Classes:
An Abstract Data Type Facility for the C Language

Bjarne Stroustrup

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Language constructs for definition and use of *abstract data types* ease the design and maintenance of large programs. This paper describes the C *class* concept, an extension to the C language providing such constructs. A class is defined using standard C data types and functions, and it can itself be used as a building block for new classes. A class provides a way of restricting access to a data structure to a specific set of functions associated with it, without incurring significant overheads at compile time or at run time.

The C class concept is introduced by small examples of its use, and familiarity with the C language [2] is assumed. Appendix A is a complete small C program using classes.

Classes have been in use for more than a year on a dozen PDP11 and VAX UNIX† systems [1], and they are currently used for a diverse set of projects on more than 30 systems. Classes are currently implemented by an intermediate pass of the *cc* compiler, called the class pre-processor, which is invoked when the directive *#class* is found in a C source file. The class pre-processor is easily ported to a system with a version of the portable C compiler. A Motorola68000 version is in use.

August 14, 1981

† UNIX is a Trademark of Bell Laboratories.

Classes:
An Abstract Data Type Facility for the C Language

Bjarne Stroustrup

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

It is common practice to provide non-trivial data structures with a set of access functions. This practice helps preserve the consistency of the data and also aids programmers in the task of writing and modifying programs. *Classes* have been added to the C language to allow functions to be explicitly associated with data, and to protect the data against "irregular" access from other functions.

A simple example is a stack where only the functions *push()* and *pop()* are presented to the user. The exact representation of the stack, that is the type of data structure used and its initialization, is hidden from the user. The declaration of a *stack* might look like this*:

```
class stack
{
        char        s[SIZE];
        char *      min;
        char *      top;
        char *      max;
        void        new(void);
public:
        void        push(char);
        char        pop(void);
};
```

A *class* declaration can be seen as a *struct* declaration to which functions have been added. Like a structure declaration, a class declaration allocates no storage; it merely describes a template for objects of that class. Names in the public part, that is appearing after the keyword *public*, provide the interface to users. The other class member names are private, that is, they can only be used by the implementor of the class in the functions declared in the class declaration. The functions named in the class declaration must themselves be declared for the declaration to become meaningful, for example the two public functions:

* A function of type *void* does not return a value; a function declared with the argument list *(void)* cannot accept an argument. The *void* keyword, the ability to define the types of argument expected by a function in *extern* declarations, and the function declaration syntax used in this paper are recent extensions of the C language. The *new* and *delete* operators which are used to manage the free (dynamic) store are currently implemented only by the class pre-processor.

```
void stack.push(char c)
{
   if (max <= top) error("stack overflow");
   *top++ = c;
}

char stack.pop()
{
    if (top <= min) error("stack underflow");
    return *(--top);
}
```

The class name *stack* is used as a prefix to the function names in these declarations to indicate that the function is the one named in the class declaration and that it should be compiled in the context of that class. For example, *min* in *stack.pop()* is the name of the member of class *stack*, and not an undefined *extern*. This makes it possible to use the same name to denote different functions in different classes and to have class functions with the same name as a *extern* function.

The function name *new* has a special meaning in a class declaration. If present, the *new()* function is guaranteed to be executed immediately after an object of the class has been created. This can be used to provide initialization:

```
void stack.new()
{
   top = min = &s[0];
   max = &s[SIZE-1];
}
```

Given the declaration of *class stack*, objects of that class can be declared. For example,

```
class stack s1, s2;
```

creates two objects of *class stack* named *s1* and *s2*, respectively. The *public* functions of *class stack* can now be used on these objects of *class stack*

```
s1.push('h');
s1.push('o');
c = s1.pop();
```

The notation is the usual *object.member* notation from C structures. Pointers can be defined, initialized, and used in the usual way:

```
class stack * p = &s2;

p->push('s');
c = s2.pop();
```

would be a way of assigning the character *'s'* to *c*. Because the member *top* is private the statement

```
c = *(p->top);    /* attempt to read the top */
```

would be illegal and cause a compile time error.

Instantiation of Class Objects

The example above presents the idea of a facility providing an abstract data type based on C data types and functions. This idea must be developed into a proper language facility. In this task the designer was guided, not only by the obvious desire for generality, but also by a wish to provide a simple facility which causes only low overheads in compiler complexity and run time support.

