

9 Static Construction Self-Organization

9.1 Initialization

Class descriptions are long-lived objects. They are constant and they exist practically as long as an application executes. If possible, such objects are initialized at compile time. However, we have decided in chapter 6 that static initialization makes class descriptions hard to maintain: the order of the structure components must agree with all the initializations, and inheritance would force us to reveal dynamically linked methods outside their implementation files.

For bootstrapping we initialize only the class descriptions **Object** and **Class** at compile time as static structures in the file *Object.dc*. All other class descriptions are generated dynamically and the metaclass constructors beginning with **Class_ctor()** take care of inheritance and overwriting dynamically linked methods.

ooc generates initialization functions to hide the details of calling **new()** to generate class descriptions, but the fact that they must be explicitly called in the application code is a source of hard to diagnose errors. As an example, consider **initPoint()** and **initCircle()** from section 6.10:

```
void initPoint (void) {
    if (! PointClass)
        PointClass = new(Class, "PointClass",
                        Class, sizeof(struct PointClass),
                        ctor, PointClass_ctor,
                        0);
    if (! Point)
        Point = new(PointClass, "Point",
                    Object, sizeof(struct Point),
                    ctor, Point_ctor,
                    draw, Point_draw,
                    0);
}
```

The function is designed to do its work only once, i.e., even if it is called repeatedly it will generate a single instance of each class description.

```
void initCircle (void) {
    if (! Circle)
    {
        initPoint();
        Circle = new(PointClass, "Circle",
                    Point, sizeof(struct Circle),
                    ctor, Circle_ctor,
                    draw, Circle_draw,
                    0);
    }
}
```

Both functions implicitly observe the class hierarchy: **initPoint()** makes sure that **PointClass** exists before it uses it to generate the description **Point**; the call to **initPoint()** in **initCircle()** guarantees that the superclass description **Point** and its metaclass description **PointClass** exist before we use them to generate the description **Circle**. There is no danger of recursion: **initCircle()** calls **initPoint()** because **Point** is the superclass of **Circle** but **initPoint()** will not refer to **initCircle()** because *ooc* does not permit cycles in the superclass relationship.

Things go horribly wrong, however, if we ever forget to initialize a class description before we use it. Therefore, in this chapter we look at mechanisms which automatically prevent this problem.

9.2 Initializer Lists — *munch*

Class descriptions essentially are static objects. They ought to exist as long as the main program is active. This is normally accomplished by creating such objects as global or **static** variables and initializing them at compile time.

Our problem is that we need to call **Class_ctor()** and the other metaclass constructors to hide the details of inheritance when initializing a class description. Function calls, however, can only happen at execution time.

The problem is known as *static constructor calls* — objects with a lifetime equal to the main program must be constructed as soon as **main()** is executed. There is no difference between generating static and dynamic objects. **initPoint()** and similar functions simplify the calling conventions and permit calls in any order, but the actual work is in either case done by **new()** and the constructors.

At first glance, the solution should be quite trivial. If we assume that every class description linked into a program is really used we need to call every **init**-function at the beginning of **main()**. Unfortunately, however, this is not just a source text processing problem. *ooc* cannot help here because it does not know — intentionally — how classes are put together for a program. Checking the source code does not help because the linker might fetch classes from libraries.

Modern linkers such as GNU *ld* permit a compiler to compose an array of addresses where each object module can contribute elements as it is linked into a program. In our case we could collect the addresses of all **init**-functions in such an array and modify **main()** to call each function in turn. However, this feature is only available to compiler makers, not to compiler users.

Nevertheless, we should take the hint. We define an array **initializers[]** and arrange things in **main()** as follows:

```
void (* initializers [])(void) = {
0 };

int main ()
{
    extern void (* initializers [])(void);
    void (** init)(void) = initializers;

    while (* init)
        (** init ++)();
    ...
}
```

All that remains is to specify every initialization function of our program as an element of **initializers[]**. If there is a utility like *nm* which can print the symbol table of a linked program we can use the following approach to generate the array automatically:

```
$ cc -o task object... libooc.a
$ nm -p task | munch > initializers.c
$ cc -o task object... initializers.c libooc.a
```

We assume that *libooc.a* is a library with a module *initializers.o* which defines the array **initializers[]** as shown above containing only the trailing null pointer. The library module is only used by the linker if the array has not been defined in a module preceding *libooc.a* on the command line invoking the compiler.

nm prints the symbol table of the *task* resulting from the first compilation. *munch* is a small program generating a new module *initializers.c* which references all **init**-functions in *task*. In the second compilation the linker uses this module rather than the default module from *libooc.a* to define the appropriate **initializers[]** for *task*.

Rather than an array, *munch* could generate a function calling all initialization functions. However, as we shall see in chapter 12, specifically a list of classes can be put to other uses than just initialization.

The output from *nm* generally depends on the brand of UNIX used. Luckily, the option **-p** instructs Berkeley-*nm* to print in symbol table order and System-V-*nm* to produce a terse output format which happens to look almost like the output from Berkeley-*nm*. Here is *munch* for both, implemented using *awk*:

```
NF != 3 || $2 != "T" || $1 !~ /^[0-9a-fA-F]+$/ {
    next
}
$3 ~ /^_?init[A-Z][A-Za-z]+$/ {
    sub(/^_/, "", $3)
    names[$3] = 1
}
END {
    for (n in names)
        printf "extern void %s (void);\n", n

    print "\nvoid (* initializers [])(void) = {"
    for (n in names)
        printf "\t%s,\n", n
    print "0 };"
}
```

The first condition quickly rejects all symbol table entries except for those such as

```
00003ea8 T _initPoint
```

Assuming that a name beginning with **init** and a capital letter followed only by letters refers to an initialization function, an optional initial underscore is stripped (some compilers produce it, others do not) and the rest is saved as index of an array **names[]**. Once all names have been found, *munch* generates function declarations and defines **initializers[]**.

The array **names[]** is used because each name must be emitted twice. Names are stored as indices rather than element values to avoid duplication.* *munch* can even be used to generate the default module for the library:

```
$ munch < /dev/null > initializers.c
```

munch is a kludge in many ways: it takes two runs of the linker to bind a task correctly; it requires a symbol table dump like *nm* and it assumes a reasonable output format; and, worst of all, it relies on a pattern to select the initialization functions. However, *munch* is usually very easy to port and the selection pattern can be adapted to a variety of static constructor problems. Not surprisingly, the AT&T C++ system has been implemented for some hosts with a (complicated) variant of *munch*.

9.3 Functions for Objects

munch is a reasonably portable, if inefficient, solution for all conceivable static initialization problems. Building class descriptions before they are used is a simple case and it turns out that there is a much easier and completely portable way to accomplish that.

Our problem is that we use a pointer variable to refer to an object but we need a function call to create the object if it does not yet exist. This leads to something like the following macro definition:

```
#define Point (Point ? Point : (Point = initPoint()))
```

The macro **Point** checks if the class description **Point** is already initialized. If not, it calls **initPoint()** to generate the class description. Unfortunately, if we define **Point** as a macro without parameters, we can no longer use the same name for the structure tag for objects and for the corresponding class description. The following macro is better:

```
#define Class(x) (x ? x : (x = init ## x ()))
```

Now we specify **Class(Point)** to reference the class description. **initPoint()** still calls **new()** as before but it now has to return the generated class description, i.e., each class description needs its own initialization function:

```
const void * Point;

const void * initPoint (void) {
    return new(Class(PointClass),
               "Point", Class(Object), sizeof(struct Point),
               ctor, Point_ctor,
               draw, Point_draw,
               (void *) 0);
}
```

This design still observes the ordering imposed by the class hierarchy: before the class description **PointClass** is passed to **new()**, the macro expansion

* Duplication should be impossible to begin with, because a global function cannot be defined twice in one program, but it is always better to be safe rather than sorry.

Class(PointClass) makes sure the description exists. The example shows that for uniformity we will have to supply empty functions **initObject()** and **initClass()**.

If every initialization function returns the initialized object, we can do without macros and simply call the initialization function whenever we want to access the object — a static object is represented by its initialization function:

```
static const void * _Point;

const void * const Point (void) {
    return _Point ? _Point :
        (_Point = new(PointClass(),
            "Point", Object(), sizeof(struct Point),
            ctor, Point_ctor,
            draw, Point_draw,
            (void *) 0));
}
```

We could move the definition of the actual pointer **_Point** into the function; however, the global definition is necessary if we still want to implement *munch* for System V.

Replacing static objects by functions need not be less efficient than using macros. ANSI-C does not permit the declaration of a **const** or **volatile** result for a function, i.e., the boldfaced **const** qualifier in the example.* GNU-C allows such a declaration and uses it during optimization. If a function has a **const** result its value must depend only on its arguments and the call must not produce side effects. The compiler tries to minimize the number of calls to such a function and reuses the results.

9.4 Implementation

If we choose to replace a class description name such as **Point** by a call to the initialization function **Point()** to generate the class descriptions automatically, we have to modify each use of a class description and we need to tweak the *ooc* reports to generate slightly different files.

Class description names are used in calls to **new()**, **cast()**, **isA()**, **isOf()**, and in superclass selector calls. Using functions in place of pointer variables is a new convention, i.e., we will have to modify the application programs and the implementation files. A good ANSI-C compiler (or the **-pedantic** option of GNU-C) can be quite helpful: it should flag all attempts to pass a function name to a **void *** parameter, i.e., it should flag all those points in our C code where we have missed adding an empty argument list to a class name.

Changing the reports is a bit more difficult. It helps to look in the generated files for references to class descriptions. The representation file *Point.r* remains unchanged. The interface file *Point.h* declares the class and metaclass description pointers. It is changed from

* The first **const** indicates that the result of the function points to a constant value. Only the second **const** indicates that the pointer value itself is constant.

```
extern const void * Point;
extern const void * PointClass;
```

to

```
extern const void * const Point (void);
extern const void * const PointClass (void);
```

where the boldfaced **const** can only be used with GNU-C. It helps to have a portable report so we change the relevant lines in *h.rep* as follows

```
extern const void * `%const `class (void); `n `n
extern const void * `%const `meta (void); `n `n
```

and we add a new report to the common file *header.rep*:

```
% const          // GNUC supports const functions
`{if `GNUC 1 const `}
```

ooc normally defines the symbol **GNUC** with value zero but by specifying

```
$ ooc -DGNUC=1 ...
```

we can set this symbol to 1 on the command line and generate better code.

The implementation file *Point.c* contains many changes. All calls to **cast()** are changed; for the most part they are produced by the **%casts** request to *ooc* and thus by the **casts** and **checks** reports shown in section 8.4. Other calls to **cast()** are used in some selectors and superclass selectors and in the metaclass constructors, but these are generated by reports in *etc.rep*, *c.rep*, and *c-R.rep*. It now pays off that we have used *ooc* to enforce our coding standard — the standard is easy to change in a single place.

The significant change is, of course, the new style of initialization functions. Fortunately, these are also generated in *c.rep* and we derive the new versions by converting **Point()** as shown in the preceding section to report format in *c.rep*. Finally, we produce default functions such as

```
const void * const Object (void) {
    return & _Object;
}
```

by the **init** report in *c-R.rep* so that it can benefit from the **GNUC** conditional for *ooc*. This is a bit touchy because, as stated in section 7.5, the static initialization of **_Object** must be coded in *Object.dc*:

```
extern const struct Class _Object;
extern const struct Class _Class;

%init

static const struct Class _Object = {
    { MAGIC, & _Class },
    "Object", & _Object, sizeof(struct Object),
    Object_ctor, Object_dtor, Object_differ, Object_puto
};
```

extern introduces forward references to the descriptions. **%init** generates the functions which reference the descriptions as shown above. **static**, finally, gives

internal linkage to the initialized descriptions, i.e., they are still hidden inside the implementation file *Object.c*.

As an exception, **_Object** must be the name of the structure itself and not a pointer to it so that **&_Object** can be used to initialize the structure. If we do not introduce a macro such as **Class()**, this makes little difference, but it does complicate *munch* a bit:

```

NF != 3 || $1 !~ /^[0-9a-f]+$ / { next }
$2 ~ /^[bs]$/ { bsd[$3] = 1; next }
$2 == "d" { sysv[$3] = 1; next }
$2 == "T" { T[$3] = 1; next }

END {
    for (name in T)
        if ("_" name in bsd) # eliminate leading _
            names[n++] = substr(name, 2)
        else if ("_" name in sysv)
            names[n++] = name

    for (i = 0; i < n; ++ i)
        printf "extern const void * %s (void);\n", names[i]

    print "\nconst void * (* classes []) (void) = {"
    for (i = 0; i < n; ++ i)
        printf "\t%s,\n", names[i]
    print "0 };"
}

```

A class name should now occur as a global function and with a leading underscore as a local data item. Berkeley-*nm* flags initialized local data with **s** and uninitialized data with **b**, System-V-*nm* uses **d** in both cases. We simply collect all interesting symbols in three arrays and match them in the **END** clause to produce the array **names[]** which we actually need. There is even an advantage to this architecture: we can insert a simple shellsort [K&R88] to produce the class names in alphabetical order:

```

for (gap = int(n/2); gap > 0; gap = int(gap/2))
    for (i = gap; i < n; ++ i)
        for (j = i-gap; j >= 0 && \
            names[j] > names[j+gap]; j -= gap)
        {
            name = names[j]
            names[j] = names[j+gap]
            names[j+gap] = name
        }

```

If we use function calls in place of class names we do not need *munch*; however, a list of the classes in a program may come in handy for some other purpose.

9.5 Summary

Static objects such as class descriptions would normally be initialized at compile time. If we need constructor calls, we wrap them into functions without parameters and make sure that these functions are called early enough and in the proper

order. In order to avoid trivial but hard to diagnose errors, we should provide a mechanism which performs these function calls automatically — our programs should be self-organizing.

One solution is to use a linking technique, for example with the aid of a program such as *munch*, to produce an array with the addresses of all initialization functions and call each array element at the beginning of a main program. A function **main()** with a loop executing the array can be part of our project library, and each program starts with a function **mainprog()** which is called by **main()**.

Another solution is to let an initialization function return the initialized object. If the function is locked so that it does the actual work only once we can replace each reference to a static object by a call to its initialization function. Alternatively, we can use macros to produce the same effect more efficiently. Either way we can no longer take the address of a reference to a static object, but because the reference itself is a pointer value, this should hardly be necessary.

9.6 Exercises

The **Class()** macro is a more efficient, portable solution for automatic initialization of class descriptions than using functions. It is implemented by changing reports, class definitions, and application programs just as described above.

munch may have to be ported to a new system. If it is used together with the **Class()** macro, for a production system we can remove the conditional from the macro and initialize all class descriptions with *munch*. How do we initialize things in the right order? Can *ooc* be used to help here (consult the manual in appendix C about option **-M** for *occ*)? What about **cast()** in a production system?

All class descriptions should first show up in calls to **cast()**. We can define a fake class

```
typedef const void * (* initializer) (void);
% Class ClassInit: Object {
    initializer init;
%}
```

and use statically initialized instances as “uninitialized” class descriptions:

```
static struct ClassInit _Point = {
    { MAGIC, 0 }, /* Object without class description */
    initPoint    /* initialization function */
};

const void * Point = &_Point;
```

cast() can now discover a class description with a null class description pointer, assume that it is a **struct ClassInit**, and call the initialization function. While this solution reduces the number of unnecessary function calls, how does it influence the use of **cast()**?

10 Delegates Callback Functions

10.1 Callbacks

An object points to its class description. The class description points to all dynamically linked methods applicable to the object. It looks as though we should be able to ask an object if it can respond to a particular method. In a way this is a safeguard measure: given a dubious object we can check at run time if we are really allowed to apply a specific method to it. If we do not check, the method's selector will certainly check and crash our program if the object cannot respond, i.e., if the object's class description does not contain the method.

Why would we really want to know? We are out of luck if a method must be applied unconditionally to an object which does not know about it; therefore, there is no need to check. However, if it makes no difference to our own algorithm whether or not the method is applied, being able to ask makes for a more forgiving interface.

The situation arises in the context of *callback functions*. For example, if we are managing a window on a display, some inhabitants of the window might want to be informed when they are about to be covered up, displayed again, changed in size, or destroyed. We can inform our client by calling a function on which we both have agreed: either the client has given us the name of a function to be called for a particular event, or we have agreed on a specific function name.

Registering a callback function, the first technique, looks like the more flexible approach. A client registers functions only for those events which are important from its point of view. Different clients may use different sets of callback functions, and there is no need to observe a common name space. ANSI-C actually uses some callback functions: **bsearch()** and **qsort()** receive the comparison function relative to which they search and sort and **atexit()** registers functions to be called just before a program terminates.

Agreeing on specific function names looks even easier: a recognizer generated by *lex* will call a function **yywrap()** at the end of an input file and it will continue processing if this function does not return zero. Of course, this is impractical if we need more than one such function in a program. If **bsearch()** assumed its comparison function to be called **cmp**, it would be much less flexible.

