MH

1533

Bell Laboratories Cover Sheet for Technical Memorandum

Title- **Classes: An Abstract Data Type Facility for the C Language**

Date- **January 19, 1980**

TM- **80-1273-2**

Other Keywords-

Author	Location	Extension	Charging Case- 39199
Bjarne Stroustrup	**MH 2C-521**	**7393**	Filing Case- **39199-11**

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 [1] is assumed. Appendix A is a complete small C program using classes. The C pre-processor implementing the concept is described very briefly in appendix B.

Pages Text 11	Other 5	Total 16
No. Figures 0	No. Tables 0	No. Refs. 4

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM E-1328)

10 REFERENCE COPIES

<AHO,ALFRED V
<BALENSON,CHRISTINE M
<BARON,ROBERT V
<BECKER,RICHARD A
BROWN,W STANLEY
<CHERRY,LORINDA L
<FELDMAN,STUART I
<FRASER,A G
<GLASSER,ALAN L
<GOLDSTEIN,A JAY
<GRAHAM,R L
+HANNAY,N B
<JOHNSON,STEPHEN C
<KEESE,W M
<KERNIGHAN,BRIAN W
<LUDERER,GOTTFRIED W R
<MARANZANO,J F
+MC DONALD,H S
<MC GILL,R
MCILROY,M DOUGLAS
<MENNINGER,R E
<MORGAN,SAMUEL P
+PRIM,R C
<RALEIGH,T M
<RIDDLE,GUY G
<SZYMANSKI,THOMAS G
TERRY,MILTON E
<WEINBERGER,PETER J
<YANNAKAKIS,MIHALIS
29 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

AAGESEN,JOHN
ABATEMARCO,TERESA M
ABATE,JOSEPH
ACKERMAN,A FRANK
ACKERMAN,J T
ACKROFF,JOHN M

COVER SHEET ONLY TO

AGARWALA,PRAMILA
AHRENS,RAINER B
<AHUJA,SUDHIR R
ALBAGLI,V R
ALBERALLA,RICHARD J
ALCALAY,D
ALEXIS,A D,JR
ALKONS,FREDERICK
ALLEN,JAMES J,JR
ALLISON,C E,JR
AMIN,ASHOK T
AMITAY,N
AMORY,ROBERT W
AMRON,IRVING
AMSTER,S J
ANDERSON,FREDERICK L
ANDERSON,KATHRYN J
ANDERSON,MILTON M
ANDERSON,R E
ANDERSON,R R
ANDREWS,W J
ANTOLICK,DAVID R
APPELBAUM,MATTHEW A
ARCHER,RUSSELL E,JR
ARMSTRONG,D B
ARMSTRONG,F O,JR
ARNDT,DENNIS L
ARNOLD,GEORGE W
ARNOLD,JAMES Q
ARNOLD,PHYLLIS A
ARNOLD,THOMAS F
ARTHURS,E
ARVIDSON,W P
ASELTINE,EDWARD G
ASMUTH,RICHARD L
ASTHANA,ABHAYA
ATAL,BISHNU S
AXELSON,A L
BABU,RAJESH RATILAL
BAGGA,YUDHVEER S
BAILEY,CATHERINE T
BAILY,DAVID E
BAKER,BRENDA S
BAKER,DONN
BAKER,MITCHELL B
BALDWIN,GEORGE L
BALLANCE,ROBERT A
BALLARD,E D,JR
BARBATO,ROBERT R
BARCLAY,DAVID K
BARNHARDT,KARL R
BARNS,R L
BAROFSKY,ALLEN
BARR,DAVID L
BATTAGLIA,FRANCES
BAUER,BARBARA T
BAUER,H C
BAUER,HELEN A
BAUER,WOLFGANG F

