C++ Technical Notes–Number 1

H. Kanner

Development Systems Group

29 February 1988

This is the first of a series of technical notes on the C++ language. Because there is as yet only one book on the language, *The C++ Programming Language* by Bjarne Stroustrup, and many sections of the book have been written in a difficult to follow style, I felt that it might be of use to write some notes on various aspects of the language with which I initially had difficulty. By the way, the cited book will be referred to here and henceforth as The Book.

It has often been said that the ultimate definition of a programming language is the official compiler. That is, the semantics of a piece of source code is really understood by inspecting the output of the compiler. In the case of the AT&T C++ compiler, this task is made easier by the fact that this output is C code. I cannot too strongly recommend to readers that they try to clear up linguistic obscurities by constructing the simplest possible test cases, running them through *cfront*, and looking at the output. One warning: do not try this with an *inline* function.

This first note deals with *references*. I anticipate a minimum of two more notes, to be published when I feel I understand the subjects sufficiently. They will respectively cover *constructors* and *destructors*, with emphasis on their storage management aspects, and overloaded operators, with emphasis on the distinction between defining them as *members* or *friends*.

I had a little difficulty understanding why *references* had been introduced into C++. They are an exception to the symmetry of the terrible C notation for declarations, in which one can at least say that operators such as *, (), and [] have the same meaning when used in a declaration as when used in an expression. That is,

```
char *s;
char* t;
char c = *t;
```

all use the * to denote dereferencing a pointer. The first of the above three lines should literally be read: "If you were to dereference `s`, you would get a `char`." It is true that the compiler does not care where the white space is, and Bjarne prefers the style of the second line, which he likes to state as "`t` is a pointer to `char`," as if `char*` represented the type "pointer to `char`." This is very informal. In fact he cautions that if you write

```
char* s,t,u;
```

only the first of these will be a pointer to `char`; the second two will be declared as `char`.

A *reference* uses the & symbol. But `int&` means reference to int, `&x` means address of x, and neither of

```
int& x;
int &x;
```

can be interpreted as "If you were to take the address of x, you would get an `int`."

I now whet the reader's interest by making a promise. By the end of this note, I will have illustrated that *references* make it possible for the following to be perfectly legal C++:

```
int i;
f(i) = 10;
```

I introduce references by exploring to the point of tediousness what we really mean when we say `int i`. The symbol `i` is regarded as a synonym for the address, to be determined at some future time, of a bucket that can hold an `int`. The binding of `i` to this hypothetical and not yet known address occurs at declaration time and is permanent for the scope of `i`. Every subsequent use of `i` in the code implies an automatic dereferencing of `i`. That is, a use of it on the right side of an assignment means "fetch from the address it represents," and a use of it on the left side means "store into that address." This behavior is so intuitive that most language definitions do not formalize it; Algol 68 is a notable exception. However, it must be kept in mind if *references* are to be understood.

Now, to introduce the concept:

```
int i;
int& r = i;
```

means that `r` is declared to be a *reference* to `int`, and initialized to be equivalent to i. That is, r is another synonym for the same address as is represented by i. Like any other variable identifier, it is automatically dereferenced as described in the paragraph above. Therefore, any subsequent use of `r` in the left or right side of an assignment means, respectively, a store to or fetch from `i`. So

```
r = 2; // puts 2 into i
int j = r // now puts 2 into j by fetching it from i, and
r++; // increments i. Finally,
     // &r is a pointer to (the dereferenced) r,
     // that is, it is identical to &i.
```

Note that initialization of the *reference* variable is mandatory. It is illegal to write

```
int& r;
```

On page 56 of The Book is introduced a complication. What would be the meaning of

```
int& sillyref = 5;
```

Bjarne gives a C equivalent as

```
int *sillyrefp;
int temp
temp = 5;
sillyrefp = &temp;
```

that is, a temporary is allocated to hold the literal, and the *reference* variable is initialized to the address of this temporary. If any reader can find a use for this, please let me know. My theory is that since an initializer is defined grammatically to be a *constant expression*, the construct is legal and it was easier to define harmless semantics for it than to figure out how to disallow it.

The next topic, an interesting one, is the use of *references* as formal parameters for functions. In The Book it is stated in several places that the semantics of parameter passing is the same as that of initialization. In one place, The Book goes further and states that the same semantics also applies to the returning of a value by a function. The Book is clearly distinguishing the above operations from assignment. These are the kind of statements that are prone to go into one eye and out the other. After all, it is hard to discern a semantic difference between

```
int i = 7;
```

and

```
int i;
i = 7;
```

When explicitly defining initialization and assignment for classes, the distinctions between the two may be subtle. However, as has already been shown, **initialization** of a *reference* is a permanent setting of the reference to the address of the initializer, and **assignment** to a *reference* consists of assigning **into** the bucket denoted by that address. This distinction is not subtle in the least. Therefore, if a formal parameter of a function is a reference, then at call time, that parameter is permanently set (permanently for the duration of execution of the function, of course) to the address of the corresponding actual parameter. In the code of the function, any use of the formal parameter causes an automatic dereferencing, so that the code can directly fetch from and store into the actual parameter. Let us compare to C. If we are so evil as to wish p to be an output parameter of function f, we must write, for example:

```
void f(int *p)
{
        *p = 17;
}
```

and call the function by

```
int i;
f(&i);
```

Note how this works. The address of i is passed as a value parameter, and inside of the function an assignment is made through that address by use of the dereferencing operator. In C++, an equivalent function could be written:

```
void f(int& p)
{
        p = 17;
}
```

and called by

```
int i;
f(i);
```

Any Pascal programmer will immediately see that this is equivalent to making p a *var* parameter. Despite this being routine usage in many languages, Bjarne is nervous about using it, and recommends against it. His argument is that it makes code hard to read, because one cannot tell from the formal parameter prototype alone whether p is an input parameter, an output parameter, or both. So he recommends that output from functions be obtained only from return values or explicit pointers, as it would be done in C. In fact, if a reference is used only to conserve storage, as when passing a large array as an input parameter, he recommends making it clear that the function will not write to the array by writing the function header as

```
function(const arraytype& arg),
```

indicating that the function regards arg as a *reference* to a read-only variable.

One might wonder, in view of the recommendations just quoted, why *references* were introduced into C++. One immediate answer is in their application to overloaded operators. If the + operator is to be extended to apply to members of a newly defined class, say *matrix*, one wants to be able to write

```
a = b + c;
```

where all three of the above are instances of *matrix*. In order to be able to do this and yet avoid physically copying the arguments, the operator definition, which is that of a function, must have `matrix&` as the types of its arguments and return value. The declaration would be:

```
friend matrix operator+(matrix&, matrix&);
```

If the language did not have *references*, it would be necessary to write the very ugly

```
a = &b + &c;
```

This topic will be discussed further in the forthcoming Technical Note on operator overloading.

I am now ready to keep the promise made earlier in this Note. I will show you how to define a function `set` that can be used as follows:

```
main()
{
        int i;

        set(i) = 10;
}
```

where the effect of the function call is the same as the assignment

```
i = 10;
```

