). In recent years, the introduction of object-oriented technology in other languages has proven the value of object-oriented programming. It is now possible to define a minimal set of capabilities that are in keeping with the spirit of C: encapsulation, inheritance and polymorphism. If C is going to remain a language in day-to-day use, it must incorporate these features.

The purpose of this paper is to stimulate discussion of C and object-oriented programming. If we believe that C does not need object-oriented programming, or that we would need the full weight of C++ object-oriented features, then we can safely ignore this paper and move on. If on the other hand, we do believe the there is a place for OOP extensions in C, this paper's proposals or something like it will be part of the solution.

1. Classes

A class is a form of structure that contains member functions, called *methods*. In addition, members may be hidden from access outside the class. Class definitions can only appear at file scope. They cannot be nested, nor can they be defined inside a function. These constraints are intended to limit the complexity of classes. Since, in this proposal, methods are only prototyped within the body of a class, the problem of defining a method of a nested class is avoided as is the problem of nested functions.

A class is declared using the following type specifier:

```
class class-name { member-declarations }
```

The *class-name* is a tag name like any other. You cannot use class names as typedef names, which is a difference from C++. Since this is C, I believe that we should not confuse the tags name space with the normal identifiers name space. The result is that in C the `class` keyword is mandatory, while it is optional in C++. If you wish to use a class name like a typedef, you must explicitly declare a typedef as in:

```
typedef class foo { ... } foo;
```

The *member-declarations* of a class can contain function prototypes as well as data member declarations. The function prototypes within a class declaration declare the set of methods for that class.

5

A class provides for encapsulation by hiding private data from outside access and by supplying a coherent set of interface functions that manipulate the encapsulated object. Methods also help the namespace pollution problem by allowing each class to have its own set of method names.

1.1 Exposure

Member-declarations can also include *exposure markers* as in:

```
class foo {
        public:

        int     x, y;

        int     bar(int);

        private:

        double  hidden;
        };
```

Exposure markers are simply the keywords `public` or `private` followed by a colon preceding a group of declarations. By default, member declarations not preceded by any exposure marker are private.

1.2 Defining Methods

Methods are defined outside the class definition like other functions using syntax borrowed from C++.

```
class-name :: function-name ( arguments )
                        { function-body }
```

The class definition must be in scope at the point of definition of the method.

This mechanism introduces a problem: how do you spell the method name to the linker? As a standard we do not have to prescribe a particular implementation but it is enough to have something along the lines of:

```
classname@methodname
```

I do not propose full type-safe linkage and name mangling.

The address-of operator (`&`) cannot be applied to a method. The reason is that there is no type that describes such an object. This limitation is less crucial than one might at first assume. One solution does not introduce any new features, but constrains implementations. This would say that the address of a method has a function type that has the same arguments as the method, but one additional argument (inserted before the first one in the method) that is a pointer to the enclosing class. I can live with this contraint and just as prototype and non-prototype

function declaration styles are required to be nearly equivalent, this imposition would require that method calling conventions are essentially the same as the conventions for normal functions.

1.3 Calling Methods

You call a method using the C++ syntax of:

```
expr1.method(arguments)
```

or

```
expr2->method(arguments)
```

In the first form, *expr1* has class type. In the second form, *expr2* has pointer to class type.

1.4 This

In a call to a method, the address of the object mentioned in the method call is passed as a hidden argument to the method. The method can refer to the hidden parameter using the keyword `this`. For class T, the implicit declaration of this is:

```
class T * const this;
```

In all respects, `this` is treated as a function formal parameter as far as storage duration and scope are concerned.

1.5 Class Scope

You can refer to data members of the class using simply the member name. Such a reference is implicitly relative to the `this` pointer. For example:

```
class foo {
        int     i;
int     hidden(int);
        public:
int     bar();
        };

foo::bar()
        {
        i = 3;          /* same as this->i = 3; */
```

```
        hidden(4);      /* same as this->hidden(4); */
}
```

As the example illustrates, calls to other methods are allowed as well.