Some of the facilities arising from this balancing effort can be presented through a further elaboration of the stack example. All stacks created using the definition above have the size SIZE. This is not ideal, so let us try again:

```
class stack
{
      void      new(short);
      void      delete(void);
      char *    min;
      char *    top;
      char *    max;
public:
      void      push(char);
      char      pop(void);
};
```

This *class stack* declaration does not specify the amount of store to be allocated for the stack itself. Instead it is specified that an argument of type *short* must be provided for *stack.new()*. Arguments to a *new()* function are provided as part of the declaration of a class object. For example:

```
class stack s1(SIZE), s2(200);
```

The *new()* function then provides the interpretation of them. In this case:

```
void stack.new(int size)
{
   top = min = new char[size];
   max = min+size-1;
}
```

A vector of *size* characters is allocated on the free store. This, however, creates a new problem. Because we cannot (in general) assume that a garbage collector is available, we must clean up after ourselves. That is, in this case we must deallocate the vector pointed to by *min* when an object of *class stack* is deleted. This is done by defining a parameterless function called *delete*.

```
void stack.delete()
{
   delete min;
}
```

If a function of this name is mentioned in the class declaration it is guaranteed to be the last function accessing an object of that class before it is deleted. A *delete()* function cannot be explicitly called, and neither can a *new()* function*.

A *delete()* function is typically declared *void*, and the deallocation of the class object after return from *delete()* is unconditional. If, however, *delete()* returns an *int* value then an object created using the *new* operator can avoid being deallocated by returning a non-zero value. Return *0* frees the store.

* The exception to this rule can be found in the section describing classes containing class objects.

Class objects can be declared in two ways. A normal C style declaration can be used to create class objects with their scope determined in the same way as other C variables*. For example:

```
class stack s1(SIZE), s2(SIZE+100);
```

will allocate space for s1 and s2 on the stack if placed in a function body. The vector denoted by *min* will however always be allocated on the free store by *stack.new()*.

Alternatively, class objects can be generated on the free store by the *new* operator. For example:

```
class stack * p = new class stack(SIZE);
```

Here the *new()* function's argument list is placed after the specification of the type of object to be created in the same way as it was placed after the object name in the standard C style declaration.

The *new* operator first allocates space from the free store and then calls the appropriate *new()* function, if any. Class objects generated using the *new* operator do not have a name, only an address.

The only way of destroying an object generated by the *new* operator is to apply its inverse operator *delete* to a pointer to that object. The *delete* operator first executes the appropriate *delete()* function, if any, and then returns the space occupied by that object to the free store. For example: *delete p* will free the store occupied by the stack generated above after having executed *stack.delete()* for it.

Applying the *delete* operator to an object which has not been allocated using the *new* operator is a null operation.

If a *new* function does not need arguments the argument list need not be present, or it can be empty. For example, the declarations: *class foo x;* and *class foo x();* are both valid, and their meanings identical.

When writing a class declaration it is possible to specify an argument list to be used by the *new()* function in case no argument list is provided in a class object declaration. An initializer is simply provided for each (formal) parameter in the specification of *new()*. For example:

```
class stack
{
        void new(int = SIZE);
        ...
}
```

specifies that a declaration

```
class stack x;
```

is equivallent to the declaration

```
class stack x(SIZE);
```

This provides a convenient shorthand for use in classes where objects are typically given a "standard" value at creation, and only "more sophisticated" use of the class demands the use of arguments.

* However, for implementation reasons it will on many systems not be possible to provide *extern* or *static* objects of classes with *new()* or *delete()* functions.

Organization of Multi-source-file Programs

Typically a C program consists of many source files. Information needed in more than one source file is placed in a "header file" which is then "included" in all source files needing the information.

The class concept is designed so that any class declaration can be placed in a header file and included in all source files using that class. In particular, where many source files are used for the program there is still one single copy of the class declaration included both in files containing class member function declarations and in files just using objects of the class. If all argument types for class member functions are specified in the class declaration then the type information used by the "users" of the abstraction represented by the class is identical to the information used by the "implementors" of that abstraction. It is also guaranteed that there will be at most one declaration of a class member function in a program so that all users get the same implementation of the abstraction. The program in Appendix A is organized this way.

The *make* program [1] can be used to ensure that a source file using a class is recompiled if and only if the declaration of the class is changed, but not just because a function of that class is changed.