10.2 Abstract Base Classes

Once we look at dynamically linked methods, agreeing on specific method names for callback purposes does not seem to be as limiting. A method is called for a particular object, i.e., which code is executed for a callback depends on an object in addition to a specific method name.

Methods, however, can only be declared for a class. If we want to communicate with a client in the style of a callback function, we have to postulate an *abstract base class* with the necessary communication methods and the client object must belong to a subclass to implement these methods. For example:

```
% OrderedClass: Class   OrderedArray: Object {
%-
    int cmp (const _self, int a, int b);
    void swap (_self, int a, int b);
%}
```

A sorting algorithm can use **cmp()** to check on two array elements by index, and it can use **swap()** to rearrange them if they are out of order. The sorting algorithm can be applied to any subclass of **OrderedArray** which implements these methods. **OrderedArray** itself is called an abstract base class because it serves only to declare the methods; this class should have no objects if the methods are not defined.

Abstract base classes are quite elegant to encapsulate calling conventions. For example, in an operating system there could be an abstract base class for a certain variety of device drivers. The operating system communicates with each driver using the methods of the base class and each driver is expected to implement all of these methods to communicate with the actual device.

The catch is that all methods of an abstract base class must be implemented for the client because they will be called. For a device driver this is perhaps obvious, but a device driver is not exactly a representative scenario for callback functions. A window is much more typical: some clients have to worry about exposures and others could not care less — why should they all have to implement all methods?

An abstract base class restricts the architecture of a class hierarchy. Without multiple inheritance a client must belong to a particular part of the class tree headed by the abstract base class, regardless of its actual role within an application. As an example, consider a client of a window managing a list of graphical objects. The elegant solution is to let the client belong to a subclass of **List** but the implementation of a window forces the client to be something like a **WindowHandler**. As we discussed in section 4.9 we can make an aggregate and let the client contain a **List** object, but then our class hierarchy evolves according to the dictate of the system rather than according to the needs of our application problems.

Finally, an abstract base class defining callback functions tends to define no private data components for its objects, i.e., the class declares but does not define methods and the objects have no private state. While this is not ruled out by the concept of a class it is certainly not typical and it does suggest that the abstract base class is really just a collection of functions rather than of objects and methods.

10.3 Delegates

Having made a case against abstract base classes we need to look for a better idea. It takes two to callback: the client object wants to be called and the host does the calling. Clearly, the client object must identify itself to the host, if it wants the host to send it a message, but this is all that is required if the host can ask the client what callbacks it is willing to accept, i.e., what methods it can respond to.

It is significant that our viewpoint has shifted: an object is now part of the callback scenario. We call such an object a *delegate*. As soon as a delegate announces itself to the host, the host checks what callbacks the delegate can handle and later the host makes precisely those calls which the delegate expects.

As an example we implement a simple framework for a text filter, i.e., a program which reads lines from standard input or from files specified as arguments, manipulates them, and writes the results to standard output. As one application we look at a program to count lines and characters in a text file. Here is the main program which can be specified as part of the implementation file *Wc.dc*:

```
int main (int argc, char * argv [])
{
    void * filter = new(Filter(), new(Wc()));

    return mainLoop(filter, argv);
}
```

We create a general object **filter** and give it as a delegate an application-specific **Wc** object to count lines and characters. **filter** receives the arguments of our program and runs the **mainLoop()** with callbacks to the **Wc** object.

```
% WcClass: Class Wc: Object {
    unsigned lines;           // lines in current file
    unsigned allLines;        // lines in previous files
    unsigned chars;           // bytes in current file
    unsigned allChars;        // bytes in previous files
    unsigned files;           // files completed
%-
    int wc (_self, const Object @ filter,          \
            const char * fnm, char * buf);
    int printFile (_self, const Object @ filter,    \
                  const char * fnm);
    int printTotal (_self, const Object @ filter);
%}
```

The methods in **Wc** do nothing but line and character counting and reporting the results. **wc()** is called with a buffer containing one line:

```
% Wc wc {
%casts
    ++ self -> lines;
    self -> chars += strlen(buf);
    return 0;
}
```

Once a single file has been processed, **printFile()** reports the statistics and adds them to the running total:

```

% Wc printFile {          // (self, filter, fnm)
%casts
    if (fnm && strcmp(fnm, "-"))
        printf("%7u %7u %s\n",
                self -> lines, self -> chars, fnm);
    else
        printf("%7u %7u\n", self -> lines, self -> chars);
    self -> allLines += self -> lines, self -> lines = 0;
    self -> allChars += self -> chars, self -> chars = 0;
    ++ self -> files;
    return 0;
}

```

fnm is an argument with the current filename. It can be a null pointer or a minus sign; in this case we do not show a filename in the output.

Finally, **printTotal()** reports the running total if **printFile()** has been called more than once:

```

% Wc printTotal {        // (self, filter)
%casts
    if (self -> files > 1)
        printf("%7u %7u in %u files\n",
                self -> allLines, self -> allChars, self -> files);
    return 0;
}

```

Wc only deals with counting. It does not worry about command line arguments, opening or reading files, etc. Filenames are only used to label the output, they have no further significance.

10.4 An Application Framework — *Filter*

Processing a command line is a general problem common to all filter programs. We have to pick off bundled or separated flags and option values, we must recognize two minus signs — as the end of the option list and a single minus sign — additionally as standard input, and we may need to read standard input or each file argument. Every filter program contains more or less the same code for this purpose, and macros such as **MAIN** [Sch87, chapter 15] or functions such as **getopt(3)** help to maintain standards, but why regurgitate the code in the first place?

The class **Filter** is designed as a uniform implementation of command line processing for all filter programs. It can be called an *application framework* because it establishes the ground rules and basic structure for a large family of applications. The method **mainLoop()** contains command line processing once and for all and uses callback functions to let a client deal with the extracted arguments:

```

% mainLoop {              // (self, argv)
%casts
    self -> progname = * argv ++;

```

```

while (* argv && ** argv == '-')
{
    switch (* ++ * argv) {
        case 0:                // single -
            — * argv;           // ... is a filename
            break;              // ... and ends options
        case '-':
            if (! (* argv)[1]) // two —
            {
                ++ argv;       // ... are ignored
                break;         // ... and end options
            }
        default:                // rest are bundled flags
            do
            {
                if (self -> flag)
                {
                    self -> argv = argv;
                    self -> flag(self -> delegate,
                                self, ** argv);
                    argv = self -> argv;
                }
                else
                {
                    fprintf(stderr,
                        "%s: -%c: no flags allowed\n",
                        self -> progname, ** argv);
                    return 1;
                }
            }
            while (* ++ * argv);
            ++ argv;
            continue;
        }
    }
    break;
}

```

The outer loop processes arguments until we reach the null pointer terminating the array **argv[]** or until an argument does not start with a minus sign. One or two minus signs terminate the outer loop with **break** statements.

The inner loop passes each character of one argument to the **flag**-function provided by the delegate. If the delegate decides that a flag introduces an option with a value, the method **argval()** provides a callback from the delegate to the filter to retrieve the option value:

```

% argval {                                // (self)
    const char * result;
%casts
    assert(self -> argv && * self -> argv);
    if ((* self -> argv)[1])               // -fvalue
        result = ++ * self -> argv;
    else if (self -> argv[1])              // -f value
        result = * ++ self -> argv;
    else                                   // no more argument
        result = NULL;
}

```

```

        while ((* self -> argv)[1])        // skip text
            ++ * self -> argv;

        return result;
    }

```

The option value is either the rest of the flag argument or the next argument if any. **self -> argv** is advanced so that the inner loop of **mainLoop()** terminates.

Once the options have been picked off the command line, the filename arguments remain. If there are none, a filter program works with standard input. **mainLoop()** continues as follows:

```

    if (* argv)
        do
            result = doit(self, * argv);
            while (! result && * ++ argv);
        else
            result = doit(self, NULL);

    if (self -> quit)
        result = self -> quit(self -> delegate, self);
    return result;
}

```

We let a method **doit()** take care of a single filename argument. A null pointer represents the situation that there are no arguments. **doit()** produces an exit code: only if it is zero do we process more arguments.

```

% doit {                                // (self, arg)
    FILE * fp;
    int result = 0;
%casts
    if (self -> name)
        return self -> name(self -> delegate, self, arg);

    if (! arg || strcmp(arg, "-") == 0)
        fp = stdin, clearerr(fp);
    else if (! * arg)
    {   fprintf(stderr, "%s: null filename\n",
                                self -> progname);
        return 1;
    }
    else if (! (fp = fopen(arg, "r")))
    {   perror(arg);
        return 1;
    }
}

```

The client may supply a function to process the filename argument. Otherwise, **doit()** connects to **stdin** for a null pointer or a minus sign as an argument; other filenames are opened for reading. Once the file is opened the client can take over with yet another callback function or **doit()** allocates a dynamic buffer and starts reading lines:

```

        if (self -> file)
            result = self -> file(self -> delegate, self, arg, fp);
        else
        {
            if (! self -> buf)
            {
                self -> blen = BUFSIZ;
                self -> buf = malloc(self -> blen);
                assert(self -> buf);
            }

            while (fgets(self -> buf, self -> blen, fp))
                if (self -> line && (result =
                    self -> line(self -> delegate, self, arg,
                                self -> buf)))
                    break;

            if (self -> wrap)
                result = self -> wrap(self -> delegate, self, arg);
        }

        if (fp != stdin)
            fclose(fp);

        if (fflush(stdout), ferror(stdout))
        {
            fprintf(stderr, "%s: output error\n", self -> progname);
            result = 1;
        }

        return result;
    }
}

```

With two more callback functions the client can receive each text line and perform cleanup actions once the file is complete, respectively. These are the functions that `wc` uses. **doit()** recycles the file pointer and checks that the output has been successfully written.

If a client class implements line-oriented callbacks from the **Filter** class, it should be aware of the fact that it deals with text lines. **fgets()** reads input until its buffer overflows or until a newline character is found. Additional code in **doit()** extends the dynamic buffer as required, but it only passes the buffer to the client, not a buffer length. **fgets()** does not return the number of characters read, i.e., if there is a null byte in the input, the client has no way to get past it because the null byte might actually mark the end of the last buffer of a file with no terminating newline.

10.5 The *respondsTo* Method

How does an object reach its delegate? When a **Filter** object is constructed it receives the delegate object as an argument. The class description *Filter.d* defines function types for the possible callback functions and object components to hold the pointers:

```

typedef void (* flagM) (void *, void *, char);
typedef int (* nameM) (void *, const void *, const char *);
typedef int (* fileM) (void *, const void *, const char *,
                      FILE *);

```

```

typedef int (* lineM) (void *, const void *, const char *,
                      char *);
typedef int (* wrapM) (void *, const void *, const char *);
typedef int (* quitM) (void *, const void *);

% Class Filter: Object {
    Object @ delegate;
    flagM flag;           // process a flag
    nameM name;           // process a filename argument
    fileM file;           // process an opened file
    lineM line;           // process a line buffer
    wrapM wrap;           // done with a file
    quitM quit;           // done with all files

    const char * progname; // argv[0]
    char ** argv;          // current argument and byte
    char * buf;            // dynamic line buffer
    unsigned blen;         // current maximum length
%
    int mainLoop (_self, char ** argv);
    const char * argval (_self);
    const char * progname (const _self);
    int doit (_self, const char * arg);
%}

```

Unfortunately, ANSI-C does not permit a **typedef** to be used to define a function header, but a client class like **Wc** can still use the function type to make sure its callback function matches the expectations of **Filter**:

```

#include "Filter.h"

% Wc wc {                // (self, filter, fnm, buf)
%casts
    assert((lineM) wc == wc);
...

```

The assertion is trivially true but a good ANSI-C compiler will complain about a type mismatch if **lineM** does not match the type of **wc()**:

```

In function `Wc_wc':
warning: comparison of distinct pointer types lacks a cast

```

We still have not seen why our **filter** knows to call **wc()** to process an input line. **Filter_ctor()** receives the delegate object as an argument and it can set the interesting components for **filter**:

```

% Filter ctor {
    struct Filter * self = super_ctor(Filter(), _self, app);
    self -> delegate = va_arg(* app, void *);
    self -> flag = (flagM) respondsTo(self -> delegate, "flag");
    ...
    self -> quit = (quitM) respondsTo(self -> delegate, "quit");
    return self;
}

```


The trick is a new statically linked method **respondsTo()** which may be applied to any **Object**. It takes an object and a search argument and returns a suitable function pointer if the object has a dynamically linked method corresponding to the search argument.

The returned function pointer could be a selector or the method itself. If we opt for the method, we avoid the selector call when the callback function is called; however, we also avoid the parameter checking which the selector performs. It is better to be safe than to be sorry; therefore, **respondsTo()** returns a selector.

Designing the search argument is more difficult. Because **respondsTo()** is a general method for all types of methods we cannot perform type checking at compile time, but we have already shown how the delegate can protect itself. Regardless of type checking we could still let **respondsTo()** look for the selector it is supposed to return, i.e., the search argument could be the desired selector. Selector names, however, are part of the global name space of a program, i.e., if we look for a selector name we are implicitly restricted to subclasses of the class where the selector was introduced. However, the idea was not to be restricted by inheritance aspects. Therefore, **respondsTo()** uses a string as the search argument.

We are left with the problem of associating a string with a dynamically linked method. Logically this can be done in one of two places: when the method is declared in the class description file or when it is implemented in the implementation file. Either way it is a job for *ooc* because the association between the string tag and the method name must be stored in the class description so that **respondsTo()** can find it there. The class description, however, is constructed by *ooc*. We use a simple syntax extension:

```
% WcClass: Class Wc: Object {
    ...
%-
line:  int wc (_self, const Object @ filter,          \
          const char * fnm, char * buf);
wrap:  int printFile (_self, const Object @ filter,   \
          const char * fnm);
quit:  int printTotal (_self, const Object @ filter);
%}
```

In a class description file like *Wc.d* a tag may be specified as a label preceding a dynamically linked method. By default, the method name would be used as a tag. An empty label suppresses a tag altogether — in this case **respondsTo()** cannot find the method. Tags apply to dynamically linked methods, i.e., they are inherited. To make things more flexible, a tag can also be specified as a label in a method header in the implementation file. Such a tag is valid only for the current class.

10.6 Implementation

respondsTo() must search the class description for a tag and return the corresponding selector. Thus far, the class description only contains pointers to the methods. Clearly, the method entry in a class description must be extended:

```

typedef void (* Method) ();           // for respondsTo()

%prot

struct Method {
    const char * tag;                 // for respondsTo()
    Method selector;                  // returned by respondsTo()
    Method method;                     // accessed by the selector
};

% Class Object {
    ...
    Method respondsTo (const _self, const char * tag);

```

Method is a simple function type defined in the interface file for **Object**. Each method is recorded in a class description as a component of type **struct Method** which contains pointers to the tag, the selector, and the actual method. **respondsTo()** returns a **Method**. ANSI-C compilers will gripe about implicit casts from and to this type.

Given this design, a few more changes are required. In *Object.dc* we need to change the static initialization of the class descriptions **Object** and **Class** to use **struct Method**:

```

static const struct Class _Object = {
    { MAGIC, & _Class },
    "Object", & _Object, sizeof(struct Object),
    { "", (Method) 0, (Method) Object_ctor },
    { "", (Method) 0, (Method) Object_dtor },
    { "differ", (Method) differ, (Method) Object_differ },
    ...
};

```

The **-r** report in *r.rep* uses the **link** report in *va.rep* to generate an entry in the class description for the class representation file. The new version of the **link** report is very simple:

```

% link           // component of metaclass structure

struct Method `method ;

```

Finally, the **init** report in *c.rep* and *c-R.rep* uses the **meta-ctor-loop** in *etc.rep* to generate the loop which dynamically fills the class description. Here we also have to work with the new types:

```

% meta-ctor-loop // selector/tag/method tuples for `class

`t while ((selector = va_arg(ap, Method))) `n
`t { `t const char * tag = va_arg(ap, ` \
    const char *); `n
    `t `t Method method = va_arg(ap, Method); `n `n
    `{%- `%link-it `}
`t } `n

```

```
% link-it          // check and insert one selector/method pair
` t ` t if (selector == (Method) `method ) `n
` t ` t { ` t if (tag) `n
` t ` t ` t ` t self -> `method .tag = tag, `n
` t ` t ` t ` t self -> `method .selector = selector; `n
` t ` t ` t self -> `method .method = method; `n
` t ` t ` t continue; `n
` t ` t } `n
```

Rather than selector/method pairs we now specify selector/tag/method tuples as arguments to the metaclass constructor. This must be built into the **init** report in *c.rep*. Here is the initialization function for **Wc** generated by *ooc*:

```
static const void * _Wc;

const void * Wc (void) {
    return _Wc ? _Wc :
        (_Wc = new(WcClass(),
                    "Wc", Object(), sizeof(struct Wc),
                    wc, "line", Wc_wc,
                    printFile, "wrap", Wc_printFile,
                    printTotal, "quit", Wc_printTotal,
                    (void *) 0));
}
```

Given the selector/tag/method tuples in a class description, **respondsTo()** is easy to write. Thanks to the class hierarchy, we can compute how many methods a class description contains and we can implement **respondsTo()** entirely in the **Object** class, even though it handles arbitrary classes:

```
% respondsTo {
    if (tag && * tag) {
        const struct Class * class = classOf(_self);
        const struct Method * p = & class -> ctor; // first
        int nmeth =
            (sizeof(class) - offsetof(struct Class, ctor))
            / sizeof(struct Method); // # of Methods

        do
            if (p -> tag && strcmp(p -> tag, tag) == 0)
                return p -> method ? p -> selector : 0;
            while (++ p, -- nmeth);
    }
    return 0;
}
```

The only drawback is that **respondsTo()** explicitly contains the first method name ever, **ctor**, in order to calculate the number of methods from the size of the class description. While *ooc* could obtain this name from the class description of **Object**, it would be quite messy to construct a report for *ooc* to generate **respondsTo()** in a general fashion.