COVER SHEET ONLY TO

<BAUGH,C R
BAUMAN,STEVEN M
BAXTER,LESLIE A
BAYER,D L
BEAUMONT,LELAND R
BEBLO,WILLIAM
BECERRA,PEDRO D
BECKER,CURTIS A
BECKER,GARY E
BECKER,JACOB L
BECKER,SHELDON T
BECKETT,J T
BECKNER,MARK W
BEDNAR,JOSEPH A,JR
BEGLEY,ALOYSIUS A
BEIGHLEY,KEITH A
BELANGER,DAVID G
BENCO,DAVID S
BENISCH,JEAN
BENNETT,RAYMOND W
BENNETT,RICHARD L
BENNETT,WILLIAM C
BENOWITZ,P
BENSING,JAMES EDWARD
BERENBAUM,ALAN
BERGLAND,G D
BERKEY,M A
BERKLEY,DAVID A
BERK,DONALD A
BERNHARDT,RICHARD C
BERNOSKE,BEVERLY G
BERNSTEIN,DANIELLE R
BERNSTEIN,L
BERNSTEIN,PAULA R
BERRYMAN,R D
BERZINS,ALEXANDER H
BEYER,JEAN-DAVID
BEYLER,ERIC
BHATIA,RAJIV
BIANCHI,M H
BICKFORD,NEIL B
BILASH,TIMOTHY D
BILOWOS,R M
<BIREN,IRMA B
BISHOP,J DANIEL
BISHOP,VERONICA L
BITTNER,B B
BITTRICH,MARY E
BLAKE,GARY D
BLANCHARD,JOETTE D
BLAZIER,S D
BLECHMAN,RONALD I
BLEICHER,E
BLEIER,JOSEF
BLINN,J C
BLOSSER,PATRICK A
BLUM,MARION
BOCHULA,EDWARD J
BOCK,NANCY E

COVER SHEET ONLY TO

BODDIE,JAMES R
BODEN,F J
BOEHM,EARL W
BOEHM,KIM R
BOESE,J O
BOGART,F J
BOGART,THOMAS G
BOIVIE,RICHARD H
BOLSKY,MORRIS I
BONANNI,L E
BOND,F C
BORDELON,EUGENE P
BORG,KEVIN E
BORISON,ELLEN A
BOSE,DEBASISH
BOSTON,RONALD E
BOSWELL,PAULA S
BOUMA,HERMAN J
BOURNE,STEPHEN R
BOWYER,L RAY
<BOYCE,W M
BOYER,PHYLLIS J
BOYLE,GERALD C
BOZA,L B
BRADFORD,EDWARD G
BRADLEY,M HELEN
<BRANDT,RICHARD B
BRAUN,DAVID A
BREILAND,JOHN R
BRENSKI,EDWIN F
BRESLER,RENEE A
BRIGGS,GLORIA A
BRITT,WARREN D
BROAD,MARTHA M
BRONSTEIN,N
BRONZO,JOSEPH A
BROOKS,CATHERINE ANN
BROSS,JEFFREY D
BROVMAN,INNA
BROWNING,JASON DAVID
BROWN,ELLINGTON L
BROWN,LAURENCE MC FEE
BROWN,MARK S
BROWN,STUART G
BROWN,W R
BRUECKNER,DOUGLAS E
BUCK,I D
BULLEY,R M
BURGESS,JOHN T,JR
BURG,F M
BURIC,MILORAD R
BURKE,MICHAEL E
BURKE,R J
BURNETTE,W A
<BURNETT,DAVID S
BURNET,ROSE M
BUROFF,STEVEN J
BUSCH,KENNETH J
BUTLETT,DARRELL L

COVER SHEET ONLY TO

BUTTON,BEVERLY
BYERLEE,R W
BYORICK,ROBERT S
BYRNE,EDWARD R
CALL,PETER F
CALVERT,KENNETH L
CAMPBELL,JERRY H
CAMPBELL,MICHAEL R
<CANADAY,RUDD H
CANDREA,RONALD D
CANNATA,PHILIP E
CAREY,J E
CAREY,J H
CARRAN,JOHN H
CARRIGAN,RAYMOND J
CARR,DAVID C
CARR,RICHARD G
CARTER,DONALD H
CASPERS,BARBARA E
CASTELLANO,MARY ANN
CATO,H E
CAVINESS,JOHN D
CELLER,GEORGE K
CERMAK,I A
CHAFFEE,N F
CHAI,D T
CHAMBERS,B C
<CHAMBERS,J M
CHANDRA,SATISH
CHANEY,F W
CHANG,JO-MEI
CHANG,LUNG-HSIUNG
CHAO,C
CHAPMAN,LESLEY A
CHAPMAN,W P,JR
CHAPPELL,S G
CHARLES,JOYCE
CHEE,T
CHELLIS,ALICIA L
CHENG-QUISPE,ENRIQUE
CHENG,Y
CHEN,E
CHEN,ROBERT
<CHEN,STEPHEN
CHESSON,GREGORY L
CHEUNG,ROGER C
CHE,HER-DAW
CHILDS,CAROLYN
CHIN,KATHLEEN E
CHI,GOU-CHUNG
CHI,M C
CHODROW,M M
CHONG,PHEE
CHOU,PAO-LO P
CHRISTENSEN,S W
CHRISTENSON,DENNIS A
CHRIST,C W,JR
CHUNG,MICHAEL
CHUNG,TSUNG-JEN BEN
...
1461 TOTAL

\+ NAMED BY AUTHOR > CITED AS REFERENCE < REQUESTED BY READER (NAMES WITHOUT PREFIX WERE SELECTED USING THE AUTHOR'S SUBJECT OR ORGANIZATIONAL SPECIFICATION AS GIVEN BELOW)

MERCURY SPECIFICATION..