Pointers to Functions

A class function is shared by all objects of its class and remains unchanged throughout the execution of the program. If greater flexibility is desired a pointer to a function can be used instead. For example, imagine a table implemented so that initially its members are stored in a fixed size table, but later, if that initial allocation is exceeded a linked list representation is needed. This could be achieved by providing two access functions, and accessing the table through a pointer denoting the appropriate one:

```
typedef int (*PF)(int);

class table
{
   ...
   int vector_put(int);
   int linked_put(int);
public:
   PF put;
};
```

The pointer to function *put* can be used exactly as a member function would have been, that is *p->put(10)*. This allows the programmer to keep the interface the same independently of whether a function or a pointer to a function is used for the implementation of a class member function.

Class Objects as Class Members

A class object can be declared as a member of a class like this:

```
class x {
   class y a;
};
```

and its *new()* function, if any, will be executed as the first part of the class's own *new()* function, if any. If a class containing class object members does not have a *new()* function then only the member object's *new()* functions, if any, will be executed. Class members' *delete()* functions, if any, are called as the last part of the class's own *delete()* function, if any.

If a class object member needs arguments for its *new()* function then these must be provided by the *new()* function of the class of which it is a member. This is done by a call to the member object's *new()* function. For example:

```
class x { void new(int); };

class y {
   void new(int);
   class x a;
};

void y.new(int arg) {
   a.new(arg+10);
}
```

Vectors of class objects

A vector of class objects can be declared and referred to in the same way as vectors of other C types. For example:

```
class x vec[LIMIT];

vec[i].data = 10;
vec[i+10].fct(2);
```

However, if class *x* here has a *new()* function then the initialization must be performed. This is done by calling *x.new()* for each element of the vector, as if the statement

```
for (i=0; i<LIMIT; i++) vec[i].new();
```

had been written using a name *i* which was otherwise undefined in the scope.

Where such a *new()* function needs an argument list it can be provided like this:

```
class y v2[10](10);
```

Every member of *v2* is initialized using the value zero. If different value are needed for the different vector elements side effects on the expressions in the argument list can be used:

```
int i, j;
i = 0;
class z v3[1000](i++);

i = j = 0;
struct { int a1, a2; } list[] = { {1,2}, {3,7}, {0,9}, {1,7} };
class z v3[10](list[i++].a1, list[j++].a2);
```

When using such side effects it should be remembered that the order of evaluation of C function arguments is undefined.

Class Object Assignment

The assignment *a=b* is legal if *a* and *b* are objects of the same class. The semantics of this assignment are that of a C *struct* assignment, that is, after the assignment the value of *a* is a copy of the value of *b*, and *b's* value is unchanged. If a *delete()* function exists for *a* then it will be executed before the information from *b* is copied.

A more interesting use of class object assignment would be:

```
extern class x f(class x);

class x a, b;
...
a = f(b);
```

Unfortunately, this standard struct-like assignment is not always ideal. Typically a class object is only the root of a tree of information and a simple copy of that root without any notice taken of the branches is undesirable. Similarly, simply overwriting a class object can create chaos.

Changing the meaning of assignment for objects of a class provides a way of handling these problems. This is done by declaring a class member function called *"operator ="*. For example:

```
class x {
   int   a;
   class y * p;
   void operator = (class x *);
};

void x.operator = (class x * from)
{
    a = from->a;
    delete p;
    p = from->p;
    from->p = 0;
}
```

This defines a destructive read for objects of class *x*, as opposed to the copy operation implied by the standard semantics. The function *"operator ="* which performs the overloading cannot be called directly, but to simplify the explanation below assume that its name is *x_assign*.

When an assignment *a=b* of objects of class *x* is seen the compiler generates a call *a.x_assign(&b)*. The value of the original expression *a=b* is the (new) value of *a* as expected, so that, for example *c=a=b* is meaningful.

Where assignment is overloaded in this way the interpretation of the example *a=f(b)* becomes more subtle. First *b* is passed by value, that is assigned to a local variable of *f()*. Assuming that the name of the formal parameter was *arg_b* then *arg_b.x_assign(&b)* is executed.

When a class member function is executed, in this case *x_assign()*, all class objects on which it operates must have been initialized by their *new()* functions, if any. For the *a=f(b)* example this implies that if class *x* had a *new()* function, then that function must be executed for the object called *arg_b* above. If *x.new()* does not take any arguments this is trivially achieved. If it does they can be provided by declaring a default argument list for *x.new()*.