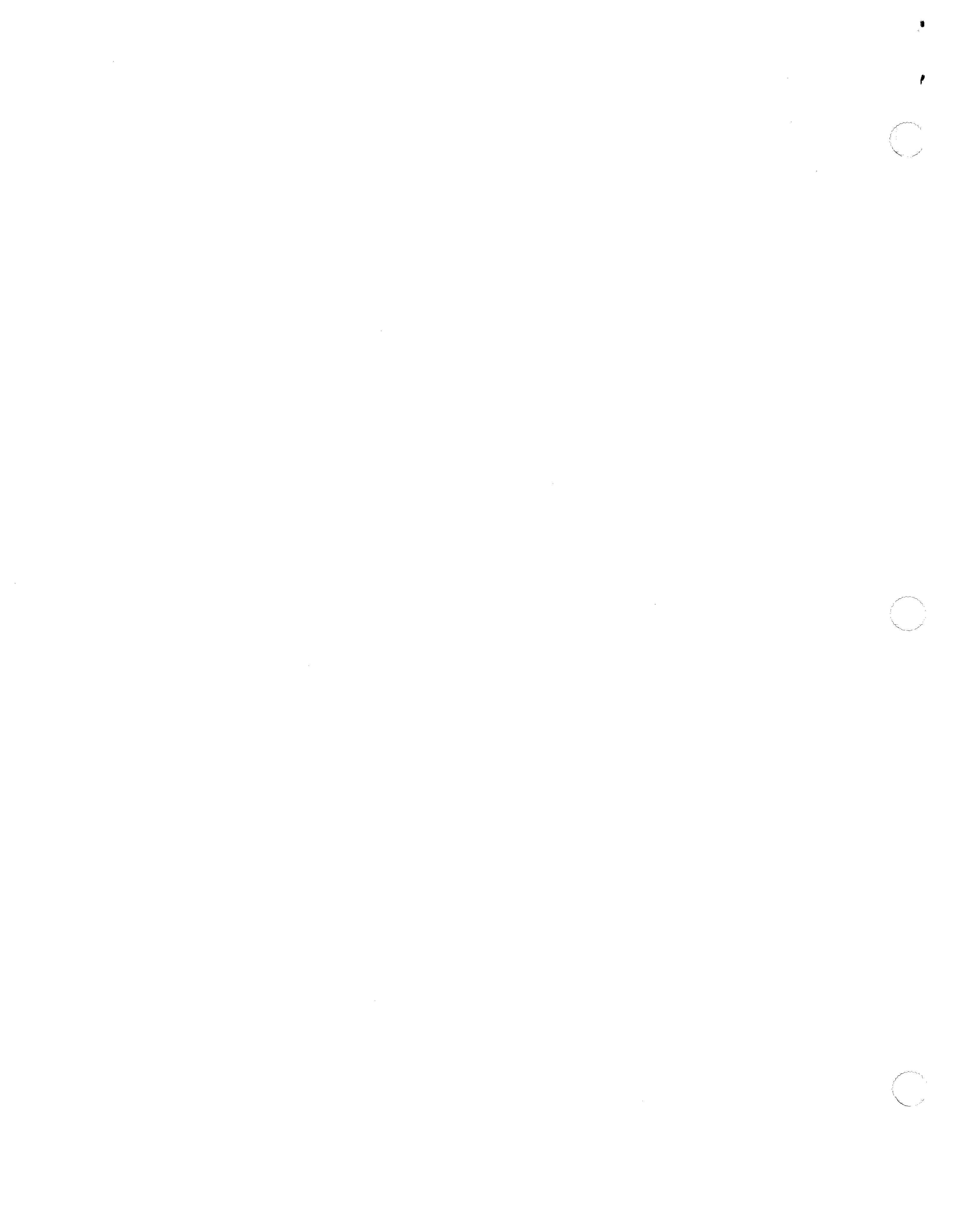
The function definition is

```
int& set(int& p)
{
        return p;
}
```

I do not claim that this function is useful. It was chosen as the simplest function that illustrates the point. For a useful example, see pages 57–58 of The Book.

Here is a sanitized version of the C code generated by the C++ compiler for the example above. I have changed only the names, not, as the phrase goes, "to protect the innocent," but to increase clarity:

```
int *set(int *ptr)
{
        return ptr;
}

main()
{
        int i;

        (*set( (int *)(&i) ) ) = 10;
}
```



C++ Technical Notes–Number 2

H. Kanner

Development Systems Group

14 March 1988

1. Introduction

The subject matter of this note is the pair of special functions called *constructors* and *destructors*—why they exist, how they are represented in C code, and some of the arcana of using them.

Section 2 begins with a discussion of why they were invented in the first place (my opinion, of course), and some of the details regarding the implicit calls that are made to them. An example is presented of the C code produced by a small C++ program that has classes with constructors and destructors. As was done in the previous Technical Note, the simplest examples I can devise are used to illustrate the mechanisms. That these examples may be useless is irrelevant. This is in contrast to the style in The Book, where the examples in general do something useful., but are consequently harder to read. In fact, now that I have been working with the source to the C++ compiler, I notice that some of the examples in The Book look strangely familiar and realize that they have been abstracted from the compiler. The reader who is already familiar with the basic concepts may with to skip section 2.

Section 3 discusses the passing of parameters to constructors for class objects that "ride along" with the object being allocated. There are two cases: a derived class whose base class has a constructor requiring parameters, and a class containing at least one member field that is itself a class having a constructor that requires parameters. An example is given of parameter passing in the case of multiple inheritance, as supported in Release 2.0.

Section 4 deals with the techniques for writing a constructor/destructor pair in which one does one's own storage management—a useful technique when generating and releasing large numbers of small objects. Because a change in the facilities for doing this has been implemented in Release 2.0, both the old and new techniques are illustrated.

Please note that AT&T requires that our usage of Release 2.0 be kept confidential. Therefore, this Technical Note has been classified Apple Confidential.

2. Why and how

The purpose of a *constructor* can be stated as selective initialization of fields of a class object. This may include storage allocation if any of these fields are pointers to areas requiring allocation. The purpose of a *destructor* can be even more glibly stated as that of undoing, when necessary, whatever was done by the corresponding *constructor*. This usually means storage deallocation.

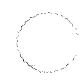
Let us look at a simple example:

```
struct silly {
     int x;
     int y;
} simplestruct = { 1, 2 };
```

Well, the structure has been declared, and fully initialized. So what more might we need? In the first place, if you try this example on any current release of C++ from AT&T and if `simplestruct` is an automatic variable, you will get the diagnostic: "Sorry, not implemented." Secondly, suppose the structure had, say, twenty fields and you only wished to initialize one or two of them that are in the middle. You would have to write an unnecessarily long initializer. Finally, if the structure becomes a class and the data fields are in the private part, then direct assignment to them is forbidden, and you would be forced to invent a (public) member function that does the initialization for you. That is all that a constructor is. It is a function that has the same name as the class or structure name and in the case of a class is declared as a public member function. There is a syntactic requirement that **no** return type be given in its declaration and definition. If, as is often the case, it is a trivially simple function, it is useful to write its definition at the point of declaration, which makes it compile as an *in-line* function. The magic thing about a constructor is that it is called automatically at the point in execution at which storage for the created object is to be allocated. This will be explained further after the next example. Almost everything which has been said about constructors applies also to destructors. The syntactic differences are that the name of a destructor is the class name preceded by the symbol "~" and that a destructor must be parameterless. A destructor is automatically called at the point in execution at which the storage for the object is to be deallocated.

Let us now look at an example which illustrates all of the simple cases: a structure that has neither a constructor or destructor, one that has both, and instances of each created respectively as automatic variables and via dynamic allocation.

```
struct a {
int x;
        };

struct b {
        int y;
        b(int);
        ~b();
};

b::b(int yy)    // This is so simple, it could have been
{               // in-line,but then the generated C code would
        y = yy; // have been too hard to read.
}

b::~b(){}       // Obviously, the same comment applies to
                // this empty function.

main()
{
        a obja;
        b objb(5);      // This is shorthand for: b objb = b(5);
        a* aptr;
        b* bptr;

        aptr = new a;
        bptr = new b(6);
        delete aptr;
        delete bptr;
}
```



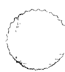
Several observations should be made about this example before displaying the C code generated from it. The in-line form of the declaration for the constructor would read:

```
b(int yy) {y = yy;}
```

The notation `b(5)`, when used as a term in an expression, as in the comment opposite `b objb(5)`, represents a literal which is an instance of class `b`, with its internal variable initialized to the value `5`. The destructor is written as a null function, because the objective here is to illustrate some implicit actions of destructors. Of course, a programmer is free to write any code he wishes in a constructor or destructor; it does not even have to be relevant. In fact, when studying this business, I was prone to write destructors whose entire body was `printf("destructor called now\n");`! To end on a more realistic note, a destructor is useful only to delete objects which were dynamically allocated by code within the body of a corresponding constructor.

The following C code corresponds to the C++ example just given. It has been edited to remove irrelevant material, to make simplifications and clarifications without changing the semantics, and to use the same identifiers as in the original source instead of the encoded identifiers produced by *cfront*.

```
struct a {
   int x;
};

struct b {
   int y;
};

extern void *_new(); /* library function    */
                     /* that calls malloc() */

extern void _delete(); /* library function  */
                       /* that calls free() */

struct b *b_constructor(this, yy)
struct b *this;
int yy;
{
    if (this == 0)
       this = (struct b *)_new( (long) sizeof (structb));
    this->y = yy;
    return this ;
}

void b_destructor(this, free )
struct b *this;
int free;
{
   if (this)
      if (free)
         _delete( (void *) this);
}

int main ()
{
   struct a obja;
   struct b objb;
```

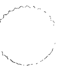


```
        struct a *aptr;
        struct b bptr;

        b_constructor(&objb, 5);
        aptr = (struct a *)_new( (long) sizeof (struct a));
        bptr = (struct b *)b_constructor( (struct b *) 0, 6);
        _delete((void *) aptr)
        b_destructor(bptr, 1) ;
        b_destructor(&objb, (int) 0);
}
```