COMPLETE MEMO TO:
127-SUP

COVER SHEET TO:
12-DIR 13-DIR 127

COPLSI = COMPUTING/PROGRAMMING LANGUAGES/SIMULATION
COPRMP = COMPUTER PROGRAMMING METHODOLOGY
COPRSQ = COMPUTER SOFTWARE QUALITY
UNPLCL = C LANGUAGE

HO CORRESPONDENCE FILES
HO 1A-127

TM-80-1273-2
TOTAL PAGES 16

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.
4. INDICATE WHETHER MICROFICHE OR PAPER IS DESIRED.

PLEASE SEND A COMPLETE

( ) MICROFICHE COPY ( ) PAPER COPY

TO THE ADDRESS SHOWN ON THE OTHER SIDE.

Bell Laboratories

Subject: **Classes: An Abstract Data Type Facility for the C Language**
Case- 39199 -- File- 39199-11

date: **January 19, 1980**

from: **Bjarne Stroustrup**

TM: **80-1273-2**

MEMORANDUM FOR FILE

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 large 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. In C the declaration of a *stack* might look like this:

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

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 of members of a class are private, that is they can only be used by the implementor of the class in functions from that class, unless they appear in the public part of the declaration. Names in the public part, that is appearing after the keyword *public*, provide the interface to users. The functions named in the class declaration must themselves be declared for the declaration to become meaningful, for example the two public functions:

```
stack.push(c) 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:

```
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 the declaration

```
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 's' to *c*.

The Basic Class Concept

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 (size) short size;
{       char * s;
        char * min, * top, * max;
        new();
        delete();
    public:
        push();
        char pop();
};
```

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 when an object of *class stack* is declared. For example:

```
class stack s1(100), s2(200);
```

Such arguments are passed to *new*, and the declaration of this function provides the interpretation of them. In this case:

```
stack.new()
{   s = alloc(size);
    top = min = s;
    max = s+size-1;
}
```

The *new* function can be declared without any specification of its argument list because that list has already been provided in the class declaration.

This example assumes that *alloc(n)* returns a pointer to a vector of *n* characters from the free store. This, however, creates a new problem. Because we cannot 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 *s* when an object of *class stack* is deleted. This is done by defining a parameterless function called *delete*.

```
stack.delete()
{   free(s);
}
```

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 return a value. Furthermore, it cannot be explicitly called, and it is illegal to declare it public. The same restrictions apply to the *new* function.

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 executed in a function. The vector denoted by *s* will however always be allocated on the free store by *class stack's new* function*.

This style of declaration can also be used to declare *static* and *extern* class objects from classes without a *new* function. The declaration of *extern* or *static* objects of a class with a *new* function is illegal.

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.

Three kinds of storage can be used in a C program:

[1] Automatic storage is allocated on the stack on entry into a function and deallocated on the return from it. Variables, for example class objects, declared in a function use this kind of store unless one of the other kinds is explicitly specified.

[2] Static storage is allocated at compile time and never deallocated during the lifetime of the program. Variables that are not declared in a function or are explicitly declared *static* or *extern* use this kind of store.

[3] Free storage is explicitly allocated using the *new* operator, and can only be deallocated using the *delete* operator. The *new* operator can be used to generate objects of all simple types, structures, and class objects. It can also generate vectors of such objects. For example:

```
char * p = new char [16];
int * ip = new int;
```

To obtain more store *new* may need help from the operating system. For example, on a Unix system *new* will typically use the C library function *malloc* which in turn may use the Unix system call *sbrk* [4].

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

The only way of destroying an object generated by *new* 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.

If a *new* function does not need arguments the argument list need not be present, or it can be empty. For example:

```
class foo x;
```

and

```
class foo x();
```

are both valid, and their meanings identical*.

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 from the list and assigns it to q
*/
{  new();
   public:
      class link * left, *right;
      put();
      class link * get();
};

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.

* To avoid confusion between declarations of class pointers and objects and declarations of functions returning values of these types the class pre-processor insists on the use of the keyword *extern* in extern declarations of functions. This is sufficient to make the syntax unambiguous.