The "this" Pointer

One pointer is always implicitly defined in a class function. It is the pointer *this*, that denotes the associated class object. That is, in a class function *this->name* is equivalent to *name* for all names of members of that class. The *this* pointer is particularly useful for linked list manipulation. For example:

```
class link
/*   link objects can form a doubly linked list.
     p->put(q) adds link q to the left of link p.
     q = p->get() removes the link to the right of p.
*/
{
      void           new(void);

      class link *   left;
      class link *   right;
public:
      void           put(class link *);
      class link *   get(void);
};

void link.new()
{
      left = right = this;
}
```

declares a type of doubly linked list where an element is always initialized in a way appropriate for circular lists.

The implied declaration for *this* in each function in a class *X* is:

```
class X * this = & this_object_of_class_X;
```

To ensure that *this* always has its defined meaning it is illegal to assign a value to it.

The function *link.put()* could be declared like this:

```
void link.put(class link * p)
{
    p->left = left;
    p->right = this;
    left->right = p;
    left = p;
}
```

This utilizes the feature that a class function can access the private members of every object of its class to which it has a pointer, not just the members of the object from which it was called.

Inline Substitution of Class Member Functions

It is not possible to use the standard C pre-processor *#define* to provide macro expansion of class functions. Instead, if it is decided that the usual overhead of C function call and return should be avoided for a class function, then it can be specified that it should be inline substituted. This is done by including the body of the function in the class declaration. For example:

```
class x {
   int a;
public:
   int reada() { return a; };
};
```

This allows the private variable *a* to be read (using *reada()*), but not written to, from outside class *x*. The use of *reada()* on the left hand side of an assignment will produce an error, but when used in a right hand context (as intended), for example, *i=p->reada()* it will generate the desired simple read of *p->a*.

Another use of inline substitution is for *new()* functions for simple classes. These commonly consist of just one or two assignments to private variables, are invoked from only a few places in a program which, however, are frequently executed. For example, class *link* could have been declared like this:

```
class link
{
public:
   class link * left;
   class link * right;
   void new() { left = right = this; };
   ...
};
```

Note that to conform with C's rule that the name of a variable must be declared before its use *new()* has to be placed textually after the declaration of *left* and *right* on which it operates.

An inline substituted function obeys the usual scope rules for functions, and its arguments are typed in the same way as arguments to other functions. In these respects an inline function differs from a C macro, which takes arbitrary strings as arguments and its body is interpreted in the (differing) contexts of the calls to it.

There will always be (implementation dependent) restrictions on what can be done with inline substituted class functions*.

* The current set of restrictions is:

[1] Side effects on the arguments to an inline function are detected and treated as an error.

[2] An inline function which returns a value must contain exactly one *return* statement, which must be the last statement in the function.

[3] An inline function which has been declared *void* cannot contain a *return* statement.

[4] Local variables cannot be declared in an inline function.

[5] Global variables cannot be accessed from an inline functions.

If these restrictions seem too draconian to you remember that inline substitution is only an optimization facility which you never need for expressing the logic of your program.

Derived Classes

An existing class, for example the *class link* defined above, often provides an abstraction that nearly, but not completely, fulfills the demands of a particular application. For example, one could wish to manipulate circular doubly linked lists where each element in a list held a word in the form of a string of characters. *Class link* provides a known and well tested representation of doubly linked circular lists, but provides no facility for associating information with a link. In other words, one would like a "tailor made" version of the "standard" class for a particular application without having to know the details of the implementation of that class, or have to design and test a completely new class.

This can be done in C by "deriving" a new class from an existing *base* class. For example:

```
class wordlink : link
{
public:
      char     word[SIZE];
      void     clear(void);
      link.put;
      link.get;
};

void wordlink.clear()
{
   short i;
   for (i=0; i<SIZE; i++) word[i] = 0;
};
```

declares a class named *wordlink* that is a *link*, and in addition to the data and functions of a link has a vector of characters called *word*, and a function called *clear*. The declarations:

```
class wordlink wl, * wp = new class wordlink;
```

will create two objects of *class wordlink* and initialize them correctly using *link.new()*, so that the following expressions are meaningful:

```
wl.put(pp);   /* chain the two wordlinks together */
wl.word[i] = 'c';
pp->clear();
c = ((class wordlink *) wl.get())->word[i];
```

The colon after the class name, *wordlink*, in the class declaration indicates that it is to be a derived class. The name of the base class, *link*, then follows.