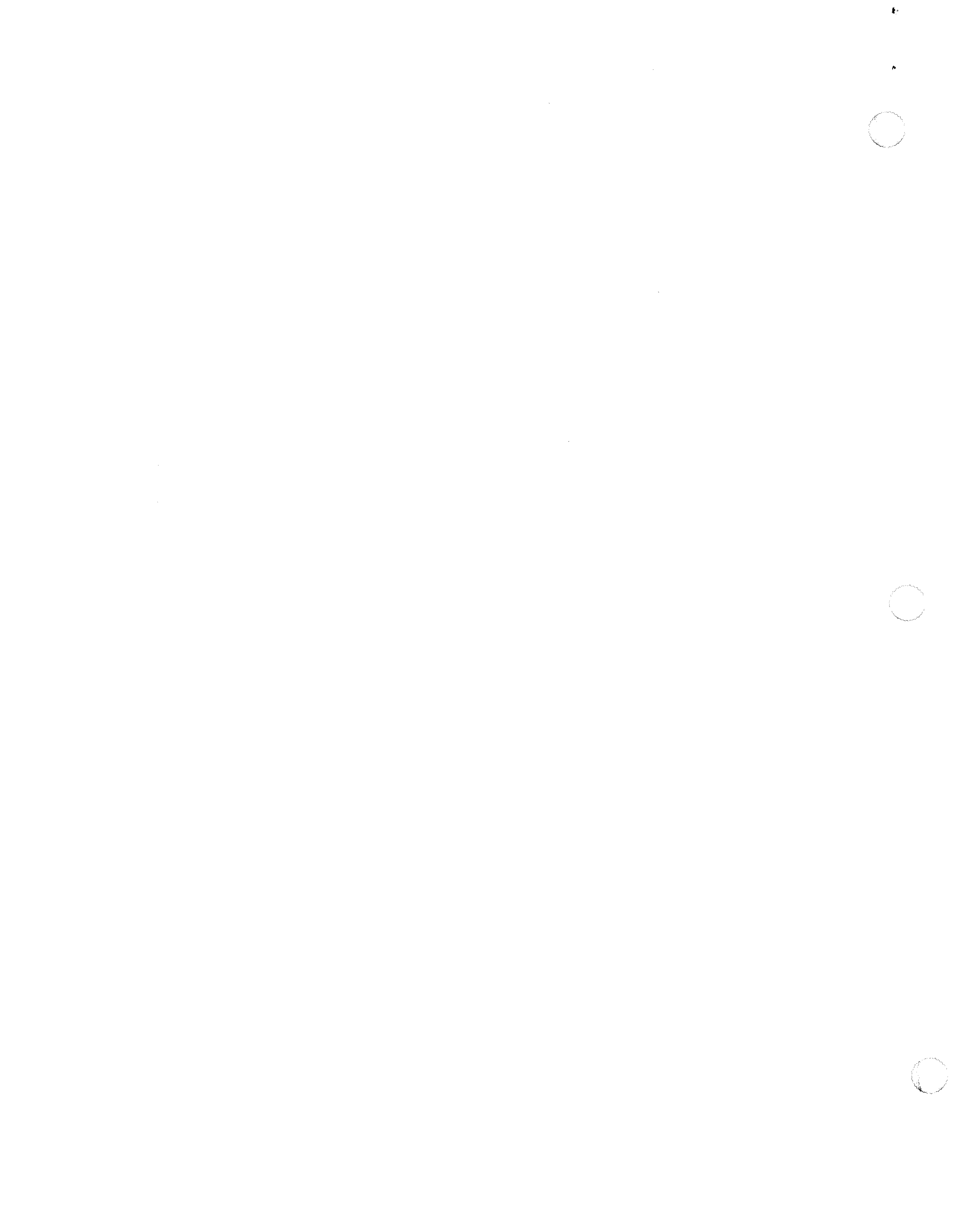
The above code tells us a number of things about constructors and destructors. We see that when an object of a class or structure that has a constructor is allocated by declaration, there is a call of the constructor prior to any other code, and the (implicit) first parameter of the constructor is the address of the allocated object. We also see that although a constructor returns a pointer to its class, in this case the return value is discarded. We next see that when an object is allocated by use in the C++ source of the `new` operator, the constructor is called with the first parameter being zero, and the constructor returns the pointer to the allocated object. Compare this to the case of a class/structure without a constructor, where the effect of the `new` operator is to generate a call to the `_new()` function. Next, looking at the code of the constructor itself, we see that prior to the execution of any code written by the user, it has a conditional call of the `_new()` function, this call taking place if and only if the first parameter is zero. Some similar facts emerge about destructors. A destructor is called implicitly when an object that was allocated by declaration goes out of scope. In this case its second parameter is zero. If a dynamically allocated object is released by use of the C++ `free` operator, the destructor is called with the second parameter having the value `1`. In both cases, the first parameter is a pointer to the object. Now, looking at the code of the destructor itself, we see that it has a call to the `_delete()` function that is conditioned on both parameters being non-zero. That it requires the first parameter to be non-zero is merely an inhibition against "destructing" a null object. This does not protect programmers particularly, because there is nothing to keep them from making a fatally destructive modification of a pointer to an object prior the action that invokes the destructor.

The code that follows is what Release 2.0 of *cfront* actually generates. By comparing to the previously shown code, one can see that identifiers except for structure or class names have been extended, and that tricky names have been invented for constructors, destructors, and the allocation and deallocation functions. Also, there appears to be a redundant test of `__0this` in the destructor code (reference to line 16). In actuality, if the destructor had a body, the the first test would determine whether the body is to be executed (don't do anything if the pointer is zero), and the second test would be relevant if the pointer was not zero to start with, but was zeroed by the body itself (don't delete via a zero pointer).

There is one major point illustrated by this code which requires explanation. At the start of the definition of the main program (reference to line 19), there is an extra "{" before the declarations, and this is preceded by a call to a function called `_main()`. This all has to do with what are called in the trade *static constructors*. As was already seen, if an object is allocated by declaration, its constructor must be called before any further code is executed. This implies that if the object is *static*, that is its *corpus* is not on the stack, the constructor must be called before any code in its scope can be executed. The way this has been handled is to arrange for the call of all such constructors before any code whatsoever is executed. The function `_main()` contains, as pointers to functions, a list all these constructors imbedded in a *for* loop. Thus executing this loop will call each such constructor once. The way in which *cfront* builds the contents of `_main()` will not be discussed here other than to say that AT&T has provided two techniques, one of them machine independent but time consuming (compile time). We have implemented our own machine dependent technique.



```
#line 1 "constr1.c"

/* <<cfront 2.0 (beta) 12/15/87>> */

#line 1 "constr1.c"
void *_vec_new ();
#line 1 "constr1.c"
void _vec_delete ();
typedef int (*__vptp)();

#line 1 "constr1.c"
struct indirect { /* sizeof indirect == 2 */
short indirect_dummy__8indirect ;
};
struct b {  /* sizeof b == 4 */

#line 6 "constr1.c"
int y__1b ;
};

#line 9 "constr1.c"
extern void *__nw__Fl ();

#line 9 "constr1.c"
extern void __dl__FPv ();

#line 11 "constr1.c"
struct b *__ct__1bFi (__0this , __0yy )
#line 11 "constr1.c"
struct b *__0this ;
int __0yy ;

#line 12 "constr1.c"
{ if (__0this == 0 )__0this = (struct b *)__nw__Fl ( (long
)(sizeof (struct b))) ;
__0this -> y__1b = __0yy ;
return __0this ;
}
;

#line 16 "constr1.c"
void __dt__1bFv (__0this , __0_free )
#line 16 "constr1.c"
struct b *__0this ;

#line 16 "constr1.c"
int __0_free ;
{ if (__0this )if (__0this )if (__0_free )__dl__FPv ( (void
*)__0this ) ;
}
;
struct a {  /* sizeof a == 4 */

#line 2 "constr1.c"
int x__1a ;
};
```