10.7 Another application — *sort*

Let us implement a small text sorting program to check if **Filter** really is reusable, to see how command line options are handled, and to appreciate that a delegate can belong to an arbitrary class.

A sort filter must collect all text lines, sort the complete set, and finally write them out. Section 7.7 introduced a **List** based on a dynamic ring buffer which we can use to collect the lines as long as we add a sorting method. In section 2.5 we implemented a simple **String** class; if we integrate it with our class hierarchy we can use it to store each line in the **List**.

Let us start with the main program which merely creates the filter with its delegate:

```
int main (int argc, char * argv [])
{
    void * filter = new(Filter(), new(Sort(), 0));
    return mainLoop(filter, argv);
}
```

Because we can attach the callback methods to any class, we can create the delegate directly in a subclass of **List**:

```
% SortClass: ListClass Sort: List {
    char rflag;
%—
    void flags (_self, Object @ filter, char flag);
    int line (_self, const Object @ filter, const char * fnm, \
                                                    char * buf);
    int quit (_self, const Object @ filter);
%}
```

To demonstrate option handling we recognize **-r** as a request to sort in reverse order. All other flags are rejected by the **flags()** method which has **flag** as a tag for **respondsTo()**:

```
% flag: Sort flags {
%casts
    assert((flagM) flags == flags);
    if (flag == 'r')
        self -> rflag = 1;
    else
        fprintf(stderr, "usage: %s [-r] [file...]\n",
                progname(filter)),
        exit(1);
}
```

Given **String** and **List**, collecting lines is trivial:

```
% Sort line {
%casts
    assert((lineM) line == line);
    addLast(self, new(String(), buf));
    return 0;
}
```

Once all lines are in, the **quit** callback takes care of sorting and writing. If there are any lines at all, we let a new method **sort()** worry about sorting the list, and then we remove each line in turn and let the **String** object display itself. We can sort in reverse order simply by removing the lines from the back of the list:

```
% Sort quit {
%casts
    assert((quitM) quit == quit);
    if (count(self))
    {   sort(self);
        do
            puto(self -> rflag ? takeLast(self)
                        : takeFirst(self), stdout);
        while (count(self));
    }
    return 0;
}
```

What about **sort()**? ANSI-C defines the library function **qsort()** for sorting arbitrary arrays based on a comparison function. Luckily, **List** is implemented as a ring buffer in an array, i.e., if we implement **sort()** as a method of **List** we should have very little trouble:

```
static int cmp (const void * a, const void * b)
{
    return differ(* (void **) a, * (void **) b);
}

% List sort {
%casts
    if (self -> count)
    {   while (self -> begin + self -> count > self -> dim)
        addFirst(self, takeLast(self));
        qsort(self -> buf + self -> begin, self -> count,
                sizeof self -> buf[0], cmp);
    }
}
```

If there are any list elements, we rotate the list until it is a single region of the buffer and then pass the list to **qsort()**. The comparison function sends **differ()** to the list elements themselves — **String_differ** was based on **strcmp()** and can, therefore, be (ab-)used as a comparison function.

10.8 Summary

An object points to its class description and the class description points to all the dynamically linked methods for the object. Therefore, an object can be asked if it will respond to a particular method. **respondsTo()** is a statically linked method for **Object**. It takes an object and a string tag as search argument and returns the appropriate selector if the tag matches a method for the object.

Tags can be specified to ooc as labels on the prototypes of dynamically linked methods in the class definition file, and as labels on a method header in the imple-

mentation file; the latter have precedence. By default, the method name is used as a tag. Empty tags cannot be found. For the implementation of **respondsTo()** a method is passed to a metaclass constructor as a triple selector/tag/method.

Given **respondsTo()**, we can implement delegates: a client object announces itself as a delegate object to a host object. The host queries the client with **respondsTo()** if it can answer certain method calls. If it does, the host will use these methods to inform the client of some state changes.

Delegates are preferable to registering callback functions and to abstract base classes for defining the communication between a host and a client. A callback function cannot be a method because the host does not have an object to call the method with. An abstract base class imposes unnecessary restrictions on application-oriented development of the class hierarchy. Similar to callback functions, we can implement for delegates just those methods which are interesting for a particular situation. The set of possible methods can be much larger.

An application framework consists of one or more objects which provide the typical structure of an application. If it is well designed, it can save a great deal of routine coding. Delegates are a very convenient technique to let the application framework interact with the problem-specific code.

10.9 Exercises

Filter implements a standard command line where options precede filename arguments, where flags can be bundled, and where option values can be bundled or specified as separate arguments. Unfortunately, *pr(1)* is a commonly available program that does not fit this pattern. Is there a general solution? Can a flag introduce two or more argument values which all appear as separate arguments?

The **line** callback should be modified so that binary files can be handled correctly. Does it make sense to provide a **byte** callback? What is an alternative?

A much more efficient, although not portable, implementation would try to map a file into memory if possible. The callback interface does not necessarily have to be modified but a modification would make it more robust.

respondsTo() has to know the name of the first **struct Method** component of every class description. The reports **-r** in **r-R.rep** or rather **init** in *c-R.rep* can be modified to define a structure to circumvent this problem.

The **init** report can be modified to generate a **puto()** method for **Class** which uses the same technique as **respondsTo()** to display all method tags and addresses.

Piping the output of our *sort* program into the official *sort(1)* for checking may produce a surprise:

```
$ sort -r Sort.d | /usr/bin/sort -c -r
sort: disorder: int quit (_self, const Object @ filter);
```

There are more efficient ways for **List_sort()** to compact the list in the ring buffer before passing it to **qsort()**. Are we really correct in rotating it?

11 Class Methods Plugging Memory Leaks

Modern workstations have lots of memory. If a program loses track of a byte here and there it will probably not make a whole lot of difference. However, memory leaks are usually indicative of algorithmic errors — either the program reacts in unexpected ways to strange input or, worse, the program was inadvertently designed to break connections to dynamically allocated memory. In this chapter we will look at a general technology available with object-oriented programming which can be used, among other things, to combat memory leaks.

11.1 An Example

All resources acquired by a program should be properly recycled. Dynamic memory is a resource and production programs should certainly be checked for memory leaks. As an example, consider what happens when we make a syntax error while using the calculator developed in the third and fifth chapter:

```
$ value
(3 * 4) --
bad factor: '' 0x0
```

The recursive descent algorithm tries to build an expression tree. If something goes wrong, the **error()** function uses **longjmp()** to eliminate whatever is on the stack and continue processing in the main program. The stack, however, contains the pieces of the expression tree built thus far. If there is a syntax error, these pieces are lost: we have a memory leak. This is, of course, a standard problem in constructing interpreters.

NeXTSTEP provides a simple application *MallocDebug* which can be used to locate at least some of the more serious problems. If we link *value* with **-IMallocDebug**, the standard versions of **malloc()** and related functions are replaced by a module that can communicate with the *MallocDebug* application. We start *MallocDebug* after *value*, connect the two, and push a button **Leaks** once we have received the first error message. Unfortunately, the output is simply:

```
No nodes.
```

MallocDebug uses a fairly naive method to check for leaks: it has a list of all allocated areas and scans the words in the client task to see if they point to allocated areas. Only areas to which no word in the client task points are considered to be memory leaks. For the input

```
(3 * 4) --
```

sum() will have the first subtree built by **product()** before **factor()** runs into the end of the input line. However, when **error()** clips the stack from **factor()** back to **main()**, the address of the root of this subtree is still in the local variable **result** of **sum()** and, by chance, does not get overwritten in the **longjmp()**. The remaining

nodes are connected to the root, i.e., from the point of view of *MallocDebug*, all nodes can still be reached. However, if we enter another expression the old stack is overwritten and *MallocDebug* will find the leak.

value:

```
$ value
(3 * 4) --
bad factor: '' 0x0
1 + 3
      4
```

MallocDebug:

Zone:	Address:	Size:	Function:
default	0x050ec35c	12	mkBin, new, product, sum, factor, product, sum, stmt

If *value* is compiled with debugging information, we can start a debugger in a second window and investigate the leak:

```
$ gdb value
GDB is free software ...
(gdb) attach 746
Attaching program `value', pid 746
0x5007be2 in read ()
(gdb) print * (struct Bin *) 0x050ec35c
Reading in symbols for mathlib.c...done.
$1 = {
  type = 0x8024,
  left = 0x50ec334,
  right = 0x50ec348
}
(gdb) print process(0x050ec35c)
Reading in symbols for value.c...done.
$3 = void
(gdb)
```

The GNU debugger can be attached to a running process. With **print** we can display the contents of the leaky node if we copy the address from the *MallocDebug* window and supply the proper type: **mkBin()** was the original caller of **malloc()**, i.e., we must have obtained a **struct Bin**. As the output shows, **print** can even call a method like **process()** in *value* and display the result. The output from **process()** appears in the window where *value* is running:

```
$ value
(3 * 4) --
bad factor: '' 0x0
1 + 3
      4
      12
```

The memory leak is alive and well.

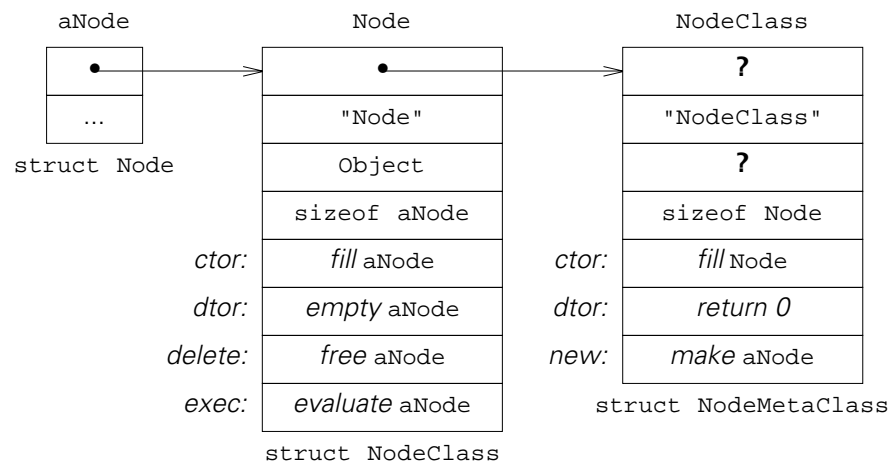
11.2 Class Methods

How do we plug this specific memory leak? The leak has occurred by the time **error()** returns to the main loop. Either we collect and release all expression pieces before **longjmp()** is executed, or we need a different way to reclaim the allocated nodes.

Collecting the pieces is a lost cause because they are held by various activations of the functions involved in the recursive descent algorithm. Only each activation knows what must be released, i.e., in place of a **longjmp()** we would have to cope with error returns in every function. This is likely to be botched once the program is later extended.

Designing a reclamation mechanism is a much more systematic approach for solving this problem. If we know what nodes are currently allocated for the expression tree we can easily release them and reclaim the memory in case of an error. What we need are versions of **new()** and **delete()** which maintain a linear list of allocated nodes which a function like **reclaim()** can traverse to free memory. In short, for expression tree nodes we should overwrite what **new()** and **delete()** do.

delete() is sent to objects, i.e., it is a method that can be given dynamic linkage so that it may be overwritten for a subtree of the class hierarchy. **new()**, however, is sent to a class description. If we want to give **new()** dynamic linkage, we must add its pointer to the class description of the class description object, to which we want to send **new()**:



With this arrangement we can give **new()** dynamic linkage for the call

```
new(Node, ...)
```

However, we create a problem for the descriptions of class descriptions, i.e., at the right edge of this picture. If we start to introduce new method components in metaclass descriptions such as **NodeClass**, we can no longer use **struct Class** to store them, i.e., our diagram must be extended at least one more level to the right before we might be able to tie it to the original **Class**.

Why did we decide to store methods in class descriptions? We assume that we have many objects and few classes. Storing methods in class descriptions rather than in the objects themselves costs one level of indirection, i.e., the dereferencing of the pointer from the object to its class description, but it avoids the high memory requirement of letting each object contain all method pointers directly.

There are fewer class descriptions than other objects; therefore, the expense of storing a method address directly in the class description to which the method is applied is not as prohibitive as it would be for other objects. We call such methods *class methods* — they are applied to the class description in which they are stored rather than to the objects sharing this class description.

A typical class method is **new()** which would be overwritten to manipulate memory allocation: provide statistics or a reclamation mechanism; allocate objects in memory zones to improve the paging behavior of a program; share memory between objects, etc. Other class methods can be introduced, for example, if we want to circumvent the convention that **new()** always calls the constructor **ctor()**.

11.3 Implementing Class Methods

The internal difference between a class method and another dynamically linked method is encapsulated in the selector. Consider **exec()**, a dynamically linked method to evaluate a node. The selector applies **classOf()** to get the class description and looks for **.exec** in there:

```
double exec (const void * _self) {
    const struct NodeClass * class =
        cast(NodeClass(), classOf(_self));

    assert(class -> exec.method);
    return ((double (*) ()) class -> exec.method)(_self);
}
```

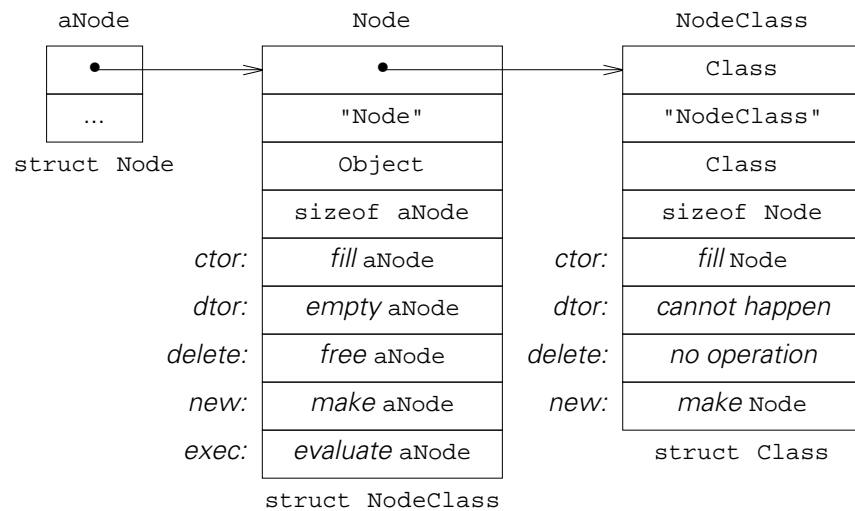
In contradistinction, consider **new()**, a class method which is applied to a class description. In this case **self** refers to the class description itself and the selector looks for **.new** as a component of ***self**:

```
struct Object * new (const void * _self, ...) {
    struct Object * result;
    va_list ap;
    const struct Class * self = cast(Class(), _self);

    assert(self -> new.method);
    va_start(ap, _self);
    result = ((struct Object * (*) ()) self -> new.method)
        (_self, & ap);

    va_end(ap);
    return result;
}
```

Here is a picture describing the linkage of **exec()** and **new()**:



Both, class methods and dynamically linked methods, employ the same super-class selector because it receives the class description pointer as an explicit argument.

```
struct Object * super_new (const void * _class,
                          const void * _self, va_list * app) {
    const struct Class * superclass = super(_class);
    assert(superclass -> new.method);
    return
        ((struct Object * (*) ()) superclass -> new.method)
            (_self, app);
}
```

Selectors are generated by *ooc* under control of the **selectors** report in *etc.rep*. Because the selectors differ for class methods and dynamically linked methods, *ooc* needs to know the method linkage. Therefore, class methods are specified in the class description file following the dynamically linked methods and the separator **%+**. Here is an excerpt from *Object.d*:

```
% Class Object {
    ...
%
    const Class @ classOf (const _self);    // object's class
    ...
%-
    void * ctor (_self, va_list * app);    // constructor
    ...
    void delete (_self);                    // reclaim instance
%+
    Object @ new (const _self, ...);    // create instance
%}
```

delete() is moved to the dynamically linked methods and **new()** is introduced as a class method.

```
% Class Class: Object {
    ...
%
    Object @ allocate (const _self);    // memory for instance
    ...
%}
```

Once we remove **new()** as a statically linked method for **Class**, we package the memory allocation part as a new statically linked method **allocate()**.