Functions from a derived class have no special access to the private data of its base class. Like other functions they can use the public names only. Public names from the base class are treated as private in the derived class, unless they are explicitly declared to be public there also. That is, statements like

```
wl.put(pp);
```

are only legal because the qualified name *link.put* was mentioned in the public part of the declaration of *class wordlink*. To make a public name from a base class a legal public name for a derived class, leaving its meaning unchanged, its fully qualified name should be quoted, but no type information can be supplied.

It is often useful to have all public names from the base class be legal public names for a derived class. This is achieved by preceding the name of the base class with the keyword *public*. For example:

```
class wordlink : public link
{
public:
      char     word[SIZE];
      void     clear(void);
};
```

When a derived class is declared then two *new()* functions can exist, one for the base and one for the derived class, and each must be executed to ensure the proper initialization of an object of the derived class. First the *new()* for the base class is executed, and then the *new()* from the derived class. The *delete()* functions are executed in the reverse order; that is, derived class before base.

Arguments are supplied to the derived *new()* in the usual way, and *base.new()'s* argument list, if any, is specified in the declaration of the derived class's *new()* function. For example, using the declaration of *class stack* from the previous section, we can write:

```
class mystack : public stack
{
   void     new(short);
   ...
};

void mystack.new(size) short size;
: (size+10)
{
   ...
};
```

which specifies that objects of class *mystack* need a *short* as argument, that the argument will be called *size*, and that it will pass the value of the expression *size+10* to *stack.new()*.

The expressions in the argument list for a base class need not be expressed exclusively in terms of the derived class's formal parameters. For example:

```
class base
{
   void     new(int, int, int);
   ...
};

class derived : base
{
   void     new(int, int);
   ...
};

void derived.new(int a, int b)
: (a,f(b),2)
{
   ...
};
```

This will ensure that *base.new()* is executed as the first statement in *derived.new()*:

```
void derived.new(int a, int b)
{
   base.new(a,f(b),2);
   {
      the actual body of derived.new

   }
}
```

So, as a result of the declaration

```
class derived x(1,2+3);
```

derived.new is called with the argument list *(1,2+3)*, but before the first of its statements is executed *base.new* is called with the argument list *(1,f(5),2)*.

Any class can be used as a base class, in particular, a derived class can be used as a base.

The first obvious use of the derived class concept is to provide libraries of useful base classes for general use. An example of such a library is the classes for linked list manipulation and coroutine style programming that are described in Reference 3. For many programmers such libraries will be the only use of derived classes, they need never write a class intended to be used as a base class or worry about the finer details of the derived class concept. It is, however, only through the use of derived classes that the class concept can significantly affect the logic of programs written in C

In the class *wordlink* example above the use of *link.get()* on objects of class *wordlink* was made annoyingly clumsy by the need for a cast. Furthermore, because *link.put()* takes a "plain link" as argument it is not trivial to ensure that a chain of *wordlinks* does not, through a programming error, have a "plain link" on it.

The solution to this problem is to provide class *wordlink* with its own *get()* and *put()*, and by specifying these to be inline substituted ensure that the type checking thus achieved will cause no runtime overhead at all:

```
class wordlink : link
{
   char  word[SIZE];
public:
   void  clear(void);
   class wordlink * get(void) { return (class wordlink *) link.get(); };
   void  put(class wordlink * p) { link.put(p); };
};
```

Name Clashes in Derived Classes

Sometimes it is useful to provide a derived class with a member name identical to a name in the base class. One reason for this would be to provide a different service with an identical interface. For example:

```
class unsafe            /* assume responsible user */
{
      int      data;
public:
      int      pub;
      void     update();
};

class safe : unsafe     /* validate request before
                           using ``unsafe''
                        */
{
public:
     int       pub;
     void      update();
};
```

A program using *safe* can now be identical to one using *unsafe* once the declarations have been taken care of. To access the clashing names from *unsafe* functions from *safe* must use fully qualified member names. For example:

```
void safe.update()
{
   unsafe.pub = 1;
   unsafe.update();
}
```

Such complete qualification are in fact legal for every class member name, but does not affect the meaning of the program in the absence of name clashes.

Generic Functions

The use of derived classes introduces a form of generic functions that can be used without violating typing requirements. If *f()* is a function from class *x* then

```
p->f();
```

will be legal both if *p* is a pointer to class *x*, and if it is a pointer to a class derived from class *x* where *f()* has been made public.