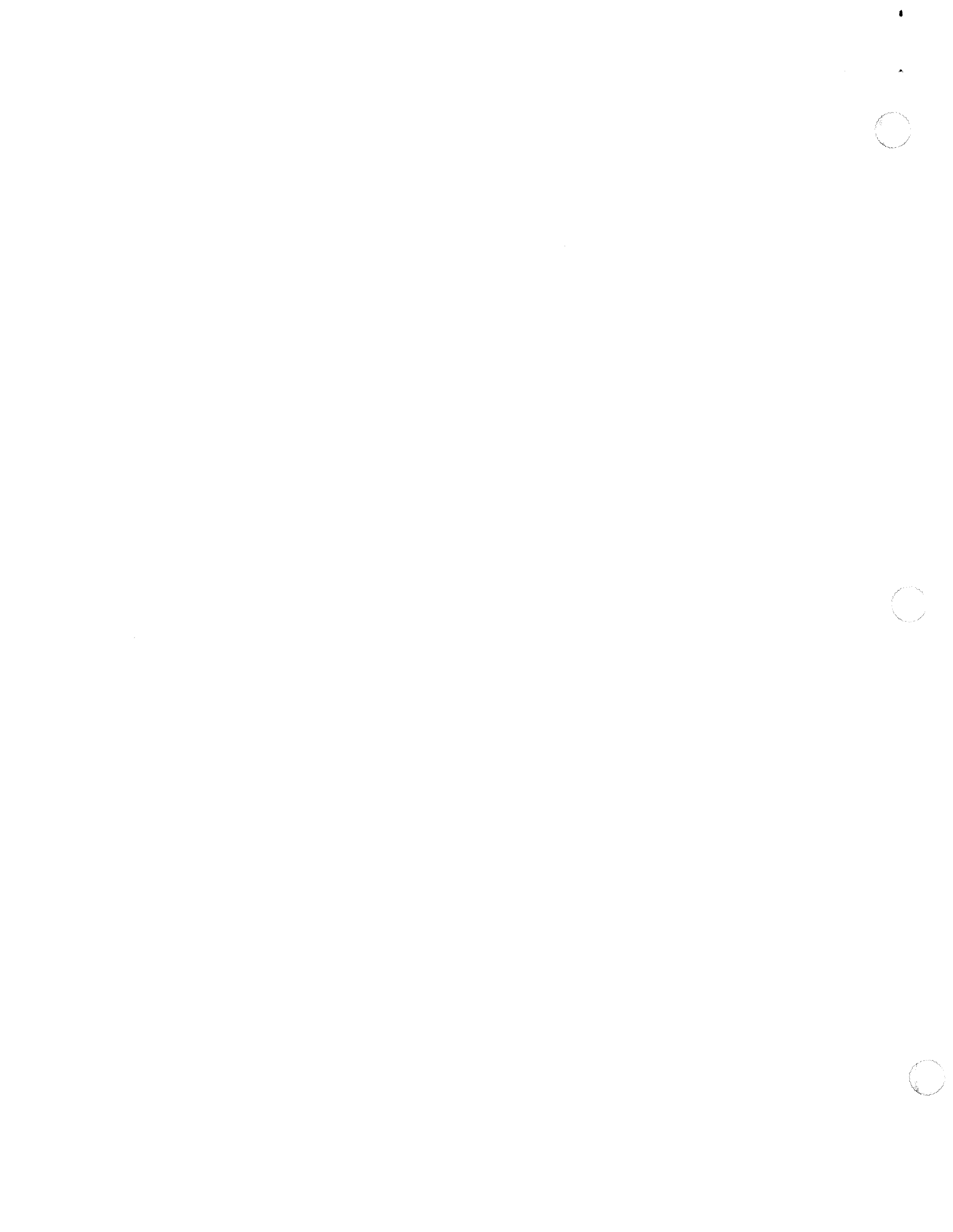
```
#line 19 "constrl.c"
int main (){ _main();
#line 20 "constrl.c"
{
#line 21 "constrl.c"
struct a __1obja ;
struct b __1objb ;
struct a *__1aptr ;
struct b *__1bptr ;

#line 22 "constrl.c"
__ct__1bFi ( & __1objb , 5 ) ;
#line 26 "constrl.c"
__1aptr = (((struct a *)__nw__Fl ( (long )(sizeof (struct a
))) ));
__1bptr = (struct b *)__ct__1bFi ( (struct b *)0 , 6 ) ;
__dl__FPv ( (void *)__1aptr ) ;
__dt__1bFv ( __1bptr , 1) ;
__dt__1bFv ( & __1objb , (int )0 ) ;
}
};

#line 30 "constrl.c"

/* the end */
```

3. Further parameter passing

There is another way in which parameters may be passed to constructors, a mechanism described in the syntax as a *base-initializer*. Two circumstances require the use of this mechanism. The first is that of a class member which is itself a class that contains a constructor. The (mandatory) constructor for the outer class contains as part of its definition a parameter list for the member's constructor. The second is that of a base class which has a constructor. A constructor is mandatory for any derived class, and the latter contains as part of its definition a parameter list for the base's constructor.

Here is an example of the first case:

```
class inner {
      int y;
public:
      inner(int);
};

class outer {
      int x;
      inner aninner;
      inner anotherinner;
putlic:
      outer(int);
};

outer::outer(int xx) : anotherinner(xx+1), aninner(xx-1) {
      x = xx;
}
```



```
inner::inner(int yy) {
        y = yy;
}
```

The class outer has two fields that are instances of the class inner. The new material between outer::outer(int xx) and the { are respectively parameter values for the constructors that initialize the instances anotherinner and aninner of the class inner. It is suggested that the reader add to the above example a function that declares an instance of outer and compile the example. This can most simple be done by executing something like:

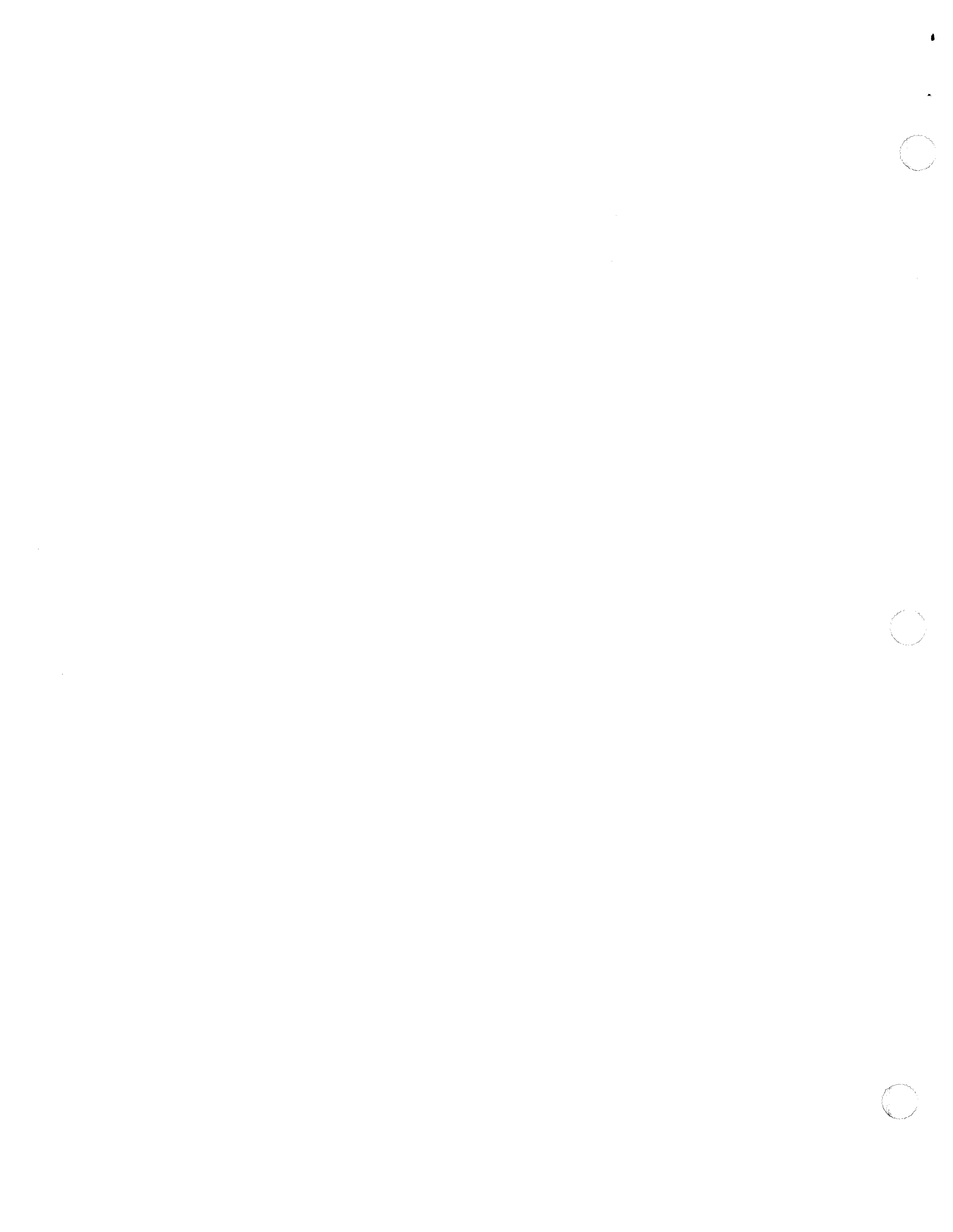
```
cfront < input.c > output
```

By now, the reader should be able to make sense of the generated C code. This exercise will show that the constructor for outer actually calls the constructors for the two instances of inner before proceeding with its own body. If destructors had been defined for the classes in the example, they would be called, on exit, in just the opposite order from that in which the constructors were called, that is, the destructor for outer would execute its explicit body before calling the destructors for the instances of inner.