Given **%-** and **%+** as separators, ooc knows the linkage of every method and the report **selectors** can be extended to generate the selectors shown above. Other reports generate selector declarations for the interface file, superclass selector declarations and the layout of the metaclass description for the representation file, the loop in the metaclass constructor which recognizes selector/tag/method triples and enters them into the class description, and, finally, the initialization functions for class and metaclass descriptions. All of these reports need to be extended. For example, in the report **-h** in *h.rep* the declarations of dynamically linked methods are generated with

```
`{% - `%header ; `n `}n
```

A new loop adds the class method declarations:

```
`{% + `%header ; `n `}n
```

{+} is a loop over the class methods of the current class.

Once we access **new()** and **delete()** through selectors, we have to implement them for **Object** in *Object.dc*:

```
% Object new {
%casts
    return ctor(allocate(self), app);
}
```

new() creates the area for the new object and calls the appropriate constructor to initialize it. **allocate()** contains most of the old code of **new()**. It obtains dynamic memory and installs the class description pointer so that the dynamic linkage of **ctor()** in **new()** works correctly:

```
% allocate {
    struct Object * object;
%casts
    assert(self -> size);
    object = calloc(1, self -> size);
    assert(object);
    object -> magic = MAGIC;
    object -> class = self;
    return object;
}
```

delete() calls the destructor **dtor()** as before and passes the result to **free()**:

```
% Object delete {
%casts
    free(dtor(self));
}
```

Whenever we add new methods to **Object** which are accessed through selectors, we must not forget to install them by hand in the class descriptions in *Object.dc*. As an example here is **_Object**:

```
static const struct Class _Object = {
    { MAGIC, & _Class },
    "Object", & _Object, sizeof(struct Object),
    { "", (Method) 0, (Method) Object_ctor },
    ...
    { "delete", (Method) delete, (Method) Object_delete },
    ...
    { "", (Method) 0, (Method) Object_new },
};
```

11.4 Programming Savvy — A Classy Calculator

With the technology for plugging memory leaks in place, we can now engineer our calculator to take advantage of the class hierarchy. First we need to add the descriptions from chapter 5 to the hierarchy.

Node

The basic building block for the expression tree is **Node**, an abstract base class. A **Number** is a **Node** which contains a floating point constant:

```
// new(Number(), value)
% NodeClass Number: Node {
    double value;
%}
```

Our tree can grow if we have nodes with subtrees. A **Monad** has just one subtree, a **Dyad** has two:

```
% NodeClass Monad: Node {
    void * down;
%}
%prot
#define down(x) (((struct Monad *)(x)) -> down)
% NodeClass Dyad: Node {
    void * left;
    void * right;
%}
%prot
#define left(x) (((struct Dyad *)(x)) -> left)
#define right(x) (((struct Dyad *)(x)) -> right)
```

Technically, **.down**, **.left** and **.right** should only be filled by the constructors for these nodes, but if we plan to copy a tree, a subclass may need to modify the pointers.

We use single subtrees to build two entirely different things. **Val** is used to get the value from a symbol in the symbol table and **Unary** represents an operator such as a minus sign:

```
% NodeClass Val: Monad {
%}

// new(Minus(), subtree)

% NodeClass Unary: Monad {
%}

% NodeClass Minus: Unary {
%}
```

One kind of **Val** is a **Global** which points to a **Var** or **Const** symbol and obtains its value from there. If we implement user defined functions we use a **Parm** to fetch the value of a single parameter.

```
// new(Global(), constant-or-variable)
// new(Parm(), function)

% NodeClass Global: Val {
%}

% NodeClass Parm: Val {
%}
```

We will derive symbol table entries from a base class **Symbol** which is independent of **Node**. Therefore, we need **Val** and its subclasses because we can no longer let an expression tree point directly to a **Symbol** which would not understand the **exec()** method.

There are many nodes with two subtrees. **Add**, **Sub**, **Mult**, and **Div** combine the values of their subtrees; we can simplify things by inserting **Binary** as a common base class for these:

```
// new(Add(), left-subtree, right-subtree)
...

% NodeClass Binary: Dyad {
%}

% NodeClass Add: Binary {
%}
...
```

Just as **Val** is used to access symbol values, **Ref** is used to combine a symbol and an expression tree: **Assign** points to a **Var** and stores the value of its other subtree there; **Builtin** points to a **Math** symbol which computes the value of a library function for the value of **Builtin**'s right subtree as an argument; **User**, finally, points to a **Fun** symbol which computes the value of a user defined function for the value of **User**'s other subtree as an argument.

```
// new(Assign(), var, right-subtree)
// new(Builtin(), math, arg-subtree)
// new(User(), fun, arg-subtree)

% NodeClass Ref: Dyad {
%}

% NodeClass Assign: Ref {
%}

% NodeClass Builtin: Ref {
%}

% NodeClass User: Ref {
%}
```

For the most part, the methods for **Node** subclasses can be copied from the solution in chapter 5. Very little adapting is required. The following table shows how the various methods are linked into **Node** and its subclasses:

CLASS	DATA	METHODS
Node		<i>see below</i>
Number	value	ctor, exec
Monad	down	ctor
Val		exec
Global		
Parm		
Unary		dtor
Minus		exec
Dyad	left, right	ctor
Ref		dtor
Assign		exec
Builtin		exec
User		exec
Binary		dtor
Add		exec
Sub		exec
Mult		exec
Div		exec

While we are violating the principle that constructors and destructors should be balanced, we do so for a reason: the destructors send **delete()** to their subtrees. This is acceptable as long as we delete an expression subtree, but we clearly should not send **delete()** into the symbol table. **Val** and **Ref** were introduced exactly to factor the destruction process.

At this point it looks as if we need not distinguish **Global** and **Parm**. However, depending on the representation of their symbols, we may have to implement different **exec()** methods for each. Introducing the subclasses keeps our options open.

Symbol

Looking at possible expression trees we have discovered the necessary nodes. In turn, once we design the nodes we find most of the symbols which we need. **Symbol** is the abstract base class for all symbols that can be entered into a symbol table and located by name. A **Reserved** is a reserved word:

```
// new(Reserved(), "name", lex)

% Class  Reserved: Symbol {
%}
```

A **Var** is a symbol with a floating point value. **Global** will point to a **Var** symbol and use **value()** to obtain the current value; **Assign** similarly uses **setvalue()** to deposit a new value:

```
// new(Var(), "name", VAR)

% Class  Var: Symbol {
    double value;
%
    double value (const _self);
    double setvalue (_self, double value);
%}
```

A **Const** is a **Var** with a different constructor:

```
// new(Const(), "name", CONST, value)

% Class  Const: Var {
%}
```

If we make **Const** a subclass of **Var** we avoid the glitches that **setvalue()** would have to access **.value** in the base class and that we would have to initialize a **Var** during construction. We will syntactically protect **Const** from being the target of an **Assign**.

A **Math** represents a library function. **Builtin** uses **mathvalue()** to pass an argument in and receive the function value as a result:

```
// new(Math(), "name", MATH, function-name)

typedef double (* function) (double);

% Class  Math: Symbol {
    function fun;
%
    double mathvalue (const _self, double value);
%}
```

Finally, a **Fun** represents a user defined function with a single parameter. This symbol points to an expression tree which can be originally set or later replaced with **setfun()** and evaluated by a **User** node with **funvalue()**:


```
// new(Fun(), "name", FUN)
% Class Fun: Var {
    void * fun;
%
    void setfun (_self, Node @ fun);
    double funvalue (_self, double value);
%}
```

Ignoring recursion problems, we define **Fun** as a subclass of **Var** so that we can store the argument value with **setvalue()** and build a **Parm** node into the expression wherever the value of the parameter is required. Here is the class hierarchy for **Symbol**:

CLASS	DATA	METHODS
Symbol	name, lex	<i>see below</i>
Reserved		delete
Var	value	% value, setvalue
Const		ctor, delete
Fun	fun	% setfun, funvalue
Math	fun	ctor, delete % mathvalue

Again, almost all the code can be copied from chapter 5 and requires little adapting to the class hierarchy. **Const** and **Math** should never be deleted; therefore, we can add dummy methods to protect them:

```
% : Const delete {          // don't respondTo delete
}
```

The only new idea are user defined functions which are implemented in the class **Fun**:

```
% Fun setfun {
%casts
    if (self -> fun)
        delete(self -> fun);
    self -> fun = fun;
}
```

If we replace a function definition we must first delete the old expression tree, if any.

```
% Fun funvalue {
%casts
    if (! self -> fun)
        error("undefined function");
    setvalue(self, value); // argument for parameter
    return exec(self -> fun);
}
```

In order to compute the function value, we import the argument value so that **Parm** can use **value()** to retrieve it as a parameter value. **exec()** can then compute the function value from the expression tree.

Symtab

We could try to extend a **List** as a symbol table, but the binary search function used in chapter 5 must be applied to arrays and we only need the methods **screen()** and **install()**:

```
// new(Symtab(), minimal-dimension)

#include <stddef.h>

% Class Symtab: Object {
    const void ** buf;          // const void * buf [dim]
    size_t dim;                 // current buffer dimension
    size_t count;               // # elements in buffer
%
    void install (_self, const Symbol @ entry);
    Symbol @ screen (_self, const char * name, int lex);
%}
```

The array is allocated just as for a **List**:

```
% Symtab ctor {
    struct Symtab * self = super_ctor(Symtab(), _self, app);

    if (! (self -> dim = va_arg(* app, size_t)))
        self -> dim = 1;

    self -> buf = malloc(self -> dim * sizeof(void *));
    assert(self -> buf);
    return self;
}
```

search() is an internal function which uses **binary()** to search for a symbol with a particular name or to enter the name itself into the table:

```
static void ** search (struct Symtab * self, const char ** np)
{
    if (self -> count >= self -> dim)
    {
        self -> buf = realloc(self -> buf,
                               (self -> dim *= 2) * sizeof(void *));
        assert(self -> buf);
    }
    return binary(np, self -> buf, & self -> count,
                  sizeof(void *), cmp);
}
```

This is an internal function; therefore, we use a little trick: **binary()** will look for a symbol, but if it is not found **binary()** will enter the string at ***np** rather than a symbol. **cmp()** compares the string to a symbol — if we used a string class like **Atom** we could implement **cmp()** with **differ()**:

```
static int cmp (const void * _key, const void * _elt)
{
    const char * const * key = _key;
    const void * const * elt = _elt;

    return strcmp(* key, name(* elt));
}
```

name() is a **Symbol** method returning the name of a symbol. We compare it to the string argument of **search()** and do not create a symbol before we know that the search really is unsuccessful.

With table search and entry in place, the actual **Symtab** methods are quite simple to implement. **install()** is called with a second argument produced by **new()**. This way we can enter arbitrary **Symbol** objects into the symbol table:

```
% Symtab install {
    const char * nm;
    void ** pp;
%casts
    nm = name(entry);
    pp = search(self, & nm);
    if (* pp != nm)          // found entry
        delete(* pp);
    * pp = (void *) entry;
}
```

install() is willing to replace a symbol in the table.

```
% Symtab screen {
    void ** pp;
%casts
    pp = search(self, & name);
    if (* pp == name)          // entered name
    {
        char * copy = malloc(strlen(name) + 1);
        assert(copy);
        * pp = new(Symbol(), strcpy(copy, name), lex);
    }
    return * pp;
}
```

screen() either finds an entry by name or makes a new **Symbol** with a dynamically stored name. If we later decide that the table entry should rather belong to a subclass of **Symbol** we can call **install()** to replace an entry in the table. While this is a bit inefficient, it requires no new functions for the symbol table interface.

The Abstract Base Classes

Symbol is the base class for symbol table entries. A **Symbol** consists of a name and a token value for the parser which are both passed in during construction:

Symbol.d

```
// new(Symbol(), "name", lex)          "name" must not change
% Class Symbol: Object {
    const char * name;
    int lex;
%
    const char * name (const _self);
    int lex (const _self);
%}
```

Symbol.dc

```
% Symbol ctor {
    struct Symbol * self = super_ctor(Symbol(), _self, app);

    self -> name = va_arg(* app, const char *);
    self -> lex = va_arg(* app, int);
    return self;
}
```

We let **Symbol** assume that external arrangements have been made for a symbol name to be reasonably permanent: either the name is a static string or the name must be saved dynamically before a symbol is constructed with it. **Symbol** neither saves the name nor deletes it. If **screen()** saves a name dynamically, and if we decide to replace a symbol using **install()**, we can simply copy the name from the previous symbol which is deleted by **install()** and avoid more traffic in dynamic memory. Using a class like **Atom** would be a much better strategy, however.

The really interesting class is **Node**, the abstract base class for all parts of an expression tree. All new nodes are collected into a linear list so that we can reclaim them in case of an error:

Node.d

```
% NodeClass: Class Node: Object {
    void * next;
%
    void sunder (_self);
%-
    double exec (const _self);
%+
    void reclaim (const _self, Method how);
%}
```

Node.dc

```
static void * nodes;          // chains all nodes

% Node new {
    struct Node * result =
        cast(Node(), super_new(Node(), _self, app));

    result -> next = nodes, nodes = result;
    return (void *) result;
}
```

According to Webster's, *sunder* means to "sever finally and completely or with violence" and this is precisely what we are doing:

```
% Node sunder {
%casts
    if (nodes == self)          // first node
        nodes = self -> next;
```

```

        else if (nodes) // other node
        {
            struct Node * np = nodes;

            while (np -> next && np -> next != self)
                np = np -> next;

            if (np -> next)
                np -> next = self -> next;
        }
        self -> next = 0;
    }
}

```

Before we delete a node, we remove it from the chain:

```

% Node delete {
%casts
    sunder(self);
    super_delete(Node(), self);
}

```

Plugging the Memory Leaks

Normally, the parser in *parse.c* will call **delete()** after it is done with an expression:

```

if (setjmp(onError))
{
    ++ errors;
    reclaim(Node(), delete);
}

while (gets(buf))
    if (scan(buf))
    {
        void * e = stmt();

        if (e)
        {
            printf("\t%g\n", exec(e));
            delete(e);
        }
    }
}

```

If something goes wrong and **error()** is called, **reclaim()** is used to apply **delete()** to all nodes on the chain:

```

% Node reclaim {
%casts
    while (nodes)
        how(nodes);
}

```

This plugs the memory leak described at the beginning of this chapter — *MallocDebug* does not find any leaks, neither immediately after an error nor later. For test purposes we can

```
reclaim(Node, sunder);
```

after an error and let *MallocDebug* demonstrate that we really have lost nodes.

The elegance of the scheme lies in the fact that the entire mechanism is encapsulated in the base class **Node** and inherited by the entire expression tree. Given

class functions, we can replace **new()** for a subtree of the class hierarchy. Replacing **new()** exactly for all nodes, but not for symbols or the symbol table, provides reclamation for broken expressions without damaging variables, functions, and the like.

Technically, **reclaim()** is declared as a class method. We do not need the ability to overwrite this function for a subclass of **Node**, but it does leave room for expansion. **reclaim()** permits a choice as to what should be applied to the chain. In case of an error this will be **delete()**; however, if we save an expression for a user defined function in a **Fun** symbol, we need to apply **sunder()** to the chain to keep the next error from wiping out the expression stored in the symbol table. When a function is replaced, **setfun()** will delete the old expression and **delete()** still uses **sunder()** — this is why **sunder()** does not demand to find its argument on the chain.

11.5 Summary

Class Methods are applied to class descriptions, rather than to other objects. We need at least one class method: **new()** creates objects from a class description.

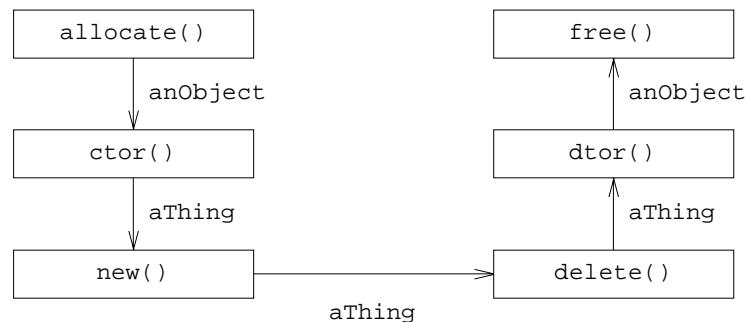
Just like other methods, class methods can have static or dynamic linkage, but the syntax of *ooc* only permits static linkage for class methods that apply to the root metaclass. Therefore, the term *class method* has been introduced here to only describe a method with dynamic linkage that is applied to a class description.