In the declaration of a class function a member of that class need not be qualified by the class name, but such qualification is not illegal. For example, *link.new* can be declared like this:

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

without any change of meaning, should the programmer prefer the redundant complete qualification of class member names.

A private name from a class can only be used in the declaration of a function from that class. So, when compiling a source file in which a class is declared, but no function from that class is declared, the private names from that class need not be remembered by the compiler. On the other hand, where a class function is declared the class declaration must appear before the function declaration. This implies that a one pass compiler cannot, when reading the class declaration, know whether it needs to remember the private names. In the interest of efficient compiling you are therefore requested to specify in advance that you want to declare a function from a class. For example:

```
classdef x;

#include "x.h"

x.f()
{
    ...
}
```

using the new keyword *classdef* for this purpose. Here, the the declaration of *class x* itself is placed in the "header file" *x.h*.

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];
       clear();
       link.put;
       link.get;
};

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];
        clear();
};
```

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. For example, using the declaration of *class stack* from the previous section, we can write:

```
class mystack(size) short size;
: public stack(size+10)
{
};
```

which specifies that objects of class *mystack* need the argument *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(a,b,c); int a,b,c;
{
};

class derived(a,b,c) int a,b,c;
: base(a,f(b),2);
{
};
```

To ensure proper initialization of an object of *class derived base.new* is executed as if it had been the first executable statement in *derived.new:* .

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

So, as a result of the declaration

```
class derived x(1,2+3,4);
```

derived.new is called with the argument list *(1,2+3,4)*, 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 co-routine 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. The following sections will examine derived classes in greater detail.

Call and Return Functions

A base class can be seen as providing a special environment for the derived class to work in. In general such an environment cannot be represented statically, but aspects of it have to be set up each time a function from the derived class is entered. For example, a base class may be used in a multi-process system to ensure that only one process at a time can execute functions from classes derived from it:

```
class monitor
{   short count;
    struct queue q;
    sleep_on_queue();
    wake_from_queue();
    call();
    return();
};

monitor.call()
{   if (count++) sleep_on_queue();
}

monitor.return()
{   if (--count) wake_from_queue();
}
```

If all functions from a derived class call *monitor.call* as their first statement, and *monitor.return* as their last statement, then an appropriate interlock is maintained by *monitor*. Explicit calls to these functions are, however, not necessary because the function names *call* and *return* are treated specially in a class declaration. If a function named *call* exists in a base class, then it is guaranteed to be the first statement executed in every function of a derived class. Similarly, *return* will be the last, so the interlock in the *class monitor* example above will be correctly maintained without any effort from the writer of a class derived from *class monitor*. For example:

```
class indivisible: monitor
{   int data;
    public:
        read();
        write();
}
```

is a way of providing an interlocked data structure.

Like *delete*, *call* and *return* functions take no arguments, return no value, and cannot be called explicitly.

If you derive a class containing a *call* or a *return* function from a base class which has one or both of these functions declared then everything will still work fine. For example, a *call* function from a derived class will simply execute the *call* function from the base class, if any, as its first statement just like any other function.

In principle the writer of a derived class does not need to know if *call* or *return* functions are declared for the base class. An example of the use of a *call* is to produce a trace of all calls to derived classes.

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;
      update();
};

class safe: unsafe      /* validate request before
                           using ''unsafe''
                          */
{  public:
     int pub;
     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:

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

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

will accept a pointer to a class derived from class x as argument, provided all 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:
       multiply();
};

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

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 only *p* and *q* are pointers to objects of the derived classes *sparse* and *dense*.

Derived Classes and Unions

Derived classes can sometimes be used as an alternative to *unions* when defining data structures. For example:

```
struct common
{   int a;
    union {
      int b;
      struct {
         int x;
         int y;
      } sval;
    } uval;
};
```

could be written as

```
class cc
{   public:
      int a;
};

class cb: public cc
{   public:
      int b;
};

class cx: public cc
{   public:
      int x;
      int y;
};
```

This use of classes enables the compiler to allocate the minimum amount of store in all cases, whereas for unions it must always allocate enough store to hold the largest alternative. Furthermore, members can be accessed more simply:

```
class cb c1;
class cx c2;

c1.b = 2;
c2.x = 3;
```

instead of

```
struct common s1, s2;

s1.uval.b = 2;
s2.uval.sval.x = 3;
```

Both styles of declaration look overelaborate. The *public* keyword in the class declaration is an eyesore, but necessary in those cases (hopefully the majority) where not all member names are public.

Acknowledgements

The basic technique for the development of the design of C classes was to repeatedly present ideas, early 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 Richie and Steve Johnson proved to be especially good sources of ideas and useful problems.