Here, now, is an example of the second case, a hierarchy of classes where the base class has a constructor.

```
class base {
        int x;
public:
        base(int);
};

class derived : public base {
        int y;
public:
        derived(int);
};

base::base(int xx) { x  = xx; }

derived::derived(int yy) : (yy-1) {y = yy * 2;}
```

Look now at the definition of the constructor for derived. The item (yy-1), a parameter list not preceded by a variable name, is the parameter value to be given to the constructor for the base. Experiment shows that the compiler also accepts the form base(yy-1), i.e., the parameter list preceded by the name of the base class. One point not mentioned in The Book: it appears that in a multi-level class hierarchy, if the immediate parent class does not have a constructor, the parameter applies to the nearest ancestor that does have one. Finally, a word about the order of events. If a constructor for a derived class has parameters both for a base class and for its own members, then the base constructor is called first, followed by the member constructors in the order of their parameter lists, finally followed by the body of the constructor for the derived class. I suggest reading of section 4 of *The Evolution of C++: 1985 to 1987* by Bjarne Stroustrup for a more complete description of the order of execution of constructors.



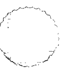
The extension to multiple inheritance is straightforward. The parameters must be preceded by the class names in order to identify them. Experiment shows that even if only one of the multiple base classes has a constructor, the parameter must nevertheless be named. Here is an example:

```
class base1 {
      int x;
public:
base1(int);
};

class base2 {
      int y;
      int z;
public:
base2(int, int);
};

class derived : public base1, public base2 {
      int w;
public:
      derived(int);
};

base1::base1(int xx) {x = xx;}

base2::base2(int yy, int zz)
{
      y = yy;
      z   = zz;
}

derived::derived(int ww) : base1(ww+1), base2(ww-1, ww * 3)
{
      w = ww;
}

void f()
{
      derived a(2);
}
```

Here, the declaration `derived a(2);` causes the constructor for `derived` to be called with parameter value 2. Before the body of the constructor can be executed, there are calls respectively to the constructors for `base1`, with parameter value 3, and `base2`, with parameter values 1, 6.

4. Homemade storage management

In The Book, section 5.5.6, there is discussion of a technique for improving performance when many small objects of the same class are to be allocated. The example used is a simplified version of some code that is actually to be found in the C++ compiler. Because Bjarne considers the technique to be a "hack," and will in time disallow it in favor of a new technique that has become available in Release 2.0, namely redefining of the `new` and `delete` operators, I feel it is instructive to further simplify his example, and use it as a vehicle for comparing the techniques.

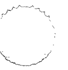


Consider the following complete program. The purpose of the function `main()` is to exercise the memory manager by numerous requests for creation and deletion of instances of the class `name`, in fact, to exercise it sufficiently so that execution can be timed.

```
struct name {
      char* string;
      name* next;  // This field will be used later,
      name(char*); // that is, in the next example.

};


name::name(char* s)
{
      string = s;
}


main()
{
      const COUNT = 10000;
      const ITER = 100;
      name* nameptr[ITER]; // array of pointers to name

      for (int j = 0; j < COUNT; j++) {

            for (int i = 0; i < ITER; i++)
                  nameptr[i] = new name("Herb");
      /* constructor assigns "Herb" to field "string" */

            for (i = 0; i < ITER; i++)
                  delete nameptr[i];
      }
}
```

When executing the above code, each instance of `new name` results in a call to the library function that corresponds to the `new` operator, and this function in turn calls `malloc()`. Similarly, each use of `delete` results in a call to `free`. Wouldn't it be nice if we could instead call the first time for allocation of enough memory for a reasonable number of instances of `name`, link all but one of them together by means of the pointer field `next`, and return to the caller a pointer to the remaining one? Then, subsequent requests for instances could be handled by normal list processing techniques. The operator `delete` would merely link the instance back on the list of available instances.

The next stretch of code shows how the constructor and destructor for `name` can be modified to provide do-it-yourself storage allocation. This is the technique which is used in the C++ compiler, is still supported in Release 2.0, and which will eventually be disallowed. The function `main` is not shown in the next two examples because it suffers no change.



```
struct name {
      char* string;
      name* next;
      name(char*);
      ~name();
};


const NALL = 128;
name* nfree; // the free list header, a global variable

name::name(char* s) // the constructor does its own
                    // storage allocation
{
      name* p = nfree;

      if (p)
            nfree = p->next; // get the next free one
      else { // allocate 128 (NALL) of them
            name* q = (name*)new char[NALL * sizeof(name)];
            for (p=nfree=&q[NALL-1]; q<p; p--)
                  p->next = p-1;
            (p+1)->next = 0;
      }

      this = p; // here is the gimmick, a hack
      string = s;
}

name::~name() // the destructor returns the item to the
              // free list
{
      next = nfree;
      nfree = this;
      this = 0; // also a gimmick, but not a hack
}

/* Definition of main() should follow here */
```

Note now that the constructor in the above code tests `nfree`, which as a global variable is initialized to zero. So, the first time the constructor is called, space for `NALL` instances is allocated, and `NALL-1` of them are chained together, with `nfree` pointing to the start of the chain. The address of the remaining one is assigned to `this`. Now, the hack is that the C++ compiler, which ordinarily generates a conditional call to the "new" function, a call conditioned on `this` being zero, is inhibited from generating this call if there is an assignment to `this` in the code. Therefore, the implicit call of the memory allocator is not made. A similar thing happens in the destructor code, which obviously returns the "deleted" instance to the free list. The assignment of zero to `this` prevents the call of the "delete" function. However, there is no hack here. The normally generated call to "delete," which is done at the point of exit from the destructor, is conditioned on `this` being non-zero.

Finally, we come to the recommended method, which consists of defining private `new` and `delete` operators as members of the class `name`. Overloading of operators is planned to be the subject of the next Technical Note, but this particular case fits exactly into the present discussion. The code is:

```
struct name {
      char* string;
      name* next;
      name(char*);
      void* operator new(long);
      void operator delete(name*);
};


const NALL = 128;
name* nfree;

void* name::operator new(long n)
{
      name* p = nfree;

      if (p)
            nfree = p->next;
      else {
            name* q = (name*)new char[NALL * n];
            for (p=nfree=&q[NALL-1]; q<p; p--)
                  p->next = p-1;
            (p+1)->next = 0;
      }

      return p;
}

void name::operator delete(name* p)
{
      p->next = nfree;
      nfree = p;
}

name::name(char* s)
{
      string = s;
}

/* Definition of main() should follow here */
```

I will endeavor here to explain the redefinition of `new` and `delete` without direct reference to the intermediate C code. If the reader has any difficulty with the explanation, it is suggested that he create a file, say `constr.c`, containing the above example, being sure to include the function `main()` taken from the first example in this section, and execute something like:

```
cfront < constr.c > constr.cc
```

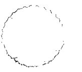
Examination of `constr.cc` should clarify matters.

The operator `new` is a special case in behavior and representation. When an operator is redefined, one has a case of overloading, that is the same operator has more than one meaning. The resolution of the ambiguity is determined by the type of at least one of its operands. To tell the relevant part of the story, if an operand is an instance of a class or an instance of a reference to a class, this is sufficient to identify an overloaded operator which has been redefined as a class member. In the case of `new`, the operand is not a class instance, it is the class **name**. In fact, the relevant instance does not yet exists; the purpose of `new` is to create it. As a member of the class, the redefinition of the operator has the form of a function defini-

tion. In the case of `new` it reads: `void* name::operator new(long n) {...}`. The reason for the return `void*` instead of `name*` is that one might want this definition of `new` to be inherited by a class derived from `name`, in which case returning a pointer to the base class would be an error. So, a generic return pointer is used, and casts will appear where needed in the generated C code. The next anomaly is the formal parameter, `n`. Normally, a formal parameter in this form of an operator definition corresponds to the second argument of a binary operator (the first argument is the implicit `this`, a pointer to the class object). However, here there is only one argument, the class name. One discovers on looking at the C code that the call to the function `new`—the call is found in the code of the constructor—uses as the actual parameter corresponding to `n` the expression `sizeof(struct name)`. This code also applies the cast `(struct name *)` to the value returned by the function. Note that the operator `new` that is used inside the definition of the function `new` is the plain, vanilla, system operator. This is determined by the fact that its argument is a basic type: `char`. There is no infinite recursion!

Nothing much need be said about `delete`. Its argument and the formal parameter of the corresponding function are consistent: pointer to `name`. Note that a destructor is not needed in this example. If a dummy destructor is constructed, to wit: `name::~name(){}`, then if this destructor is written as an in-line function, its presence causes no modification of the C code. If it is an explicitly defined function, then it does nothing but call the redefined `delete` function.

The performance on both the Mac II and Mac Plus of the examples given here may be of interest. In order to make the most realistic possible comparison, the following functions were made in-line: the constructor in the first example, the destructor in the second example, and both the constructor and the `operator delete` function in the third example. On both machines, the two programs in which one does ones own memory management ran at an equal rate. On the Mac Plus, the program using the system memory management ran a factor of eight slower, and on the Mac II, it ran a factor of eleven slower.