Since there are relatively few class descriptions, we can provide the dynamic linkage for a class method by storing it in the class description itself to which it applies. This has two advantages: we can overwrite class methods for a subclass without introducing a new metaclass to store it; and our basic scheme remains intact where objects point to class descriptions, class descriptions point to their own descriptions, and the latter can all be stored in **struct Class**, i.e., they will all point to **Class**, thus completing the class hierarchy in a clean fashion.

Defining **new()** as a class method for **Object** rather than as a method with static linkage for **Class** permits redefining **new()** for subtrees of the class hierarchy. This can be used for memory allocation tracking, memory sharing, etc. *ooc* makes no provisions for extending the data part of a class description. If it did, a class method could have local data applicable to its class as a whole and we could count objects per class, etc. **static** variables in an implementation file are not quite the same because they exist once for the class and all its subclasses.

There is a tradeoff between **new()** and a constructor. It is tempting to do all the work in **new()** and leave the constructor empty, but then invariants normally established by the constructor can break once **new()** is overwritten. Similarly, a constructor is technically capable of substituting a different memory area in place of the one passed in from **new()** — this was demonstrated in the implementation of **Atom** in section 2.6 — but a proper life cycle for this memory is difficult to maintain.

As a rule of thumb, class methods like **new()** should only connect an allocation function with a constructor and refrain from doing any initializations themselves. Allocation functions such as **allocate()** should initialize the class description pointer — too much can go horribly wrong if they do not. Reclamation functions such as **delete()** should let the destructor dispose of the resources which the constructor and the object's life cycle have accumulated, and only pass the empty memory area to a recycler function like **free()**:



There is a balance: **allocate()** and **free()** deal with the same region of memory; by default, **new()** gives it to its constructor, **delete()** to its destructor; and the constructor and destructor only deal with resources represented inside the object. **new()** and **delete()** should only be overwritten to interfere with the flow of memory from **allocate()** to **free()**.

11.6 Exercises

For the *ooc* parser it makes absolutely no difference, if class methods are described before or after dynamically linked methods in the class description file, i.e., if **%+** precedes or follows **%-**. There is, however, a convincing point in favor of the arrangement described in this chapter. Why can the separators not be repeated to achieve an arbitrary mix of both types of methods?

There is a rather significant difference once **delete()** is implemented with dynamic linkage. What can no longer be passed to **delete()**?

It is not helpful to move **value()** back into the abstract base class **Symbol** and give it dynamic linkage there. **mathvalue()** is applied to a **Math** symbol and requires a function argument, **value()** is applied to a **Var** or **Const** symbol and has no use for an argument. Should we use variable argument lists?

We can detect recursion among user defined functions. We can use words like **\$1** to support functions with more than one parameter. We can even add parameters with names that hide global variables.

If we add a generic pointer to the data area of **Class** in *Object.d* class methods can attach a chain of private data areas there. This can be used, e.g., to count objects or to provide object lists per class.

12

Persistent Objects

Storing and Loading Data Structures

Section 6.3 introduced the dynamically linked method **puto()** in class **Object** which takes an object and describes it on a stream. For example,

```
void * anObject = new(Object());
...
puto(anObject, stdout);
```

produces about the following standard output:

```
Object at 0x5410
```

If we implement **puto()** for every class in a hierarchy we can display every object. If the output is designed well enough, we should be able to recreate the objects from it, i.e., objects can be parked in files and continue to exist from one invocation of an application to another. We call such objects *persistent*. Object oriented databases consist of persistent objects and mechanisms to search for them by name or content.

12.1 An Example

Our calculator contains a symbol table with variables, constants, and functions. While constants and mathematical functions are predefined, we loose all variable values and user defined function definitions whenever we terminate execution. As a realistic example for using persistent objects we add two statements to the calculator: **save** stores some or all variables and function definitions in files; **load** retrieves them again.

```
$ value
def sqr = $ * $
def one = sqr(sin($)) + sqr(cos($))
let n = one(10)
1
save
```

In this session with *value* we define the functions **sqr()** and **one()** to test that $\sin^2 x + \cos^2 x \equiv 1$. Additionally, we create the variable **n** with value 1. **save** without arguments writes the three definitions into a file *value.stb*.

```
$ value
load
n + one(20)
2
```

Once we start *value* again we can use **load** to retrieve the definitions. The expression demonstrates that we recover the value of the variable and both function definitions.

save is implemented in the parser function **stmt()** just like **let** or **def**. With no argument we have to store all variables and functions; therefore, we simply pass the problem to **Symtab**:

```
#define SYMTABFILE  "value.stb"
#define SYMBOLFILE  "%s.sym"

static void * stmt (void)
{   void * sym, * node;

    switch (token) {
        ...
    case SAVE:
        if (! scan(0))                /* entire symbol table */
        {   if (save(table, 0, SYMTABFILE))
            error("cannot save symbol table");
        }
        else                          /* list of symbols */
        do
        {   char fnm [BUFSIZ];

            sprintf(fnm, SYMBOLFILE, name(symbol));
            if (save(table, symbol, fnm))
                error("cannot save %s", name(symbol));
        } while (scan(0));
    }
    return 0;
}
```

A more complicated syntax could permit a file name specification as part of **save**. As it is, we predefine **SYMTABFILE** and **SYMBOLFILE**: the entire symbol table is saved in *value.stb* and a single symbol like **one** would be filed in *one.sym*. The application controls the file name, i.e., it is constructed in *parse.c* and passed to **save()**.

Symtab.d

```
% Class  Symtab: Object {
    ...
%
    ...
    int save (const _self, const Var @ entry, const char * fnm);
    int load (_self, Symbol @ entry, const char * fnm);
%}
```

Symtab.dc

```
% Symtab save {
    const struct Symtab * self = cast(Symtab(), _self);
    FILE * fp;

    if (entry)                                // one symbol
    {   if (! respondsTo(entry, "move"))
        return EOF;
        if (! (fp = fopen(fnm, "w")))
            return EOF;
        puto(entry, fp);
    }
}
```

A single symbol is passed as **entry**. There is no point in saving undefined symbols, constants, or math functions. Therefore, we only save symbols which support the method **move()**; as we shall see below, this method is used in loading a symbol from a file. **save()** opens the output file and lets **puto()** do the actual work. Saving all symbols is almost as easy:

```

else                                     // entire table
{   int i;

    if (! (fp = fopen(fnm, "w")))
        return EOF;
    for (i = 0; i < self -> count; ++ i)
        if (respondsTo(self -> buf[i], "move"))
            puto(self -> buf[i], fp);
}

return fclose(fp);                       // 0 or EOF
}

```

save() is defined as a method of **Symtab** because we need to loop over the elements in **.buf[]**. The test whether or not a symbol should be stored in a file is not repeated outside of **save()**. However, **Symtab** should not know what kinds of symbols we have. This is why the decision to store is based on the acceptance of a **move()** and not on membership in certain subclasses of **Symbol**.

It looks like loading should be totally symmetrical to storing. Again, *parse.c* decides on the file name and lets **load()** do the actual work:

```

case LOAD:
    if (! scan(0))                       /* entire symbol table */
    {   if (load(table, 0, SYMTABFILE))
        error("cannot load symbol table");
    }
    else                                 /* list of symbols */
    do
    {   char fnm [BUFSIZ];

        sprintf(fnm, SYMBOLFILE, name(symbol));
        if (load(table, symbol, fnm))
            error("cannot load %s", name(symbol));
    } while (scan(0));
    reclaim(Node(), sunder);
    return 0;

```

Unfortunately, **load()** is entirely different from **save()**. There are two reasons: we should at least try to protect our calculator from somebody who tinkers with file names or contents; and while **save()** can just display something in the symbol table by applying **puto()**, it is quite likely that we have to enter or modify symbols in the table during the course of a **save()**. Retrieving persistent objects is very much like allocating and constructing them in the first place.

Let us walk through **load()**. If a single symbol is to be loaded, its name is already in the symbol table. Therefore, we are either looking at an undefined **Symbol** or the symbol knows how to answer a **move()**:

```
% Symtab load {
    struct Symtab * self = cast(Symtab(), _self);
    const char * target = NULL;
    FILE * fp;
    int result = EOF;
    void * in;

    if (entry)
        if (isOf(entry, Symbol())
            || respondsTo(entry, "move"))
            target = name(entry);
        else
            return EOF;

    if (! (fp = fopen(fnm, "r")))
        return EOF;

    while (in = retrieve(fp))
        ...
    if (! target && feof(fp))
        result = 0;

    fclose(fp);
    return result;
}
```

If there is an **entry**, checking it early keeps us from working entirely in vain. Next, we access the file and try to read as many symbols from it as we can:

```
if (! (fp = fopen(fnm, "r")))
    return EOF;

while (in = retrieve(fp))
    ...
if (! target && feof(fp))
    result = 0;

fclose(fp);
return result;
}
```

If we are not looking for a particular **entry**, we are happy if we reach the end of file. **retrieve()** returns an object from a stream; this will be discussed in section 12.4.

The body of the **while** loop deals with one symbol at a time. We are in real trouble if the retrieved object does not know about **move()**, because the stream cannot possibly have been written by **save()**. If that happens, it is time to quit the loop and let **load()** return **EOF**. Otherwise, if we are looking for a particular **entry**, we skip all symbols with different names.

```
{    const char * nm;
    void ** pp;

    if (! respondsTo(in, "move"))
        break;

    if (target && strcmp(name(in), target))
        continue;
}
```

Technically, *parse.c* has set things up so that a file should either contain the desired single entry or an entire symbol table, but the **strcmp()** protects us from renamed or modified files.

We are ready to bring the retrieved symbol into the symbol table. This is why **load()** is a **Symtab** method. The process is quite similar to **screen()**: we assume that **retrieve()** has taken care to dynamically save the name and we use **search()** to locate the name in the table. If we rediscover the retrieved name, we have just read a new symbol and we can simply insert it into the table.

```

nm = name(in);
pp = search(self, & nm);

if (* pp == nm)                // not yet in table
    * pp = in;

```

Most likely, however, **load** has been given the name of a new symbol to load. In this case, the name is already in the symbol table as an undefined **Symbol** with a dynamically saved name. We remove it completely and insert the retrieved symbol in its place.

```

else if (isA(* pp, Symbol()))
    // not yet defined
{
    nm = name(* pp), delete(* pp), free((void *) nm);
    * pp = in;
}
// might free target, but then we exit below

```

If we reach this point we are faced with an existing symbol, i.e., a variable gets a new value or a function is redefined. However, we only overwrite a symbol that knows about **move()**, to protect against somebody changing the contents of our input file.

```

else if (! respondsTo(* pp, "move"))
{
    nm = name(in); delete(in); free((void *) nm);
    continue;                // should not happen
}
else
{
    move(* pp, in);
    delete(in), free((void *) nm);
}

if (target)
{
    result = 0;
    break;
}
}

```

If we found the desired **entry** we can leave the loop.

We have to be very careful not to replace an existing symbol, because some expression might already point to it. This is why **move()** is introduced as a dynamically linked method to transfer the value from one symbol to another.

Symbol.d

```

% VarClass: Class Var: Symbol {
    ...
%—
    void move (_self, _from);
%}

```

Symbol.dc

```

% Var move {
%casts
    setvalue(self, from -> value);
}

```

```

% : Const move {          // don't respondTo move
}

% Fun move {
%casts
    setfun(self, from -> fun), from -> fun = 0;
}

```

Var and **Fun** are willing to **move()**, but **Const** is not. **move()** is similar to a *shallow copy* operation: for a **Fun** which points to an expression, it transfers the pointer but does not copy the entire expression. **move()** actually clears the source pointer so that the source object may be deleted without destroying the transferred expression.

12.2 Storing Objects — *puto()*

puto() is an **Object** method which writes the representation of an object to a **FILE** pointer and returns the number of bytes written.

Object.d

```

% Class Object {
    ...
%-
    int puto (const _self, FILE * fp);          // display

```

Object.dc

```

% Object puto {
%casts
    class = classOf(self);
    return fprintf(fp, "%s at %p\n", class -> name, self);
}

```

As we shall see in section 12.3, it is essential that the output starts with the class name of the object. Emitting the object's address is not strictly necessary.

While each subclass is free to implement its own version of **puto()**, the easiest solution is to let **puto()** operate just like a constructor, i.e., to cascade calls up the superclass chain all the way back to **Object_puto()** and thus guarantee that the output starts with the class name and picks up the instance information from each class involved. This way each **puto** method only needs to worry about the information that is added in its own class and not about the complete content of an object. Consider a **Var** and a **Symbol**:

```

% Var puto {
    int result;
%casts
    result = super_puto(Var(), _self, fp);
    return result + fprintf(fp, "\tvalue %g\n", self -> value);
}

```

```
% Symbol puto {
    int result;
%casts
    result = super_puto(Symbol(), _self, fp);
    return result + fprintf(fp, "\tname %s\n\tlex %d\n",
                           self -> name, self -> lex);
}
```

This produces something like the following output:

```
Var at 0x50ecb18      Object
    name x           Symbol
    lex 118
    value 1          Var
```

It is tempting to streamline the code to avoid the **int** variable:

```
% Var puto {      // WRONG
%casts
    return super_puto(Var(), _self, fp)
        + fprintf(fp, "\tvalue %g\n", self -> value);
}
```

However, ANSI-C does not guarantee that the operands of an operator like **+** are evaluated from left to right, i.e., we might find the order of the output lines scrambled.

Designing the output of **puto()** is easy for simple objects: we print each component with a suitable format and we use **puto()** to take care of pointers to other objects — at least as long as we are sure not to run into a loop.

A *container class*, i.e., a class that manages other objects, is more difficult to handle. The output must be designed so that it can be restored properly, especially if an unknown number of objects must be written and read back; unlike **save()** shown in section 12.1 we cannot rely on end of file to indicate that there are no more objects.

In general, we need a prefix notation: either we write the number of objects prior to the sequence of actual objects, or we prefix each object by a character such as a plus sign and use, e.g., a period in place of the plus to indicate that there are no more objects. We could use **ungetc(getc(fp), fp)** to peek at the next character, but if we use the absence of a particular lead character to terminate a sequence, we are effectively relying on other objects not to accidentally break our scheme.

Fun in our calculator is a different kind of container class: it is a symbol containing an expression composed of **Node** symbols. **puto()** outputs the expression tree in preorder, nodes before subtrees; if the degree of each node is known, it can be easily restored from this information:

```
% Binary puto {
    int result;
%casts
    result = super_puto(Binary(), self, fp);
    result += puto(left(self), fp);
    return result + puto(right(self), fp);
}
```

The only catch is a function which references other functions. If we blindly apply **puto()** to the reference, and if we don't forbid recursive functions, we can easily get stuck. The **Ref** and **Val** classes were introduced to mark symbol table references in the expression tree. For a reference to a function we only write the function name:

```
% Ref puto {
    int result;
%casts
    result = super_puto(Ref(), self, fp);
    result += putsymbol(left(self), fp);
    return result + puto(right(self), fp);
}
```

For reasons that will become clear in the next section, **putsymbol()** is defined in *parse.c*:

```
int putsymbol (const void * sym, FILE * fp)
{
    return fprintf(fp, "\tname %s\n\tlex %d\n",
                  name(sym), lex(sym));
}
```

It is sufficient to write the reference name and token value.

12.3 Filling Objects — *geto()*

geto() is an **Object** method which reads information from a **FILE** pointer and fills an object with it. **geto()** is applied to the uninitialized object; therefore, its job is quite similar to that of a constructor like **ctor()**. However, **ctor()** takes the information for the new object from its argument list; **geto()** reads it from the input stream.

Object.d

```
% Class Object {
    ...
%-
:   void * geto (_self, FILE * fp);    // construct from file
```

An empty tag is specified by the leading colon because it does not make sense for an initialized object to **respondTo** the method **geto**.