In the current definition of C, structure members are identified as a distinct name space, apart from normal identifiers. This distinction isn't strictly speaking necessary, since a member name can only appear in expressions in Standard C as the right operand of a dot or arrow operator. Also, since each structure can have its own set of members which do not have to be unique across all structures, Standard C makes member names behave almost as if they were scoped already.

For method functions, the normal C scoping rules are modified. An additional scope is presumed to exist, interposed between the outermost function block scope and file scope. This new scope is called *class scope* and contains all of the members defined for the class to which the method belongs. As we shall see below under inheritance, inherited members may also be available in that scope.

The implication of these scoping rules are that file scope identifiers are masked within methods by identically spelled class member names. For example:

```
class   foo     {
        int     i, j;

        int     bar();
        };

int     i;

foo::bar()
        {
        i = 3;          // refers to this->i, not global i
}
```

Note that automatic variables can mask class members. Extern declarations can also mask member names:

```
class   foo     {
        int     i, j;

        int     bar();
        };

int     i;

foo::bar()
        {
        int     j;
        extern int i;

        i = 3;          // refers to global i, not this->i
        j = 7;          // refers to local j, not this->j
        i = this->i + this->j;
                        // you can still access the members
}
```

As in this previous example, you can use the `this` pointer quite naturally to circumvent some of the constraints

of class scope.

2. Virtual Functions and Inheritance

Inheritance and virtual functions together give you the ability to create polymorphic objects. If you use base classes to define abstract types, then derived classes can encapsulate their individual implementations while supporting a common public interface. Becasue of this I believe that polymorphism is the single most important contribution of object-oriented programming. To exclude this capability from C would be to deny it something vital.

I propose that C allow a class to inherit from a single base class. To declare such a class use the following syntax:

```
class foo inherit bar { ... }
```

An instance of the derived class `foo` contains an instance of the base class `bar`. That instance has the same address as the object as a whole. In effect, the derived class forms a variant, with the base class as the common prefix. All of the public members of the base class are available as members or methods of the derived class.

I have proposed the use of a new keyword `inherit` rather than the C++ syntax. I have done this because I believe the C++ syntax of a colon leaves the relationship between base and derived class less clear. The proposed syntax is clear about which is the derived and which is the base class.

2.1 Redefinition of members

Any member can be redefined in a derived class. Except for virtual functions (discussed below), the redefined member can have any type whatsoever. If a data member is redefined, it masks the definition in the base class, but space is still reserved for the base class member. Methods can be redefined as well. In calling a method for an object in a derived class, the call names the method in the derived class and uses its argument and return types.

2.2 Pointers to class objects

A pointer to a class object of a derived type is also a valid pointer to any of the base class objects. Pointers should compare equal (when appropriately cast) if one pointer points at a derived class object and the other points at any of its base class objects.

A base class may have several different classes derived from it, so a programmer converting from a base to a derived type must determine that the base being pointed at is in fact part of the correct derived object. But a derived class has only one nested set of bases and only one object of each of the base types to convert to. Therefore,

converting a pointer from a derived type to a base type should be allowed without an explicit cast. In C++, such a conversion is disallowed without an explicit cast.

Converting from a base to a derived class is more dangerous since it is valid only if the referenced object is also an instance the derived class. Therefore a cast should be required. I do not recommend dynamic casts as in C++. Such casts are a new feature of C++ and I do not feel prepared to accept them into C. I would prefer to wait and see the reaction to C++ dynamic casts before we commit C to them.

2.3 Virtual functions

A virtual function is a method hat uses a special calling convention. The basic principle in C is that an object does not have a type. Any type is imposed by the lvalue used to refer to the object. Virtual functions, however, rely on an identity that persists independently of the lvalues used to refer to the object. In C++ implementations, this identity is a `vtbl` pointer. Such a mechanism would suffice for this proposal as well.