C++ Technical Notes–Number 3

H. Kanner

Development Systems Group

Apple Computer Inc.

16 May 1988

1. Introduction

The principal subject matter of this note is the redefining of operators. This cannot be discussed without some mention of the general subject of overloading; it also necessitates some discussion of overloaded functions. To say that an operator (or function) is overloaded is to say that it takes differing actions which are dependent on the types of the arguments (or operands). Overloading, at least for operators, is a common feature of most programming languages that have arithmetic typing. In C, or Pascal, or FORTRAN, the "+" symbol is used to denote either integer or floating-point addition, depending on the types of the things to be added. There is even an instance of a prototype computer having been built in which data items had some bits dedicated to being type flags, and in which the interpretation of a command depended on the type of the operand (J. K. Iliffe, *Basic Machine Principles*, Macdonald Computer Monographs, 1968). Iliffe put it to me that while context switching in virtual memory schemes protected programmers from each other, his Basic Machine was designed to protect programmers from themselves!

The reference materials for this note are the book *The C++ Programming Language* (referred to henceforth as The Book) and the paper *The Evolution of C++: 1985 to 1987*, both by Bjarne Stroustrup. The second of these describes language changes that have been made since the book was written, and which will presumably be implemented by AT&T. In style, this note will depend more heavily on examples from the Stroustrup book than the previous notes, primarily because I feel that some of the examples illustrate the points as well as any that I could make up. I will enlarge upon these examples by showing the result of compiling selected portions of them. As before, I will edit the output of the compiler to enhance readability.

As is usual with this language, attempts to explain matters in a logical sequence invariably trap one in a vicious circle. In this case, I would like to make an early introduction of the selection criteria by which one of a set of overloaded functions or operators is to be chosen. In order to do that, I have to bring in user-defined conversions and that cannot be done completely without describing conversion operators, thus completing the circle because the latter is one of several topics to be discussed in the section on user-defined operators.

2. Simple function overloading

Consider the function that produces the absolute value of its argument. In the standard C libraries, there are two such functions, declared in ANSI C as `int abs(int)`, and `double fabs(double)`. It would be nice if the programmer could use the same function in both cases, just as the same symbol, "+," is used for addition of these respective types. By use of the overloading facility, this indeed can be done in C++. The following declarations and definitions could do the trick:

```
overload abs;       // in a header
int abs(int);
double abs(double);

int abs(int x)      // in a library
{
      return x >= 0 ? x : -x;
}

double abs (double x)
{
      return x >= 0 ? x : -x;
}

main()              // usage demonstrated here
{
      int i;
      double d;

      d = -5.0;
      d = abs(d);

      i = -5;
      i = abs(i);
}
```

This produces C code that when edited for readability would look something like this:

```
int abs__1(x)
int x;
{
      return (x >= 0 ) ? x : (- x );
}

double abs__2(x)
double x;
{
      return (x >= 0 ) ? x : (- x );
}

int main ()
{
      int i;
      double d;

      d = -5.0;
      d = abs__2(d);

      i = -5;
      i = abs__1(i);
}
```

Note that the two incarnations of the overloaded `abs` function are distinguished in the C code by the names `abs__1` and `abs__2`. The actual output of *cfront* produces much more complicated names, which contain enough information to deduce from the name the numbers and respective types of the function's arguments.

In real life, one would choose to define the functions as inline, i.e.:

```
inline int abs(int x)
{
        return x >= 0 ? x : -x;
}

inline double abs (double x)
{
        return x >= 0 ? x : -x;
}
```

producing the following C code for the function `main`:

```
int main ()
{
        int i;
        double d;

        d = -5.0;
        d = (d >= 0) ? d : -d;
        i = -5;
        i = (i >= 0) ? i : -i;
}
```

3. Simple operator overloading

The idea of providing a facility in programming languages for redefining operators so that they can be applied to user-defined types is not a new one. After all, in a language in which representations for matrices, vectors and complex numbers can be defined by the programmer, and in a world in which some of the ordinary arithmetic operators may be applied to these entities, it is a natural desire to want to be able to extend the meaning of said operators because such extensions may yield more readable programs. For example, I define a complex number type as:

```
struct complex {
        double re;
        double im;
};
```

I can then define a function that returns the sum of two complex variables as:

```
complex sum(complex a, complex b)
{
        complex temp;

        temp.re = a.re + b.re;
        temp.im = a.im + b.im;
        return temp;
}
```

and perform the addition by calling the function, as in

```
complex a, b, c;

a = sum(b, c);
```

I can even declare `complex` as a class, and make `sum` a friend of the class. But, wouldn't it be lovely to be able to write `a = b + c;` ?

This can be done by the provided methods for overloading an operator. The notation is simple. For the operator "+," one simply defines, as a member or friend of the class, a function whose name is `operator+`. For the simple example above, one would write:

```
class complex {
      double re, im;
public:
      complex(double r, double i) // the constructor
      {
            re = r; im = i
      }

      friend complex operator+(complex, complex);
};
```

and define the operator by:

```
inline complex operator+(complex a, complex b)
{
      return complex(a.re + b.re, a.im + b.im);
}
```

The use of a constructor in the above definition shortens the source code; the code shown previously as the definition of the function `sum` could have been used instead.

4. General rules

- The language does not provide for the creation of new operators. Only a large subset of the existing operators can be redefined.

- The syntax rules for the use of a redefined operator do not change. A binary operator remains a binary operator. A prefix operator remains a prefix operator. An operator which has both a binary and unary definition, e.g. "–," may be redefined independently in each form. The precedence of an operator cannot be changed. There is no provision in the methodology for redefining operators to indicate whether a unary operator is prefix or postfix. Therefore, a redefinition of the operators "--" and "++" cannot distinguish between prefix and postfix usage.

- The effect of the application of operators to basic types cannot be altered. Therefore a redefined (overloaded) operator must be defined as a member of a class (or structure), or must have at least one argument that is a class (or structure) object or a reference to a class (or structure). Note that I say reference, not pointer. The operators `new` and `delete` are exceptions to this rule; they were discussed in the previous Technical Note. Three other operators: `[]`, `()`, and `->` are exceptions in that they must be class (or structure) members.

- The operators which may not be redefined are

  `::` `sizeof` `&` (as a unary operator) `*` (as a unary operator)

  `()` (as a cast) `?` `:` `.` `,`

- With the exception of `[]`, `()`, `->`, and a category, to be described in Section 5 below, called *conversion* operators, all redefined operators map to functions in the following way:

  A unary operator `@`, used in the context `@x` if it is a prefix operator and the context `x@` if it is postfix, where `x` is an object of the class for which `@` has been redefined, is interpreted as `x.operator@()` if `operator@` has been defined as a class member, and as `operator@(x)` if `operator@` has been defined as a friend of the class. (Remember that `this`, the pointer to the object, is an implicit argument of `operator@` when that function is a class member.)

  A binary operator `@`, used in the context `x@y`, is interpreted as `x.operator@(y)`, if `operator@` has been defined as a class member and `x` is an object of that class. If `operator@` has instead been defined as a friend of the class, then at least one of the arguments `x` and `y` must be an object of that class, and the interpretation is `operator@(x, y)`.

With respect to the above rules, for any argument which must be a class object, the corresponding formal parameter in the operator definition may be either the class itself or a reference to the class. It is time now for a more complete example. What follows is a subset of the material that would normally be in a header file called `complex.h`. In the real header, the constructor and all of the operator definitions are declared as *inline*. Because of the total unintelligibility of the C code produced from inline expansions, I have modified the source so that nothing is inline.

```
class complex {
      double re, im;
public:
      complex (double r = 0.0, double i = 0.0);
      friend double real(const complex);
      friend double imag(const complex);
      friend complex operator+(complex, complex);
      void operator+=(complex);
};

 complex::complex(double r, double i)
{
      re = r;
      im = i;
}

 double real(const complex a)
{
      return a.re;
}

 double imag(const complex a)
{
      return a.im;
}
```

```
 complex operator+(complex x, complex y)
{
      return complex(x.re + y.re, x.im + y.im);
}

 void complex::operator+=(complex x)
{
      re += x.re;
      im += x.im;
}
```

The C produced by the above code, modified for readability, is:

```
struct complex {
      double re;
      double im;
};



struct complex *_complex_constructor(_this, r, i)
struct complex *_this;
double r;
double i;

{
      if (_this == 0 )
            _this = (struct complex *)
                   _new( (long) sizeof (struct complex));
      _this->re = r;
      _this->im = i;
      return _this;
}


double _real(a)
struct complex a;
{
      return a.re;
}

double _imag(a)
struct complex a;
{
      return a.im;
}

struct complex _plus(x, y)
struct complex x;
struct complex y;
{
      struct complex _temp;

      _complex_constructor(& _temp, x.re + y.re,
                                              x.im + y.im);
      return _temp;
}
```

```
void _complex_plus_assign_op(_this, x)
struct complex *_this;
struct complex x;
{
      _this->re += x.re;
      _this->im += x.im;
}
```