Symbol.dc

```
% Var geto {
    struct Var * self = super_geto(Var(), _self, fp);
    if (fscanf(fp, "\tvalue %lg\n", & self -> value) != 1)
        assert(0);
    return self;
}
```

Var_geto() lets the superclass methods worry about the initial information and simply reads back what **Var_puto()** has written. Normally, the same formats can be specified for **fprintf()** in the **puto** method and for **fscanf()** in the **geto** method. However, floating point values reveal a slight glitch in ANSI-C: **fprintf()** uses **%g** to

convert a **double**, but **fscanf()** requires **%lg** to convert back. Strings usually have to be placed into dynamic memory:

```
% Symbol geto {
    struct Symbol * self = super_geto(Symbol(), _self, fp);
    char buf [BUFSIZ];

    if (fscanf(fp, "\tname %s\n\tlex %d\n",
               buf, & self -> lex) != 2)
        assert(0);
    self -> name = malloc(strlen(buf) + 1);
    assert(self -> name);
    strcpy((char *) self -> name, buf);
    return self;
}
```

Normally, **geto()** reads exactly what the corresponding **puto()** has written, and just like constructors, both methods call their superclass methods all the way back to **Object**. There is one very important difference, however: we saw that **Object_puto()** writes the class name followed by an address:

```
Var at 0x50ecb18      Object
    name x           Symbol
    lex 118
    value 1          Var
```

Object_geto() is the first method to fill the **Var** object on input. The class name **Var** written by **puto()** must be read and used to allocate a **Var** object *before* **geto()** is called to fill the object, i.e., **Object_geto()** starts reading just *after* the class name:

```
% Object geto {
    void * dummy;
%casts
    if (fscanf(fp, " at %p\n", & dummy) != 1)
        assert(0);
    return self;
}
```

This is the only place where **geto** and **puto** methods do not match exactly. The variable **dummy** is necessary: we could avoid it with the format element **%p**, but then we could not discover if the address really was part of the input.

12.4 Loading Objects — *retrieve()*

Who reads the class name, allocates the object, and calls **geto()** to fill it? Perhaps strangely, this is accomplished by the function **retrieve()** which is declared in the class description file *Object.d*, but which is not a method:

```
void * retrieve (FILE * fp);           // object from file
```

retrieve() reads a class name as a string from a stream; somehow finds the appropriate class description pointer; uses **allocate()** to create room for the object; and asks **geto()** to fill it. Because **allocate()** inserts the final class description pointer, **geto()** can actually be applied to the allocated area:


```

struct classList { const char * name; const void * class; };

void * retrieve (FILE * fp)
{
    char buf [BUFSIZ];
    static struct classList * cL;           // local copy
    static int cD = -1;                     // # classes

    if (cD < 0)
        ... build classList in cL[0..cD-1] ...

    if (! cD)
        fputs("no classes known\n", stderr);

    else if (fp && ! feof(fp) && fscanf(fp, "%s", buf) == 1)
    {
        struct classList key, * p;

        key.name = buf;
        if (p = bsearch(& key, cL, cD, sizeof key,
            (int (*)(const void *, const void *)) cmp))
            return geto(allocate(p -> class), fp);
        fprintf(stderr, "%s: cannot retrieve\n", buf);
    }
    return 0;
}

```

retrieve() needs a list of class names and class description pointers. The class descriptions point to the methods and selectors, i.e., the list actually guarantees that the code for the classes is bound with the program using **retrieve()**. If the data for an object is read in, the methods for the object are available in the program — **geto()** is just one of them.

Where does the class list come from? We could craft one by hand, but in chapter 9 we looked at *munch*, a simple *awk* program to extract class names from a listing of object modules produced by *nm*. Because *nm* can be applied to a library of object modules, we can even extract the class list supported by an entire library. The result is an array **classes[]** with a list of pointers to the class initialization functions, alphabetically sorted by class names.

retrieve() could search this list by calling each function to receive the initialized class description and applying **nameOf()** to the result to get the string representation of the class name. This is not very efficient if we have to retrieve many objects. Therefore, **retrieve()** builds a private list as follows:

```

extern const void * (* classes[]) (void); // munch

if (cD < 0)
{
    for (cD = 0; classes[cD]; ++ cD)
        ; // count classes
    if (cD > 0) // collect name/desc
    {
        cL = malloc(cD * sizeof(struct classList));
        assert(cL);
        for (cD = 0; classes[cD]; ++ cD)
            cL[cD].class = classes[cD](),
            cL[cD].name = nameOf(cL[cD].class);
    }
}

```

The private class list has the additional advantage that it avoids further calls to the class initialization functions.

12.5 Attaching Objects — *value* Revisited

Writing and reading a strictly tree-structured, self-contained set of objects can be accomplished with **puto()**, **retrieve()**, and matching **geto** methods. Our calculator demonstrates that there is a problem once a collection of objects is written which references other objects, written in a different context or not written at all. Consider:

```
$ value
def sqr = $ * $
def one = sqr(sin($)) + sqr(cos($))
save one
```

The output file *one.sym* contains references to **sqr** but no definition:

```
$ cat one.sym
Fun at 0x50ec9f8
  name one
  lex 102
  value 10
=
Add at 0x50ed168
User at 0x50ed074      Ref
  name sqr            putsymbol
  lex 102
Builtin at 0x50ecfd0   Ref
  name sin            putsymbol
  lex 109
Parm at 0x50ecea8      Val
  name one            putsymbol
  lex 102
User at 0x50ed14c
  name sqr
  lex 102
Builtin at 0x50ed130
  name cos
  lex 109
Parm at 0x50ed118
  name one
  lex 102
```

User is a **Ref**, and **Ref_puto()** has used **putsymbol()** in *parse.c* to write just the symbol name and token value. This way, the definition for **sqr()** is intentionally not stored into *one.sym*.

Once a symbol table reference is read in, it must be attached to the symbol table. Our calculator contains a single symbol table **table** which is created and managed in *parse.c*, i.e., a reference from the expression tree to the symbol table must employ **getsymbol()** from *parse.c* to attach the reference to the current sym-

bol table. Each kind of reference employs a different subclass of **Node** so that the proper subclass of **Symbol** can be found or created by **getsymbol()**. This is why we must distinguish **Global** as a reference to a **Var** and **Parm** as a reference to a **Fun**, from where the parameter value is fetched.

```
% Global geto {
    struct Global * self = super_geto(Global(), _self, fp);

    down(self) = getsymbol(Var(), fp);
    return self;
}

% Parm geto {
    struct Parm * self = super_geto(Parm(), _self, fp);

    down(self) = getsymbol(Fun(), fp);
    return self;
}
```

Similarly, **Assign** looks for a **Var**; **Builtin** looks for a **Math**; and **User** looks for a **Fun**. They all employ **getsymbol()** to find a suitable symbol in **table**, create one, or complain if there is a symbol with the right name but the wrong class:

```
void * getsymbol (const void * class, FILE * fp)
{
    char buf [BUFSIZ];
    int token;
    void * result;

    if (fscanf(fp, "\\tname %s\\ntlex %d\\n", buf, & token) != 2)
        assert(0);
    result = screen(table, buf, UNDEF);
    if (lex(result) == UNDEF)
        install(table, result =
                    new(class, name(result), token));
    else if (lex(result) != token)
    {
        fclose(fp);
        error("%s: need a %s, got a %s",
              buf, nameOf(class), nameOf(classOf(result)));
    }
    return result;
}
```

It helps that when a **Fun** symbol is created we need not yet supply the defining expression:

```
$ value
load one
one(10)
undefined function
```

one() tries to call **sqr()** but this is undefined.

```
let sqr = 9
bad assignment
```

An undefined **Symbol** could be overwritten and assigned to, i.e., **sqr()** really is an undefined function.

```

def sqr = $ * $
one(10)
    1
def sqr = 1
one(10)
    2

```

Here is the class hierarchy of the calculator with most method definitions. Meta-classes have been omitted; boldface indicates where a method is first defined.

CLASS	DATA	METHODS
Object	magic, ...	% classOf , ... %- delete , puto , geto , ... %+ new
Node		% sunder %- delete , exec %+ new , reclaim
Number	value	%- ctor , puto , geto , exec
Monad	down	%- ctor
Val		%- puto , exec
Global		%- geto
Parm		%- geto
Unary		%- dctor , puto , geto
Minus		%- exec
Dyad	left, right	%- ctor
Ref		%- dctor , puto
Assign		%- geto , exec
Builtin		%- geto , exec
User		%- geto , exec
Binary		%- dctor , puto , geto
Add		%- exec
Sub		%- exec
Mult		%- exec
Div		%- exec
Symbol	name, lex	% name , lex %- ctor , puto , geto
Reserved		%- delete
Var	value	% value , setvalue %- puto , geto , move
Const		%- ctor , delete , move
Fun	fun	% setfun , funvalue %- puto , geto , move
Math	fun	% mathvalue %- ctor , delete
Symtab	buf, ...	% save , load , ... %- ctor , puto , delete

A slight blemish remains, to be addressed in the next chapter: **getsymbol()** apparently knows enough to close the stream **fp** before it uses **error()** to return to the main loop.

12.6 Summary

Objects are called *persistent*, if they can be stored in files to be loaded later by the same or another application. Persistent objects can be stored either by explicit actions, or implicitly during destruction. Loading takes the place of allocation and construction.

Implementing persistence requires two dynamically linked methods and a function to drive the loading process:

```
int puto (const _self, FILE * fp);
void * geto (_self, FILE * fp);
void * retrieve (FILE * fp);
```

puto() is implemented for every class of persistent objects. After calling up the superclass chain it writes the class' own instance variables to a stream. Thus, all information about an object is written to the stream beginning with information from the ultimate superclass.

geto() is also implemented for all persistent objects. The method is normally symmetric to **puto()**, i.e., after calling up the superclass chain it fills the object with values for the class' own instance variables as recorded by **puto()** in the stream. **geto()** operates like a constructor, i.e., it does not allocate its object, it merely fills it.

Output produced by **puto()** starts with the class name of the object. **retrieve()** reads the class name, locates the corresponding class description, allocates memory for an object, and calls **geto()** to fill the object. As a consequence, while **puto()** in the ultimate superclass writes the class name of each object, **geto()** starts reading *after* the class name. It should be noted that **retrieve()** can only load objects for which it knows class descriptions, i.e., with ANSI-C, methods for persistent objects must be available *a priori* in a program that intends to **retrieve()** them.

Apart from an initial class name, there is no particular restriction on the output format produced by **puto()**. However, if the output is plain text, **puto()** can also aid in debugging, because it can be applied to any object with a suitable debugger.

For simple objects it is best to display all instance variable values. For container objects pointing to other objects, **puto()** can be applied to write the client objects. However, if objects can be contained in more than one other object, **puto()** or **retrieve()** must be designed carefully to avoid the effect of a deep copy, i.e., to make sure that the client objects are unique. A reasonably foolproof solution for loading objects produced by a single application is for **retrieve()** to keep a table of the original addresses of all loaded objects and to create an object only, if it is not already in the table.

12.7 Exercises

If **retrieve()** keeps track of the original address of each object and constructs only new ones, we need a way to skip along the stream to the end of an object.

System V provides the functions **dlopen()**, **dlsym()**, and **dlclose()** for dynamic loading of shared objects. **retrieve()** could employ this technology to load a class module by name. The class module contains the class description together with all methods. It is not clear, however, how we would efficiently access the newly loaded selectors.

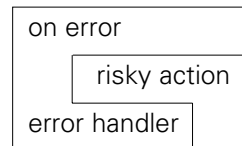
value can be extended with control structures so that functions are more powerful. In this case **stmt()** needs to be split into true statements such as **let** and commands like **save**, **load**, or **def**.

13 Exceptions Disciplined Error Recovery

Thirteen seems quite appropriate as the chapter number for coping with misfortune. If we get lost inside a single function, the much maligned **goto** is a boon for bailing out. ANSI-C's **setjmp()** and **longjmp()** do away with a nest of function activations if we discover a problem deep inside. However, if cleanup operations must be inserted at various levels of a bailout we need to harness the crude approach of **setjmp()**.

13.1 Strategy

If our calculator has trouble loading a function definition we run into a typical error recovery problem: an open stream has to be closed before we can call **error()** to produce a message and return to the main loop. The following picture indicates that a simple *risky action* should be wrapped into some error handling logic:



First, an error handler is set up. Either the risky action completes correctly or the error handler gets a chance to clean up before the compound action completes. In ANSI-C, **setjmp()** and **longjmp()** are used to implement this error recovery scheme:

```

#include <setjmp.h>

static jmp_buf onError;

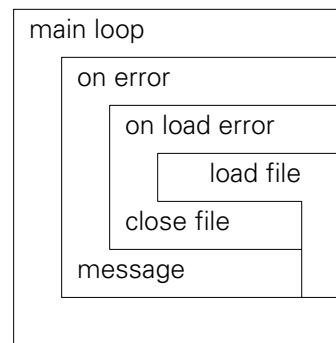
static void cause() {
    longjmp(onError, 1);
}

action () {
    if (! setjmp(onError))
        risky action
    else
        error handler
}
  
```

setjmp() initializes **onError** and returns zero. If something goes wrong in *risky action*, or in a function called from there, we signal the error by calling **cause()**. The **longjmp()** in this function uses the information in **onError** to effect a second return from the call to **setjmp()** which initialized **onError**. The second return delivers the second argument of **longjmp()** as a function value; one is returned if this value is zero. Things go horribly wrong if the function which called **setjmp()** is no longer active.

In the terminology of the picture above, *on error* refers to calling **setjmp()** to deposit the information for error handling. *risky action* is executed if **setjmp()** returns zero; or otherwise, *error handler* is executed. **cause()** is called to initiate error recovery by transferring control to *error handler*.

We have used this simple model to recover from syntax errors deep inside recursive descent in our calculator. Things get more complicated if error handlers must be nested. Here is what happens during a **load** operation in the calculator:



In this case we need two **longjmp()** targets for recovery: **onError** returns to the main loop and **onLoadError** is used to clean up after a bad loading operation:

```
jmp_buf onError, onLoadError;

#define cause(x)    longjmp(x, 1)

mainLoop () {
    if (! setjmp(onError))
        loadFile();
    else
        some problem
}

loadFile () {
    if (! setjmp(onLoadError))
        work with file
    else
        close file
        cause(onError);
}
```

The code sketch shows that **cause()** somehow needs to know how far recovery should go. We can use an argument or a hidden global structure for this purpose.

If we give **cause()** an explicit argument, it is likely that it has to refer to a global symbol so that it may be called from other files. Obviously, the global symbol must not be used at the wrong time. It has the additional drawback that it is part of the client code, i.e., while the symbol is only meaningful for a particular error handler, it is written into the code protected by the handler. If this code is called from some other place, we have to make sure that it does not inadvertently refer to an inactive recovery point.

A much better strategy is a stack of **jmp_buf** values. A function establishes an error handler by pushing this stack, and **cause()** uses the top entry. Of course, the error handler has to be popped from the stack before the corresponding function terminates.

13.2 Implementation — *Exception*

Exception is a class that provides nestable exception handling. **Exception** objects must be deleted in the reverse order of their creation. Normally, the newest object represents the error handler which receives control from a call to **cause()**.

```
// new(Exception())
#include <setjmp.h>
void cause (int number);           // if set up, goto catch()
% Class Exception: Object {
    int armed;                     // set by a catch()
    jmp_buf label;                 // used by a catch()
%
    void * catchException (_self);
%}
```

new(Exception()) creates an exception object which is pushed onto a hidden stack of all such objects:

```
#include "List.h"
static void * stack;
% Exception ctor {
    void * self = super_ctor(Exception(), _self, app);
    if (! stack)
        stack = new(List(), 10);
    addLast(stack, self);
    return self;
}
```

We use a **List** object from section 7.7 to implement the exception stack.

Exception objects must be deleted exactly in the opposite order of creation. The destructor pops them from the stack:

```
% Exception dtor {
    void * top;
%casts
    assert(stack);
    top = takeLast(stack);
    assert(top == self);
    return super_dtor(Exception(), self);
}
```

An exception is caused by calling **cause()** with a nonzero argument, the exception code. If possible, **cause()** will execute a **longjmp()** to the exception object on top of the stack, i.e., the most recently created such object. Note that **cause()** may or may not return to its caller.

```

void cause (int number) {
    unsigned cnt;

    if (number && stack && (cnt = count(stack)))
    {
        void * top = lookAt(stack, cnt-1);
        struct Exception * e = cast(Exception(), top);

        if (e -> armed)
            longjmp(e -> label, number);
    }
}

```

cause() is a function, not a method. However, it is implemented as part of the implementation of **Exception** and it definitely has access to the internal data of this class. Such a function is often referred to as a *friend* of the class.

cause() employs a number of safeguards: the argument must not be zero; the exception stack must exist, must contain objects, and the top object must be an exception; and finally, the exception object must have been armed to receive the exception. If any of the safeguards fails, **cause()** returns and its caller must cope with the situation.

An exception object must be armed before it can be used, i.e., the **jmp_buf** information must be deposited with **setjmp()** before the object will be used by **cause()**. For several reasons, creating and arming the object are two separate operations. An object is usually created with **new()**, and the object is the result of this operation. An exception object must be armed with **setjmp()**, and this function will return two integer values: first a zero, and the second time the value handed to **longjmp()**. It is hard to see how we could combine the two operations.