Method calls for plain mathods are resolved to direct calls to the named method at compile time. For pointers to objects, however, if the call is to a virtual function then the call is made not to the declared type of the pointer, but to the actual type of the identity of the referenced object. For example:

```
class base { public:
int     virtual foo();
        };

class derived inherit base { public:
int     virtual foo();
        };

main()
        {
        derived x;
        base    *p;

        p = &x;         /* or (base *)&x if implicit is not ok
*/

        p->foo();
}
base::foo()
        {
        printf("base::foo()\n");
}

derived::foo()
        {
        printf("derived::foo()\n");
}
```

This program prints:

```
derived::foo()
```

My experience from C++ and Parasol is that this mechanism is very efficient. The cost of a call to a virtual function is hardly more than an indirect call through a function pointer or even a direct call.

The conventional implementation for this mechanism involves storing in the object itself a pointer to a table of pointers (called the `vtbl`) to the virtual methods of that object. The `vtbl` pointer can be stored in a static object at compile time and in an automatic object at entry to the block. The problem is how to get this value stored in a malloc'ed object.

There are three possible approaches. The first option, which requires no special machinery in the language, is to copy a static object into the malloc'ed space, which would copy the `vtbl` pointer. This is wasteful, and highly inefficient if the object is large. We could exploit the compound literals extension, if that is adopted, but all that avoids is having to create a static name. The second option is to define some general purpose way to assign the `vtbl` pointer directly. The last option is to introduce an operator `new`.

2.4 Pointers to virtual functions

While pointers to methods simply store the address of one function or another, with virtual functions you may want to store the offset into the `vtbl` for a given method. That way, you can record the name of a method, then when you make a call it will call that method whose implementation is appropriate for the object named in the call.

This is not an overwhelmingly important feature. Quite a large amount of code can be written without this capability.

3. Operator new

This operator is intended as a bit of syntactic sugar on top of the `malloc` function. Except where a type has virtual functions, the following two expressions are equivalent (with one exception detailed below):

```
p = new T;

p = (T *)malloc(sizeof T);
```

The above equivalence holds except for types with virtual functions. For such objects, the new operator would also set the `vtbl` pointer of the allocated space. If the allocated type has embedded members which in turn have `vtbl` pointers, then they would also be initialized.

Note that this semantics for operator `new` here are different from C++. The proposal for C is a simple bit of syntactic sugar without altering the underlying memory allocation scheme of C.

3.1 Memory allocators

A memory allocator is a class containing a method called `malloc` with aparticular prototype signatiure. An allocator class can define this function to implement any sort of memory allocation the programmer wants. The `malloc` method must be declared as:

```
void *malloc(size_t);
```

or

```
virtual void *malloc(size_t);
```

If a member is declared named `malloc` but using some other type, the class is not a memory allocator class. The name `malloc` is in all ways a normal identifier and can be used in automatic variables and so on.

I would like to propose the following form of the new operator:

```
p = A new T;
```

In this form, the expression A names a memory allocator object (an object with a method named `malloc`). Instead of calling the global `malloc` this form calls the method as if:

```
p = (T *)A.malloc(sizeof T);
```

This of course would extend to include setting the `vtbl` pointer for objects that need them.

4. Virtual tables

Most implementations have solved the problems of how to manage virtual tables for C++. The above proposals do not impose any additional burdens on an implementation than already found in C++.

Still, virtual tables are a mechanism that introduces hidden machinery that the C programmer cannot describe or manipulate. You may also wish to store additional information in the virtual table yourself. Or you may wish to modify a virtual function pointer in the table during execution. If virtual tables could be described in C such things could be done in a portable manner.

In effect, for each class containing a virtual function, the compiler must create a second structure definition that describes the function pointers in the virtual table. A solution to this problem should allow the compiler to supply the virtual table definition as a default (making the simple cases easy). I could see a proposed extension that would allow a programmer to include in a class definition a reference to a structure definition that is the virtual table for

that class. One could then gain access to the virtual table pointer in an object with some construct like '`this->virtual`'.

C++ has not provided general purpose access to the virtual tables, so one could conclude that such capability is not absolutely necessary. On the other hand, if the philosophy of C providing direct control over as much of the environment as possible is followed, then we might want to provide such a capability.