In a similar way a function declared

```
f(class x * p)
{
   ...
}
```

will accept a pointer to a class derived from class *x* as argument, provided all public names from class *x* have been made public in that derived class.

Another form of generic function can be obtained by declaring functions with the same name in several classes derived from a common base class. For example:

```
class matrix
{
public:
      short           type;
};

class sparse : public matrix
{
public:
      class matrix *  multiply(class matrix *);
};

class dense : public matrix
{
public:
      class matrix *  multiply(class matrix *);
};
```

If now both *multiply()* functions are declared to accept a pointer to an object of class *matrix*, rather than to a specific type of matrix. Then the expressions

```
p->multiply(q);
q->multiply(p);
```

are legal if *p* and *q* are pointers to objects of the derived classes *sparse* and *dense*. The purpose of the member *type* of class *matrix* is to enable the two *multiply()* functions to determine the types of their arguments so that they can operate accordingly. Casting must be used to convert the "plain" class *matrix* pointer passed to a pointer of the appropriate derived class type. For example:

```
class matrix * sparse.multiply(class matrix * arg)
{
        ...

        switch (type) {
                case DENSE:
                        d = (class dense *) arg;
                        ...
                        break;
                case SPARSE:
                        s = (class sparse *) arg;
                        ...
                        break;
        }
        ...
}
```

Generic Classes

The class *stack* example in the introduction explicitly defined the stack to be a stack of characters. That is sometimes too specific. What if a stack of long integers was also needed? What if a class *stack* was needed for a library so that the actual stack element type could not be known in advance? In these cases the class *stack* declaration and its associated function declarations should be written so that the element type can be provided as an argument when a stack is created in the same way as the size was.

There is no direct language support for this, but the effect can be achieved through the facilities of the standard C pre-processor. For example:

```
class ELEM_stack {
   ELEM * min, * top, * max;
   void new(int), delete(void);
public:
   void push(ELEM);
   ELEM pop(void);
};
```

This declaration can then be placed in a header file and macro-expanded once for each type ELEM for which it is used:

```
#define ELEM long
#include "stack.h"
#undef ELEM

typedef class x X
#define ELEM X
#include "stack.h"
#undef ELEM

class long_stack ls(1024);
class long_stack ls2(512);
class X_stack xs(512);
```

This is certainly not perfect, but it is simple, and it is quite easy to ensure that the necessary copying of the class member function declarations take place exactly once for each type ELEM.

This style of writing lends itself well to the representation of simple aggregates of simple objects, for example stacks, sets, queues, or vectors of characters., integers, or pointers.

For abstractions where the objects are more substantial, like symbol tables or geometrical figures, or where the duplication of the class member functions becomes a significant overhead an alternative solution using base classes becomes attractive. For example:

```
typedef class elem * ELEM;

class stack
{
   ...
public:
   void push(ELEM);
   ELEM pop(void);
};
```

Thereafter pointers to objects of any class derived from class *elem* can be put onto the stack. This technique avoids the copying of code implicit in the first generic solution by using more data space. Both techniques lend themselves to the production of library software.

Friends

Some concepts are best represented by a set of mutually dependent classes, rather than by a single class or by a set of independent classes. For example, consider the problem involved in writing the function *multiply()* of class *sparse* as described in the matrix example above. To perform the desired matrix multiplication *sparse.multiply()* must be able to read not only the representation of its "own" object of class *sparse*, but also the representation of its argument. Because the argument may be of class *dense*, *sparse.multiply()* must be able to access the private part both of objects of class *sparse* and of class *dense*. Alternatively, it must use public functions to read each element of a *dense* matrix, or be able to make an accessible copy of an object of class *dense*, or use some other inefficient and/or complicated mechanism to achieve its basically simple aim.

The basic problem is that functions which operate on two or more objects of different classes are occasionally very useful, but do not fit the abstract data type model where all operations on an object are functions of its class. The mechanism provided to enable such functions to be written without unacceptable overheads or undetectable breaches of the protection rules is a declaration which specifies that the private members of one class can be accessed by another. For example:

```
class dense : matrix
{
friend sparse;

        /* rest of declaration */
};
```

specifies that functions of class *sparse* can access the private part of an object of class *dense* as well as the public part. Functions of class *dense* cannot access private members of class *sparse* unless the declaration of class *sparse* include an appropriate *friend* declaration.

As in real life friends should be chosen with care.