More importantly, ANSI-C imposes severe restrictions as to where **setjmp()** may be called. It has to pretty much be specified alone as an expression and the expression can only be used as a statement or to govern a loop or selection statement. An elegant solution for arming an exception object is the following macro, defined in *Exception.d*:

```
#define catch(e)    setjmp(catchException(e))
```

catch() is used where **setjmp()** would normally be specified, i.e., the restrictions imposed by ANSI-C on **setjmp()** can be observed for **catch()**. It will later return the value handed to **cause()**. The trick lies in calling the method **catchException()** to supply the argument for **setjmp()**:

```

% catchException {
%casts
    self -> armed = 1;
    return self -> label;
}

```

catchException() simply sets **.armed** and returns the **jmp_buf** so that **setjmp()** can initialize it. According to the ANSI-C standard, **jmp_buf** is an array type, i.e., the name of a **jmp_buf** represents its address. If a C system erroneously defined **jmp_buf** as a structure, we would simply have to return its address explicitly. We do not require **catch()** to be applied to the top element on the exception stack.

13.3 Examples

In our calculator we can replace the explicit **setjmp()** in *parse.c* with an exception object:

```
int main (void)
{
    volatile int errors = 0;
    char buf [BUFSIZ];
    void * retry = new(Exception());
    ...

    if (catch(retry))
    {
        ++ errors;
        reclaim(Node(), delete);
    }

    while (gets(buf))
        ...
}
```

Causing an exception will now restart the main loop. **error()** is modified so that it ends by causing an exception:

```
void error (const char * fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    fprintf(stderr, fmt, ap), putc('\n', stderr);
    va_end(ap);
    cause(1);
    assert(0);
}
```

error() is called for any error discovered in the calculator. It prints an error message and causes an exception which will normally directly restart the main loop. However, while **Symtab** executes its **load** method, it nests its own exception handler:

```
% Symtab load {
    FILE * fp;
    void * in;
    void * cleanup;
    ...
    if (! (fp = fopen(fnm, "r")))
        return EOF;

    cleanup = new(Exception());
    if (catch(cleanup))
    {
        fclose(fp);
        delete(cleanup);
        cause(1);
        assert(0);
    }

    while (in = retrieve(fp))
        ...

    fclose(fp);
    delete(cleanup);
    return result;
}
```

We saw in section 12.5 that we have a problem if **load()** is working on an expression and if **getsymbol()** cannot attach a name to the appropriate symbol in **table**:

```
else if (lex(result) != token)
    error("%s: need a %s, got a %s",
        buf, nameOf(class), nameOf(classOf(result)));
```

All it takes now is to call **error()** to print a message. **error()** causes an exception which is at this point caught through the exception object **cleanup** in **load()**. In this handler it is known that the stream **fp** must be closed before **load()** can be terminated. When **cleanup** is deleted and another exception is caused, control finally reaches the normal exception handler established with **retry** in **main()** where superfluous nodes are reclaimed and the main loop is restarted.

This example demonstrates it is best to design **cause()** as a function which only passes an exception code. **error()** can be called from different protected contexts and it will automatically return to the appropriate exception handler. By deleting the corresponding exception object and calling **cause()** again, we can trigger all handlers up the chain.

Exception handling smells of **goto** with all its unharnessed glory. The following, gruesome example uses a **switch** and two exception objects to produce the output

```
$ except
caused -1
caused 1
caused 2
caused 3
caused 4
```

Here is the code; extra points are awarded if you trace it with a pencil...

```
int main ()
{
    void * a = new(Exception()), * b = new(Exception());

    cause(-1); puts("caused -1");

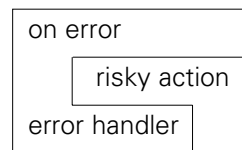
    switch (catch(a)) {
    case 0:
        switch (catch(b)) {
        case 0:
            cause(1); assert(0);
        case 1:
            puts("caused 1");
            cause(2); assert(0);
        case 2:
            puts("caused 2");
            delete(b);
            cause(3); assert(0);
        default:
            assert(0);
        }
    }
```

```

    case 3:
        puts("caused 3");
        delete(a);
        cause(4);
        break;
    default:
        assert(0);
    }
    puts("caused 4");
    return 0;
}

```

This code is certainly horrible and incomprehensible. However, if exception handling is used to construct the package shown at the beginning of this chapter



we still maintain a resemblance of the *one entry, one exit* paradigm for control structures which is at the heart of structured programming. The **Exception** class provides a clean, encapsulated mechanism to nest exception handlers and thus to nest one protected risky action inside another.

13.4 Summary

Modern programming languages like Eiffel or C++ support special syntax for exception handling. Before a risky action is attempted, an exception handler is established. During the risky action, software or hardware (interrupts, signals) can cause the exception and thus start execution of the exception handler. Theoretically, upon completion of the exception handler there are three choices: *terminate* both, the exception handler and the risky action; *resume* the risky action immediately following the point where the exception was caused; or *retry* that part of the risky action which caused the exception.

In practice, the most likely choice is termination and that may be the only choice which a programming language supports. However, a language should definitely support nesting the ranges where an exception handler is effective, and it must be possible to chain exception handling, i.e., when one exception handler terminates, it should be possible to invoke the next outer handler.

Exception handling with termination can easily be implemented in ANSI-C with **setjmp()**. Exception handlers can be nested by stacking the **jmp_buf** information set up by **setjmp()** and used by **longjmp()**. A stack of **jmp_buf** values can be managed as objects of an **Exception** class. Objects are created to nest exception handlers, and they must be deleted in the opposite order. An exception object is armed with **catch()**, which will return a second time with the nonzero exception code. An exception is caused by calling **cause()** with the exception code that should be delivered to **catch()** for the newest exception object.

13.5 Exercises

It seems likely that one could easily forget to delete some nested exceptions. Therefore, **Exception_dtor()** could implicitly pop exceptions from the stack until it finds **self**. Is it a good idea to **delete()** them to avoid memory leaks? What should happen if **self** cannot be found?

Similarly, **catch()** could search the stack for the nearest armed exception. Should it pop the unarmed ones?

setjmp() is a dangerous feature, because it does not guard against attempting to return into a function that has itself returned. Normally, ANSI-C uses an *activation record stack* to allocate local variables for each active function invocation. Obviously, **cause()** must be called at a higher level on that stack than the function it tries to **longjmp()** into. If **catchException()** passes the address of a local variable of its caller, we could store it with the **jmp_buf** and use it as a coarse verification of the legality of the **longjmp()**. A fancier technique would be to store a magic number in the local variable and check if it is still there. As a nonportable solution, we might be able to follow a chain along the activation record stack and check from **cause()** if the stack frame of the caller of **catchException()** is still there.

14

Forwarding Messages

A GUI Calculator

In this chapter we look at a rather typical problem: one object hierarchy we build ourselves to create an application, and another object hierarchy is more or less imposed upon us, because it deals with system facilities such as a graphical user interface (GUI, the pronunciation indicates the generally charming qualities). At this point, real programmers turn to multiple inheritance, but, as our example of the obligatory moused calculator demonstrates, an elegant solution can be had at a fraction of the cost.

14.1 The Idea

Every dynamically linked method is called through its selector, and we saw in chapter 8 that the selector checks if its method can be found for the object. As an example, consider the selector **add()** for the method to add an object to a **List**:

```
struct Object * add (void * _self, const void * element) {
    struct Object * result;
    const struct ListClass * class =
        cast(ListClass(), classOf(_self));

    assert(class -> add.method);
    cast(Object(), element);

    result = ((struct Object * (*) ())
              class -> add.method)(_self, element);
    return result;
}
```

classOf() tries to make sure that **_self** references an object; the surrounding call to **cast()** ascertains that the class description of **_self** belongs to the metaclass **ListClass**, i.e., that it really contains a pointer to an **add** method; finally, **assert()** guards against a null value masquerading as this pointer, i.e., it makes sure that an **add** method has been implemented somewhere up the inheritance chain.

What happens if **add()** is applied to an object that has never heard of this method, i.e., what happens if **_self** flunks the various tests in the **add()** selector? As it stands, an **assert()** gets triggered somewhere, the problem is contained, and our program quits.

Suppose we are working on a class **X** of objects which themselves are not descendants of **List** but which know some **List** object to which they could logically pass a request to **add()**. As it stands, it would be the responsibility of the user of **X** objects, to know (or to find out with **respondsTo()**) that **add()** cannot be applied to them and to reroute the call accordingly. However, consider the following, slightly revised selector:

```

struct Object * add (void * _self, const void * element) {
    struct Object * result;
    const struct ListClass * class =
        (const void *) classOf(_self);

    if (isOf(class, ListClass()) && class -> add.method) {
        cast(Object(), element);
        result = ((struct Object * (*) ())
            class -> add.method)(_self, element);
    } else
        forward(_self, & result, (Method) add, "add",
            _self, element);

    return result;
}

```

Now, **_self** can reference any object. If its class happens to contain a valid **add** pointer, the method is called as before. Otherwise, all the information is passed to a new method **forward()**: the object itself; an area for the expected result; a pointer to and the name of the selector which failed; and the values in the original argument list. **forward()** is itself a dynamically linked method declared in **Object**:

```

% Class Object {
    ...
%-
    void forward (const _self, void * result, \
        Method selector, const char * name, ...);

```

Obviously, the initial definition is a bit helpless:

```

% Object forward {
%casts
    fprintf(stderr, "%s at %p does not answer %s\n",
        nameOf(classOf(self)), self, name);
    assert(0);
}

```

If an **Object** itself does not understand a method, we are out of luck. However, **forward()** is dynamically linked: if a class wants to forward messages, it can accomplish this by redefining **forward()**. As we shall see in the example in section 14.6, this is almost as good as an object belonging to several classes at the same time.

14.2 Implementation

Fortunately, we decided in chapter 7 to enforce our coding standard with a preprocessor *ooc*, and selectors are a part of the coding standard. In section 8.4 we looked at the **selector** report which generates all selectors. Message forwarding is accomplished by declaring **forward()** as shown above, by defining a default implementation, and by modifying the **selector** report in *etc.rep* so that all generated selectors reroute what they do not understand:*

* As before, the presentation is simplified so that it does not show the parts which deal with variable argument lists.


```

`%header { `n
`%result
`%classOf

`%ifmethod
`%checks
`%call
`t } else `n
`%forward
`%return
} `n `n

```

This is almost the same code as in section 8.4: as we saw above, the **cast()** in the **classOf** report is turned into a call to **isOf()** as part of the **ifmethod** report and an **else** clause is added with the **forward** report to generate the call to **forward()**.

The call to **forward()** is routed through another selector for argument checking. It is probably not helpful to get stuck in recursion here, so if the selector **forward()** itself is generated, we stop things with an **assert()**:

```

% forward // forward the call, but don't forward forward
`{if `method forward
`t `t assert(0);
`} `else
`t `t forward(_self, \
    `{if `result void 0, `} `else & result, `} \
    (Method) `method , " `method ", `%args );
` } `n

```

The additional **{if** concerns the fact that a selector eventually has to return the result expected by its caller. This result will have to be produced by **forward()**. The general approach is to pass the result area to **forward()** to get it filled somehow. If, however, our selector returns **void**, we have no **result** variable. In this case we pass a null pointer.

It looks as if we could write slightly better code by hand: in some cases we could avoid the **result** variable, assignment, and a separate **return** statement. However, tuning would complicate the *ooc* reports unnecessarily because any reasonable compiler will generate the same machine code in either case.

classOf is the other report that gets modified significantly. A call to **cast()** is removed, but the interesting question is what happens if a call to a class method needs to be forwarded. Let us look at the selector which *ooc* generates for **new()**:

```

struct Object * new (const void * _self, ...) {
    struct Object * result;
    va_list ap;
    const struct Class * class = cast(Class(), _self);
    va_start(ap, _self);
    if (class -> new.method) {
        result = ((struct Object * (*) ()) class -> new.method)
            (_self, & ap);
    }
}

```

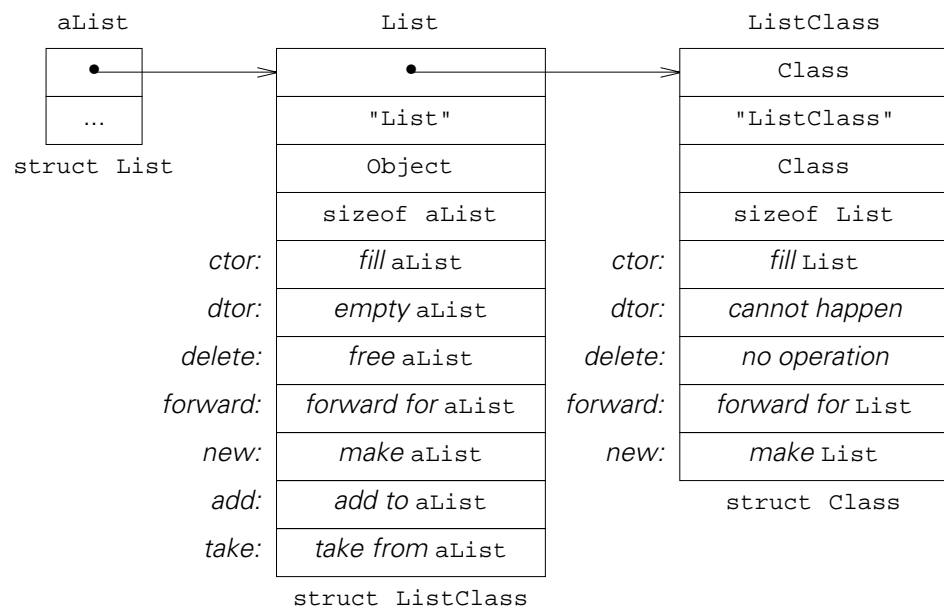
```

    } else
        forward((void *) _self, & result, (Method) new, "new",
                _self, & ap);
    va_end(ap);
    return result;
}

```

new() is called for a class description like **List**. Calling a class method for something other than a class description is probably a very bad idea; therefore, **cast()** is used to forbid this. **new** belongs into **Class**; therefore, no call to **isOf()** is needed.

Let's assume that we forgot to define the initial **Object_new()**, i.e., that **List** has not even inherited a **new** method, and that, therefore, **new.method** is a null pointer. In this case, **forward()** is applied to **List**. However, **forward()** is a dynamically linked method, not a class method. Therefore, **forward()** looks in the class description of **List** for a **forward** method, i.e., it tries to find **ListClass_forward()**:



This is perfectly reasonable: **List_forward()** is responsible for all messages which **aList** does not understand; **ListClass_forward()** is responsible for all those which **List** cannot handle. Here is the **classOf** report in *etc.rep*:

```

`{if `linkage %-
`{if `meta `metaroot
`t const struct `meta * class = classOf(_self); `n
`} `else
`t const struct `meta * class = ` \
    (const void *) classOf(_self); `n
`}

```

```

    } {else
    {t const struct `meta * class = ` \
                                cast( `metaroot (), _self); `n
    } `n

```

For dynamically linked methods **`linkage** is **%-**. In this case we get the class description as a **struct Class** from **classOf()**, but we cast it to the class description structure which it will be once **isOf()** has verified the type, so that we can select the appropriate method component.

For a class method, **`linkage** evaluates as **%+**, i.e., we advance to the second half of the report and simply check with **cast()** that **_self** is at least a **Class**. This is the only difference in a selector for a class method with forwarding.

14.3 Object-Oriented Design by Example

GUIs are everybody's favorite demonstration ground for the power of object-oriented approaches. A typical benchmark is to create a small calculator that can be operated with mouse clicks or from the keyboard:

display			C
7	8	9	+
4	5	6	-
1	2	3	*
Q	0	=	/

We will now build such a calculator for the *curses* and X11 screen managers. We use an object-oriented approach to design, implement, and test a general solution. Once it works, we connect it to two completely incompatible graphical environments. In due course, we shall see how elegantly message forwarding can be used.

It helps to get the application's algorithm working, before we head for the GUI library. It also helps to decompose the application's job into interacting objects. Therefore, let us just look at what objects we can identify in the calculator pictured above.

Our calculator has buttons, a computing chip, and a display. The display is an information sink: it receives something and displays it. The computing chip is an information filter: it receives something, changes its own state, and passes modified information on. A button is an information source or even a filter: if it is properly stimulated, it will send information on.

Thus far, we have identified at least four classes of objects: the display, the computing chip, buttons, and information passed between them. There may be a fifth kind of object, namely the source of the stimulus for a button, which models our keyboard, a mouse, etc.