Let us now use these definitions in a short program:

```
void f()
{
      complex x, y(3,4), z;
      double d;

      x = complex(1,2);
      d = real(x);
      z = x + y;
      z += x;
}
```

which compiles to:

```
void f()
{
      struct complex x;
      struct complex y;
      struct complex z;
      double d;

      _complex_constructor(&x, (double) 0.0, (double) 0.0);
      _complex_constructor(&y, (double) 3, (double) 4);
      _complex_constructor(&z, (double) 0.0, (double) 0.0);

      {
            struct complex _temp;

            _complex_constructor( &_temp,
                                  (double) 1,(double) 2);
            x = _temp;
      }
      d = _real(x)
      z = _plus(x, y);
      _complex_plus_assign(&z, x);
}
```

Note particularly the treatment of the "+=" operator, which is the one instance of an operator which has been defined as a member function. In the expression `z += x`, the call, in C++ terms, is to the member function `operator+=` for the class object `z`, and the single argument is `x`. In the C expansion, the implicit parameter `this` (pointer to class object) appears as an added first argument, and in the call of the function is given as the address of `z`. A second point about an overloaded assignment operator is that overloaded versions of = and an operator `@` do not have to conform to the equivalence relation that holds for the default operators, to wit: that `a = a @ b` is equivalent to `a @= b`.

5. User-defined conversions

The exact rules for conversions and the rules governing their application are singularly hard to find in The Book. Conversions may be explicit or implicit: implicit conversions are exemplified by those described in Section 6.6 of the Reference Manual in The Book, e.g. the conversion of an `int` to a `double` in an arithmetic expression that has both kinds of operands; explicit conversions are those spelled out by use of a cast. We are concerned here with two aspects of this general subject: the extension of the language by conversions that coerce operands in either direction between basic types and user-defined types, and the use of those and other conversions in deciding which instance of a group of overloaded functions is to be called.

A constructor that takes a single argument can be regarded as a conversion operator, or cast, that converts the type of the argument to the type of the class of which the constructor is a member. For example, the constructor for `complex` given in Section 4 above has defaults for its two arguments, and therefore certainly can be called with only one argument. The constructor called with one argument can be regarded as a conversion operator that converts a `double` to a `complex`. Thus, the same effect is produced by the following three statements:

```
complex z = complex(11, 0);
complex z = complex(11); and
complex z = 11;
```

In the last of the above, the conversion takes place implicitly. Note that for all of the three, a previous implicit conversion first had to take place. The `int 11` had to be converted to the `double 11.0` before the conversion to `complex` could be invoked. This demonstrates that once such a constructor has been defined, mixed expressions containing `int`, `double`, and `complex` can be written without the need for any casts.

The reverse conversion, that from a user-defined type to a basic type, is performed by a new animal, one called a *conversion operator*. The syntax for it is that of a unary operator that is a member function of the class, where the symbol representing the operator is the name of the basic type, delimited on the left by at least one space. Thus, a class member defining a conversion to `int` would be declared `operator int()`. This is well illustrated by the class "tiny," to be found in Section 6.3.2 of the book. A slightly modified version is reproduced below with the inline function definitions rewritten, and a small program using it is shown. The purpose of the class is to describe objects that are integers in the range 0 to 63, and to provide range checking after arithmetic operations wherever necessary.

```
class tiny {
      char v;
      int assign(int);
public:
      tiny(int);    // constructor: convert an int to a tiny
      tiny(tiny&); // constructor: one tiny
                     // initializes another
      int operator=(tiny&);
      int operator=(int);
      operator int();
};
```

```
tiny::assign(int i)// this private member does range
                    //checking
{
      if (i & ~63) {
            printf("range error\n");
            exit(1);
      }
      else
            return v = i;
}

tiny::tiny(int i) // construct from integer;
{                 // must check range
      assign(i);
}

tiny::tiny(tiny& t)// construct from instance of tiny;
{                  // range check unnecessary
      v = t.v;
}

int tiny::operator=(tiny& t)// assign tiny to tiny;
{                           // range check unnecessary
      return v = t.v;
}

int tiny::operator=(int i)// assign int to tiny;
{                         // must check range
      return assign(i);
}

tiny::operator int()
{
      return v;
}
```

Here is a cleaned-up version of the C code produced from the above.

```
extern int printf ();
extern int exit ();

struct tiny {
char v;
};

int _tiny_assign(_this , i)
struct tiny *_this;
int i;
{
      if (i & -64){
            print("range error\n") ;
            exit(1);
}
else
      return (int ) _this->v = i);
}
```

```
struct tiny *_tiny_constructor_1(_this, i)
struct tiny *_this;
int i;
{
        if (_this == 0)
             _this = (struct tiny *) _new( (long) sizeof
                                                    (struct
       _tiny_assign(_this, i);
       return _this;
}


struct tiny *_tiny_constructor_2(_this, t)
struct tiny *_this;
struct tiny *_t;
{
       if (_this == 0)
       _this = (struct tiny *) _new( (long) (sizeof
                                      (struct tiny));
       _this->v = (*t).v;
       return _this;
}


int _tiny_as_1(_this, t)
struct tiny *__this;
struct tiny *_t ;
{
       return (int )_this->v = (*t).v;
}

int _tiny_as_2(_this, i)
struct tiny *_this;
int i;
{
       return _tiny_assign(_this, i);
}

int _tiny_int(_this)
struct tiny *_this ;
{
       return (int) _this->v;
}
```

In looking at the above code, and the C++ source from which it was produced, I was briefly puzzled as to why the assignment operator was defined as returning an `int`, particularly because the very next example in The Book defines an overloaded assignment operator as returning `void`. The reason undoubtedly is to permit nesting, i.e. to provide for multiple assignment statements.

The reader should study the examples of the application of `tiny` given in Section 6.3.2. I will analyze only one of these in order to illustrate the significance of the conversion operator `operator int`. The example is the declaration: "`tiny c3 = c2 - c1;`", where `c2` and `c1` are objects of class `tiny`. Compilation of this declaration produces:

```
_tiny_constructor_1(& c3, _tiny_int(&c2)
                    - _tiny_int(&c1));
```

Let us analyze this example. The constructor _tiny_constructor_1 is the one which takes an integer argument. The presence of a minus sign on the right hand side of the quoted C++ declaration causes implicit call of the operator that converts an instance of tiny to an int. The difference between the two integers so obtained becomes the argument given to the constructor. Thus, by implicitly converting to int when faced with arithmetic operators wanting int operands, it becomes unnecessary to redefine the arithmetic operators for tiny.

The rules for choosing which one of a set of overloaded operators or functions is to be called can now be stated (I quote from Section 8.9 of the Reference Manual):

In the order stated—

- Look for an exact match and use it if found.
- Look for a match using standard conversion and use any one found.
- Look for a match using user-defined conversions. If a unique set of conversions is found use it.

We are are talking here about a match between the types of the actual arguments and the types used in the operator or function declaration. The terms used in the above rules require some explanation. The Book states that a zero, a char, or a short are to be considered as exact matches to a formal parameter of type int, that a float is similarly regarded as an exact match to a requested double, and that the only standard conversions to be used are int to long, int to double, and the pointer and reference conversions given in Sections 6.7 and 6.8 of the Reference Manual. Section 6.3.3 of the main portion of The Book contains a discussion of the rationale for the above rules and a number of examples. It is emphasized that only one level of user-defined conversion is accepted, so an instance of an overloaded function/operator cannot be chosen via a chain of user-defined conversions.