MH-1273-BS-unix

B Stroustrup

Bjarne Stroustrup

Att:
  References
  Appendix A
  Appendix B

References

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

[2] Stroustrup, Bjarne
The Design of an Abstract Data Type Mechanism for the C Language
In preparation

[3] Stroustrup, Bjarne
A Set of C Classes for Co-Routine Style Programming
In preparation

[4] Unix Programmer's Manual
Bell Telephone Laboratories (January 1979)

Appendix A: A C 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.x* provides the functions for these classes, and

[3] *myqueue.x* 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;
        short n;
        void new();
        void delete();
public:
        class link * get();
        void put();
        void put_head();
        short elements();
};
```

void is a keyword recently introduced into C. Placed as the type specifier for a function it indicates that the function cannot return a value. This allows for better type checking, and - in the C pre-processor - marginally better code generation.

This header file is then shared between *queue.x* and *myqueue.x*.

```
/*
        file queue.x
*/

classdef link;
classdef queue;

#include "queue.h"

void error(s) {
        printf("***** error: %s0,s);
};

void queue.new()
{
        tail = 0;
        n = 0;
};

void queue..delete() {
        if (n) error("nonempty queue deleted");
};

short queue.elements() {
        return n;
};

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(p) class link * p;
{
        p->suc = (n++) ? tail->suc : p;
        tail = p;
} /* put */

void queue.put_head(p) class link * p;
{
        p->suc = (n++) ? tail->suc : p;
        tail->suc = p;
} /* put_head */
```

File *myqueue.x* has a structure which should be similar to that of quite a few user programs using standard classes.

```
/*
        file myqueue.x
*/

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

classdef info;

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

info.new() {
        printf("new info ");
};

info.delete() {
        printf("delete info ");
};


main() {
        class queue q;
        class queue * qq = &q;
        class info * pi;
        short i,j;
        for (i=0; i<10; i++) {
                for (j=0; j<i; j++) {
                        pi = new class info;
                        q.put(pi);
                };
                for (j=0; j<i; j++) {
                        pi = (class info *) q.get();
                        delete pi;
                };
        };
} /* main */
```

Appendix B: Implementation of Classes

The initial implementation of C classes is a pre-processor that translates a C program with classes into an equivalent C program without classes. This "intermediate representation" relies heavily on structures and casts. A class is translated into a *struct* plus a set of *extern* functions operating on it.

Because the pre-processor is a lot simpler than a complete compiler implementing classes would be you will find some restrictions on the use of classes. The most basic restrictions - the ones that are least likely to be removed soon - are listed below:

[1] Class objects can only be declared as *auto* variables in the outermost block of a function.

[2] Only objects of classes without a *new* function can be declared *static* or *extern*.

[3] Class objects cannot be members of structures or classes. Use pointers or derived classes.

[4] Declarations of classes, of *extern* class objects, and of class pointers cannot be placed inside function declarations.

[5] Assignment is not defined for class objects, and a function cannot return a class. Use pointers.

[6] The syntax for class member declarations is restricted. For example, no unions and no *struct* declarations.

[7] If a name from a base class is made valid for the derived class using a

class_name . member_name

specification, then - as a side effect - all public names from the base class becomes valid in the derived class.

[8] The handling of class member references is not fully general. For example, a long chain of references "*a->b->c->d->e*" can be too complicated for the pre-processor. Similarly, an expression yielding a class pointer "*(f()->v[x])->s*" can be too complicated. The pre-processor will detect such problems and issue appropriate error messages. Use simpler expressions if this happens.

[9] The pre-processor cannot resolve the types involved in a class reference expression to the satisfaction of *lint* in all cases. Use unique names if this annoys you.

[10] Typedef cannot be used for classes.

[11] Vectors of class objects cannot be declared.

The following new keywords are used:

```
call    class    classdef  delete    public    new
```

The following names are generated by the preprocessor:

```
_this       - the ``this'' pointer.
_return     - a label denoting the end of a function.
_result     - a variable for holding the value
              to be returned by a function
_private    - a vector of characters occupying the spaced used by
              a class objects private data.
_base       - the data structure for a base class.
```

In addition to this the pre-processor generates a name for each public class function. This name is a combination of the class name and the member name prefixed by the character '_'. These names are hoped to be unique using only their first 6 characters. Complaints from *cc* or the loader referring to strange names of the form "_ab_cd_efghijkl" will originate from this combination of two names into one.

A C compiler which includes the class preprocessor can be found in /usr/bin/class. For further documentation see the manual page for *class*.