Postscript

The aim of the class design was to provide the C programmer with a set of tools allowing the structure of a "medium sized C program*" to be expressed more clearly and directly than had been possible before. The practical integration of the class concept with the facilities of the C programming environment was considered far more important than any abstract notion of programming language perfection.

The programmer is not asked to sacrifice "efficiency" to an abstract notion of beauty enforced through restrictions on the use of the language or through the provision of facilities which imply significant compile time or run time overheads. The aim has been to enhance C as a practical systems programming language, not to provide a new programming environment with a higher level of semantics. With the exception of the introduction of the new keywords *class*, *public*, etc. all older C programs preserve their meaning.

The permissive nature of C has been preserved. For example, it is possible to have public data members of a class and to use casts on class pointers, thus ignoring the ideal that all operations on an object of an abstract data type ought to be restricted to a few well defined operations. This ideal is, however, not shared by everybody, and therefore not enforced through the language design. If it is yours you can adhere to it, and because the class pre-processor records dependencies between classes you can easily verify if a program passes this criterion of cleanliness. A version of the class pre-processor, called *class*, will write out the recorded dependencies. Because a more general set of dependencies is recorded the output of *class* can be used for a variety of purposes. Another use would be to determine whether any function of a given class accessed global data, or whether any function of a class took advantage of the ability to access not only the object for which it was called, but also other objects of the same class.

Acknowledgements

Initially the basic technique for the development of the design of C classes was to repeatedly present ideas, designs, and sample programs to whoever could be interested in the problems, and then try to unify the ideas arising from this exercise into a coherent language design. I owe many people thanks for helping me in this. Dennis Ritchie and Steve Johnson proved to be especially good sources of ideas and useful problems. Sandy Fraser and Doug McIlroy influenced the design through many long debates over issues of applicability and style. During the last year or so the class concept was continually refined based on experience gained from the use of early implementations. I owe Sudhir Aggarwal, Jonathan E. Shopiro, and many others thanks for their patience with these sadly imperfect versions of the class pre-processor and their many suggestions for improvements.

References

[1] Unix Programmer's Manual
Seventh Edition, January 1979
Bell Telephone Laboratories

[2] Kernighan, B.W. and Ritchie, D.M.
The C Programming Language
Prentice-Hall 1978

[3] Stroustrup, Bjarne
A Set of C Classes for Co-Routine Style Programming
Computer Science Technical Report CSTR-90
Bell Telephone Laboratories, December 1980

* I think of a medium sized program as consisting of thousands of lines of code. If it is better measured in tens of thousands of lines of C it is considered large - and will most likely suffer from problems of human understanding from which mere programming language structuring facilities can provide little relief.

Appendix A: A Complete Program

The following program is a small test program for the use of derived classes. It consists of three files:

[1] *queue.h* provides the declarations for two classes, *link* and *queue*,

[2] *queue.c* provides the functions for these classes, and

[3] *myqueue.c* gives the program using these classes and containing the *main()* function.

```
/********************* file queue.h ************************************/

class link
{
public:
        class link * suc;
};

class queue
{
        /*
                a one way circular linked list,
                each link points to its successor,
                the tail element points to the head element
        */
        class link * tail;
        int          n
        void         new()    { tail = 0; n = 0; };
        void         delete() { if (n) error("non-empty queue deleted"); };
public:
        class link * get(void);
        void         put(class link *);
};

extern void error(char*);


/********************* file queue.c ************************************/

#include "queue.h"

class link * queue.get()
{
        class link * p;

        if (n--) {
                p = tail->suc;
                tail->suc = p->suc;
                return p;
        }
        else {
                n = 0;
                return 0;
        }
}

void queue.put(class link * p)
{
        if (n++) {
                p->suc = tail->suc;
                tail->suc = p;
        }
        else
                p->suc = p;
        tail = p;
}

void error(char* s)
{
        printf("***** error: %s\n",s);
}



/********************* file myqueue.c ************************************/

#include <stdio.h>
#include "queue.h"

class info : public link
{
        int type;
        void  new()    { printf("new info "); };
        void  delete() { printf("delete info "); };
};


main()
/*
        Put 10 objects of class info onto queue ``q'',
        then take them off again.

        Repeat this 10 times.
*/
{
        class queue q;
        class queue * qq = &q;
        class info * pi;
        int i,j;

        for (i=0; i<10; i++) {

                for (j=0; j<i; j++) q.put(new class info);

                for (j=0; j<i; j++) delete (class info *) q.get();
        }
}
```