6. Assignment and initialization

We have already seen instances of an overloaded assignment operator and initialization by overloaded constructor in the class tiny, where the type of the parameter determined whether range checking was required. Here we look at a more subtle requirement for these functions. This is illustrated quite capably in Section 6.6 of The Book. Basically, the scenario is one of a class which contains a field that is a pointer and a constructor which dynamically allocates the space to which that pointer refers. The class definition, as described to this point, is:

```
struct string {
      char* p;
      int size;
      string(int);
      ~string();
};

string::string(int sz)
{
      p = new char[size = sz];
}

string::~string()
{
      delete p;
}
```

It is shown that with merely the above definition, neither assignment nor initialization will work properly with respect to construction and destruction. The rule that you only destroy that which has been constructed, and only do it once, is violated. The cure is to add to the class the members:

```
void string::operator=(string& a) // assignment operator
{
      if (this == &a)
            return;
      delete p;
      p  = new char[size = a.size];
      strcpy(p, a.p);
}
```

and

```
string::string(string& a)  // constructor taking
{                          // reference to string as argument
      p = new char[size = a.size];
      strcpy(p, a.p);
}
```

The assignment operator would be implicitly called upon assignment of one instance of `string` to another, both having been previously declared. The constructor would be implicitly called upon initialization of a freshly-declared string by a previously-declared and initialized one. The reader should extend the class `string` as I have just indicated, and look at the C code produced by compiling it together with the sample program:

```
void f()
{
      string s1(10);
      string s2(20);
      string s3 = s1; // this will call string(string&)
      s2 = s1;        // this will call operator=(string&)
}
```

A more difficult case to follow, which I will now demonstrate, involves parameter passing and function return. A persistent refrain throughout The Book is that the semantics of parameter passing is identical to that of initialization. Since this statement is significant only when initialization has different semantics from that of simple assignment, it follows that it is significant only when the formal parameter of an initialization function is a reference. In Section 6.6, it is stated that function return also has the semantics of initialization. This means that if a constructor of the form `string(string&)` has been defined, this constructor will be implicitly called when passing a parameter of type `string` or returning a value of that type. This is demonstrated with the program:

```
string ss(10);

int g(string str){ return sizeof str;}

string h(){ return ss; }

main()
{
      int i;
      string s(5);

      i = g(s);
      s = h();
}
```

which, when considerably massaged after compilation, would look something like this:

```
struct string ss ;


void g(str)
struct string *str;
{
      return sizeof *str;
}

void h(_result)
struct string *_result;
{
      /*
         The call below is to the constructor that
         takes a reference to a string as an argument.
         The function h is passed an address, "_result,"
         and, by calling the constructor, causes allocation
         and copying so that the string whose address is in
         _result becomes a true copy of the string ss.
      */

      _string_constructor_2(_result,
            (struct string *) &ss);
}


int main ()
{
      int i;
      struct string s;

      /*
            the call below is to the constructor that
            takes an integer argument
      */
      _string_constructor_1(&s, 5);
      {
            struct string _templ;

            /*
               The constructor, which is the argument
               of g, causes the string _templ to be a
               true copy of s, and returns the address
               of _templ.
            */

            i = g(_string_constructor_2(&_templ,
                              (struct string *)&s));
```

```
                    {
                            struct string _temp2;

                            /*
                              The call of h causes _temp2 to become a
                              copy of ss.  Then the call of the string
                              assignment operator makes the assignment
                              from _temp2 to s.
                            */

                            h(&_temp2);
                            _string_as(&s, (struct string *) &_temp2);

                            /*
                               Finally, destructors are called to get
                               rid of _temp1, _temp2, and s.
                            */

                            _string_destructor(&_temp2, (int) 0);
                            _string_destructor(&_temp1, (int) 0);
                            _string_destructor(&s, (int) 0);
                    }
            }
    }
```

7. The special cases

The Book describes overloading of the subscription operator `[]` and the function call operator `()`. The paper cited in the introduction to this note describes the proposed overloading of the operator `->`. All three of these must be defined as member functions of that class for which an instance thereof is to be their left-hand operand.

This section will be mercifully short. The Book gives adequate examples of the application of overloading for the first two of the three operators. Only a brief synopsis of them will be presented here.

Two kinds of application of the subscription operator are presented. The first is simply to provide range checking. That is, if a class is defined such that a class member is an array, with other class members giving the upper and lower bounds of legal subscripts, then a subscription operator can be defined that checks the bounds, aborts with an error message if a given subscript is out of range, and otherwise does a normal subscription relative to the lower bound. The syntax is straightforward. If in a class called `alpha` the subscription operator is declared as a member, e.g. by the declaration `int& operator[](int);`, then this overloaded instance of the operator is implicitly invoked by `alphavar[5]`, where `alphavar` is an instance of `alpha`. The only interesting thing about this is that the operator has been declared as returning a **reference** to an `int`. This is to permit the subscripted variable to appear on the left side of an assignment. The other example, and a very interesting one it is, is given in Section 6.7 of The Book. It shows how an associative array can be set up and addressed. Each array element consists of a string/integer pair. Given that this array is a class member and that the subscription operator is declared for the class, then a call might look like `vec["Herb"]`, where `vec` is an instance of the given class. The operator returns a reference to the associated integer if the string is found in the array. If the string is not found, the array is extended with a new element containing the string and an initialized value (zero) for the integer. In The Book, this is used to produce a count of the number of appearances of each of a set of strings in a buffer. Each time a given string is found, the count is bumped; again, this can be done because the return value is a reference. The application of associative arrays to symbol table maintenance is obvious.

The function call operator, `()`, does not add any new capability to the language; it is purely cosmetic in nature. I will illustrate its use by taking the example of the *iterator*, Section 6.8 of The Book, and rewriting the declaration so that the call is to a member function instead of using an overloaded function call operator. Then I will show the alternate way of writing the same declaration and call, using an overloaded `()` operator. All that needs to be recalled from the previous paragraph is that the array of string/integer pairs is the private data of a class, to be called `assoc`. The object of the iterator is to return a pointer to such a data pair, and to bump an index so that the next time it is called it produces a pointer to the next pair. The iterator is a class, called `assoc_iterator`. Its private data are an index and a pointer to an object of `assoc`. The declaration of `assoc` must declare `assoc_iterator` as a friend. Now here is what this class might look like:

```
class assoc_iterator {

      /* private data */

public:
      assoc_iterator(assoc&); // constructor
      pair* iterate(); // this one does the real work
                       // "pair" is the type of the
                       // associative array members
};
```

and the code fragment that calls the iterator would look like:

```
...
assoc vec(512) // construct vec, an object of assoc
...
assoc_iterator next(vec) // construct next, an object
                         // of assoc_iterator
pair* p;
p = next.iterate(); //call the iterate function
```

Since the only member function of `assoc_iterator`, aside from its constructor, is `iterate`, the use of the function name might be regarded as a bit redundant. The ability to overload the `()` operator allows us to eliminate this name. The declaration changes to:

```
class assoc_iterator {

      /* private data */

public:
      assoc_iterator(assoc&); // constructor
      pair* operator()(); // this one does the real work
                       // "pair" is the type of the
                       // associative array members
};
```

and the call changes to:

```
...
assoc vec(512) // construct vec, an object of assoc
...
assoc_iterator next(vec) // construct next, an object
                         // of assoc_iterator
pair* p;
p = next(); //call the iterate function
```

The last operator to be considered in this group of special cases is `->`. Having been described in *The Evolution of C++: 1985 to 1987*, I anticipate that it will appear in the same release of the C++ compiler that implements other features, such as multiple inheritance, that are described in the paper. It differs from the other cases of operator overloading in that activation of the function corresponding to -> starts a two-step process. For the expression `x->m`, where `x` is an object of a class having as a member the function `operator->()`, the function is first called. It must return a pointer to an object that has a member named `m`. Call this pointer `p`. The original expression `x->m` is now replaced by `p->m`. This definition has the following implications:

> Given a class `X` in which `P* operator->()` is defined as a member, and given `x` as an instance of or reference to `X`, then for `x->m` to be valid, the function `operator->()` must return `p`, where the type of `p` is `P*`, where `P` has a field named `m`, where `p` is in the scope of the function `operator->()`, and where `p->m` is in scope at the place where `x->m` appears.

The example below illustrates the simplest application of a redefined ->; it can be thought of as a vanilla -> with arbitrary side effects.

```
struct test{
        int i;
        test* operator->();
};

test* test::operator->()
{
        printf("hi\n");
        return this;
}

main()
{
        test* ptest = new test;
        int ii;
        test& testx = *ptest;

        testx->i = 5;
        ii = testx->i;
        printf("%d\n", ii);
}
```

This, when executed, should print

```
hi
hi
5
```

Each "hi" is a side effect of the overloaded operator.

For a second example, I modify slightly the example in the cited paper in order to demonstrate the generality of the overloaded operator.

```
struct Y {
      int m;
};

Y* p; // The modification is that I have made p a global.
      // In the paper, it is a member of class X.

class X {
public:
      Y* operator->();
};

Y* X::operator->()
{
      printf("hi\n");
      return p;
}

void f(X x)
{
      printf("%d\n", x->m);
}

main()
{
      Y yy;
      X xx;

      p = &yy;
      yy.m = 5;
      f(xx);
}
```

This, when executed, should print

```
hi
5
```

I want to conclude this section by clarifying a point in the paper which I found confusing. After the example that is much like the one above, the paper continues:

```
void f(X x, X& xr, X* xp)
{
      x->m;         // x.p->m
      xr->m;        // xr.p->m
      xp->m;        // error: X does not have a member m
}
```

The comment on the erroneous line is correct, but fails to make the important point, which is that the overloaded operator is not invoked in attempting to evaluate `xp->m`. This is because the left-hand operand of `->` is not the class `X`; it is a pointer variable, and as such does not have any members, much less the member function `operator->()`. Therefore, `xp->m` becomes `(*xp).m`, for which the error comment is true.