There is a common aspect that fits some of these classes: a display, computing chip, or button may be wired to one next object, and the information is transmitted along this wire. An information sink like the display only receives information, but that does not hurt the general picture. So far, we get the following design:

CLASS	DATA	METHODS	
Object			base class
Event	kind data		information to pass type of data text, position, etc.
Ic	out	wire gate	base class for application object I am connected to connect me to another object send information to out
LineOut		wire gate	model the display not used display incoming information
Button	text	gate	model an input device label, defines interesting information look at incoming information: if it matches <code>text</code> , send it on
Calc	state	gate	computing chip current value, etc. change state based on information, pass new current value on, if any

This looks good enough for declaring the principal methods and trying to write a main program to test the decomposition of our problem world into classes.

lc.d

```
enum react { reject, accept };

% IcClass: Class Ic: Object {
    void * out;
%-
    void wire (Object @ to, _self);
    enum react gate (_self, const void * item);
%}

% IcClass LineOut: Ic {
%}

% IcClass Button: Ic {
    const char * text;
%}
```

run.c

```
int main ()
{
    void * calc = new(Calc());
    void * lineOut = new(LineOut());
    void * mux = new(Mux());
    static const char * const cmd [] = { "C", "Q",
        "0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
        "+", "-", "*", "/", "=", 0 };
    const char * const * cpp;

    wire(lineOut, calc);
    for (cpp = cmd; * cpp; ++ cpp)
    {
        void * button = new(Button(), * cpp);

        wire(calc, button), wire(button, mux);
    }
}
```

Close. We can set up a computing chip, a display, and any number of buttons, and connect them. However, if we want to test this setup with character input from a keyboard, we will have to wrap every character as an **Event** and offer it to each **Button** until one is interested and returns **accept** when we call its **gate()** method. One more class would help:

CLASS	DATA	METHODS
Object		base class
Ic		base class for application
Mux		multiplexer, one input to many outputs
	list	List of objects
		wire connects me to another object
		gate passes information until some object wants it

The main program shown above already uses a **Mux** object and connects it to each **Button**. We are ready for the main loop:

```
while ((ch = getchar()) != EOF)
    if (! isspace(ch))
    {
        static char buf [2];
        void * event;

        buf[0] = ch;
        gate(mux, event = new(Event(), 0, buf));
        delete(event);
    }
return 0;
}
```

White space is ignored. Every other character is wrapped as an **Event** with **kind** zero and the character as a string. The event is handed to the multiplexer and the computation takes its course.

Summary

This design was motivated by the *Class-Responsibility-Collaborator* technique described in [Bud91]: We identify objects more or less by looking at the problem. An object is given certain responsibilities to carry out. This leads to other objects which collaborate in carrying out the work. Once the objects are known, they are collected into classes and, hopefully, a hierarchical pattern can be found that is deeper than just one level.

The key idea for this application is the **lc** class with the capability of receiving, changing, and routing information. This idea was inspired by the *Interface Builder* of NeXTSTEP where much of the information flow, even to application-specific classes, can be “wired” by mouse-dragging when the graphical user interface is designed with a graphics editor.

14.4 Implementation — lc

Obviously, **gate()** is a dynamically bound method, because the subclasses of **lc** use it to receive and process information. **lc** itself owns **out**, the pointer to another object to which information must be sent. **lc** itself is mostly an abstract base class, but **lc_gate()** can access **out** and actually pass information on:

```
% lc gate {
%casts
    return self -> out ? gate(self -> out, item) : reject;
}
```

This is motivated by information hiding: if a subclass’ **gate** method wants to send information on, it can simply call **super_gate()**.

wire() is trivial: it connects an **lc** to another object by storing the object address in **out**:

```
% lc wire {
%casts
    self -> out = to;
}
```

Once the multiplexer class **Mux** is invented, we realize that **wire()**, too, must be dynamically linked. **Mux** overwrites **wire()** to add its target object to a **List**:

```
% Mux wire {                // add another receiver
%casts
    addLast(self -> list, to);
}
```

Mux_gate() can be defined in various ways. Generally, it has to offer the incoming information to some target objects; however, we can still decide in which order we do this and whether or not we want to quit once an object accepts the information — there is room for subclasses!

```

% Mux gate {                // sends to first responder
    unsigned i, n;
    enum react result = reject;
%casts
    n = count(self -> list);
    for (i = 0; i < n; ++ i)
    {
        result = gate(lookAt(self -> list, i), item);
        if (result == accept)
            break;
    }
    return result;
}

```

This solution proceeds sequentially through the list in the order in which it was created, and it quits as soon as one invocation of **gate()** returns **accept**.

LineOut is needed so that we can test a computing chip without connecting it to a graphical user interface. **gate()** has been defined leniently enough so that **LineOut_gate()** is little more than a call to **puts()**:

```

% LineOut gate {
%casts
    assert(item);
    puts(item);    // hopefully, it is a string
    return accept;
}

```

Of course, **LineOut** would be more robust, if we used a **String** object as input.

The classes built thus far can actually be tested. The following example *hello* connects an **Ic** to a **Mux** and from there first to two more **Ic** objects and then twice to a **LineOut**. Finally, a string is sent to the first **Ic**:

```

int main ()
{
    void * ic = new(Ic());
    void * mux = new(Mux());
    int i;
    void * lineOut = new(LineOut());

    for (i = 0; i < 2; ++ i)
        wire(new(Ic()), mux);
    wire(lineOut, mux);
    wire(lineOut, mux);
    wire(mux, ic);
    puto(ic, stdout);
    gate(ic, "hello, world");
    return 0;
}

```

The output shows the connections described by **puto()** and the string displayed by the **LineOut**:

```

$ hello
Ic at 0x182cc
wired to Mux at 0x18354
wired to [nil]
list List at 0x18440
    dim 4, count 4 {
        Ic at 0x184f0
        wired to [nil]
        Ic at 0x18500
        wired to [nil]
        LineOut at 0x184e0
        wired to [nil]
        LineOut at 0x184e0
        wired to [nil]
    }
hello, world

```

Although the **Mux** object is connected to the **LineOut** object twice, **hello, world** is output only once, because the **Mux** object passes its information only until some **gate()** returns **accept**.

Before we can implement **Button** we have to make a few assumptions about the **Event** class. An **Event** object contains the information that is normally sent from one **lc** to another. Information from the keyboard can be represented as a string, but a mouse click or a cursor position looks different. Therefore, we let an **Event** contain a number **kind** to characterize the information and a pointer **data** which hides the actual values:

```

% Event ctor { // new(Event(), 0, "text") etc.
    struct Event * self = super_ctor(Event(), _self, app);

    self -> kind = va_arg(* app, int);
    self -> data = va_arg(* app, void *);
    return self;
}

```

Now we are ready to design a **Button** as an information filter: if the incoming **Event** is a string, it must match the button's text; any other information is accepted sight unseen, as it should have been checked elsewhere already. If the **Event** is accepted, **Button** will send its text on:

```

% Button ctor { // new(Button(), "text")
    struct Button * self = super_ctor(Button(), _self, app);

    self -> text = va_arg(* app, const char *);
    return self;
}

% Button gate {
%casts
    if (item && kind(item) == 0
        && strcmp(data(item), self -> text))
        return reject;
    return super_gate(Button(), self, self -> text);
}

```


This, too, can be checked with a small test program *button* in which a **Button** is wired to a **LineOut**:

```
int main ()
{
    void * button, * lineOut;
    char buf [100];

    lineOut = new(LineOut());
    button = new(Button(), "a");
    wire(lineOut, button);
    while (gets(buf))
    {
        void * event = new(Event(), 0, buf);
        if (gate(button, event) == accept)
            break;
        delete(event);
    }
    return 0;
}
```

button ignores all input lines until a line contains the **a** which is the text of the button:

```
$ button
ignore
a
a
```

Only one **a** is input, the other one is printed by the **LineOut**.

LineOut and **Button** were implemented mostly to check the computing chip before it is connected to a graphical interface. The computing chip **Calc** can be as complicated as we wish, but for starters we stick with a very primitive design: digits are assembled into a value in the display; the arithmetic operators are executed as soon as possible without precedence; **=** produces a total; **C** clears the current value; and **Q** terminates the program by calling **exit(0)**.

This algorithm can be executed by a finite state machine. A practical approach uses a variable **state** as an index selecting one of two values, and a variable **op** to remember the current operator which will be applied after the next value is complete:

```
%prot
typedef int values[2];          // left and right operand stack
% IcClass Calc: Ic {
    values value;               // left and right operand
    int op;                     // operator
    int state;                  // FSM state
%}
```

The following table summarizes the algorithm that has to be coded:

input	state	value[]	op	super_gate()
<i>digit</i>	<i>any</i>	$v[any] *= 10$ $v[any] += digit$		value[any]
+ - * /	0 → 1	$v[1] = 0$	input	
	1	$v[0] op= v[1]$ $v[1] = 0$	input	value[0]
=	0	$v[0] = 0$		
	1 → 0	$v[0] op= v[1]$ $v[0] = 0$		value[0]
C	<i>any</i>	$v[any] = 0$		value[any]

And it really works:

```
$ run
12 + 34 * 56 = Q
1
12
3
34
46
5
56
2576
```

Summary

The **lc** classes are very simple to implement. A trivial **LineOut** and an input loop, which reads from the keyboard, enable us to check **Calc** before it is inserted into a complicated interface.

Calc communicates with its input buttons and output display by calling **gate()**. This is coupled loosely enough so that **Calc** can send fewer (or even more) messages to the display than it receives itself.

Calc operates very strictly bottom-up, i.e., it reacts to every input passed in through **Calc_gate()**. Unfortunately, this rules out the recursive descent technique introduced in the third chapter, or other syntax-driven mechanisms such as *yacc* grammars, but this is a characteristic of the message-based design. Recursive descent and similar mechanisms start from the main program down, and they decide when they want to look at input. In contradistinction, message-driven applications use the main program as a loop to gather events, and the objects must react to these events as they are sent in.

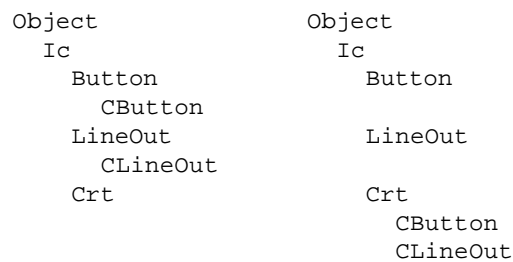
If we insist on a top-down approach for **Calc**, we must give it its own thread of control, i.e., it must be a coroutine, a thread under Mach and similar systems, or even another process under UNIX, and the message paradigm must be subverted by process communication.

14.5 A Character-Based Interface — *curses*

curses is an ancient library, which manages character-based terminals and hides the idiosyncracies of various terminals by using the *termcap* or *terminfo* databases [Sch90]. Originally, Ken Arnold at Berkeley extracted the functions from Bill Joy’s *vi* editor. Meanwhile, there are several, optimized implementations, even for DOS; some are in the public domain.

If we want to connect our calculator to *curses*, we have to implement replacements for **LineOut** and **Button** and connect them with a **Calc** object. *curses* provides a **WINDOW** data type, and it turns out that our best bet is to use a **WINDOW** for each graphical object. The original version of *curses* does not use a mouse or even provide functions to move a cursor under control of something like arrow keys. Therefore, we will have to implement another object that runs the cursor and sends cursor positions as events to the buttons.

It looks like we have two choices. We can define a subclass of **Ic** to manage the cursor and the main loop of the application, and subclasses of **Button** and of **LineOut** to provide the graphical objects. Every one of these three classes will own its own **WINDOW**. Alternatively, as shown at right below, we can start a new hierarchy with a subclass of **Ic** which owns a **WINDOW** and can run the cursor. Additionally we create two more subclasses which then may have to own a **Button** and a **LineOut**, respectively.



Neither solution looks quite right. The first one seems perhaps closer to our application, but we don’t encapsulate the existence of the **WINDOW** data type in a single class, and it does not look like we package *curses* in a way that can be reused for the next project. The second solution seems to encapsulate *curses* mostly in **Crt**; however, the subclasses need to contain objects that are very close to the application, i.e., once again we are likely to end up with a once-only solution.

Let us stick with the second approach. We will see in the next section how we can produce a better design with message forwarding. Here are the new classes:

CLASS	DATA	METHODS
Object		base class
Ic		base class for application
Crt		base class for screen management
	window	curses WINDOW
	rows, cols	size
		makeWindow create my window
		addStr display a string in my window
		crtBox run a frame around my window
		gate run cursor, send text or positions
CLineOut		output window
		gate display text in my window
CButton		box with label to click
	button	a Button to accept and forward events
	x, y	my position
		gate if text event, send to button
		if position event matches,
		send null pointer to button

Implementing these classes requires a certain familiarity with *curses*; therefore, we will not look at the details of the code here. The *curses* package has to be initialized; this is taken care of by a hook in **Crt_ctor()**: the necessary *curses* functions are called when the first **Crt** object is created.

Crt_gate() contains the main loop of the program. It ignores its incoming event and it reads from the keyboard until it sees a *control-D* as an end of input indication. At this point it will return **reject** and the main program terminates.

A few input characters are used to control the cursor. If *return* is pressed, **Crt_gate()** calls **super_gate()** to send out an event with **kind** one and with an integer array with the current row and column position of the cursor. All other characters are sent out as events with **kind** zero and the character as a string.

The interesting class is **CButton**. When an object is constructed, a box appears on the screen with the button name inside.

```
% CButton ctor {      // new(CButton(), "text", row, col)
    struct CButton * self = super_ctor(CButton(), _self, app);
    self -> button =
        new(Button(), va_arg(* app, const char *));
    self -> y = va_arg(* app, int);
    self -> x = va_arg(* app, int);
    makeWindow(self, 3, strlen(text(self -> button)) + 4,
                self -> y, self -> x);
    addStr(self, 1, 2, text(self -> button));
    crtBox(self);
    return self;
}
```

The window is created large enough for the text surrounded by spaces and a frame. **wire()** must be overwritten so that the internal button gets connected:

```
% CButton wire {
%casts
    wire(to, self -> button);
}
```

Finally, **CButton_gate()** passes text events directly on to the internal button. For a position event we check if the cursor is within our own box:

```
% CButton gate {
%casts
    if (kind(item) == 1)          // kind == 1 is click event
    {   int * v = data(item);     // data is an array [x, y]

        if (v[0] >= self -> x && v[0] < self -> x + cols(self)
            && v[1] >= self -> y && v[1] < self -> y + rows(self))
            return gate(self -> button, 0);
        return reject;
    }
    return gate(self -> button, item);
}
```

If so, we send a null pointer to the internal button which responds by sending on its own text.

Once again, we can check the new classes with a simple program *cbutton* before we wire up the entire calculator application.

```
int main ()
{   void * crt = new(Crt());
    void * lineOut = new(CLineOut(), 5, 10, 40);
    void * button = new(CButton(), "a", 10, 40);

    makeWindow(crt, 0, 0, 0, 0);    /* total screen */
    gate(lineOut, "hello, world");

    wire(lineOut, button), wire(button, crt);
    gate(crt, 0);                   /* main loop */

    return 0;
}
```

This program displays **hello, world** on the fifth line and a small button with the label **a** near the middle of our terminal screen. If we move the cursor into the button and press *return*, or if we press **a**, the display will change and show the **a**. *cbutton* ends if interrupted or if we input *control-D*.

Once this works, our calculator will, too. It just has more buttons and a computing chip:

```
int main ()
{   void * calc = new(Calc());
    void * crt = new(Crt());
    void * lineOut = new(CLineOut(), 1, 1, 12);
    void * mux = new(Mux());
```

```

static const struct tbl { const char * nm; int y, x; }
tbl [] = {
    "C", 0, 15,
    "1", 3, 0, "2", 3, 5, "3", 3, 10, "+", 3, 15,
    "4", 6, 0, "5", 6, 5, "6", 6, 10, "-", 6, 15,
    "7", 9, 0, "8", 9, 5, "9", 9, 10, "*", 9, 15,
    "Q", 12, 0, "0", 12, 5, "=", 12, 10, "/", 12, 15,
    0 };
const struct tbl * tp;

makeWindow(crt, 0, 0, 0, 0);
wire(lineOut, calc);
wire(mux, crt);

for (tp = tbl; tp -> nm; ++ tp)
{ void * o = new(CButton(), tp -> nm, tp -> y, tp -> x);
  wire(calc, o), wire(o, mux);
}

gate(crt, 0);
return 0;
}

```

The solution is quite similar to the last one. A **CButton** object needs coordinates; therefore, we extend the table from which the buttons are created. We add a **Crt** object, connect it to the multiplexer, and let it run the main loop.

Summary

It should not come as a great surprise that we reused the **Calc** class. That is the least we can expect, no matter what design technique we use for the application. However, we also reused **Button**, and the **lc** base class helped us to concentrate totally on coping with *curses* rather than with adapting the computing chip to a different variety of inputs.

The glitch lies in the fact, that we have no clean separation between *curses* and the **lc** class. Our class hierarchy forces us to compromise and (more or less) use two **lc** objects in a **CButton**. If the next project does not use the **lc** class, we cannot reuse the code developed to hide the details of *curses*.

14.6 A Graphical Interface — X11

The X Window System (X11) is the *de facto* standard for graphical user interfaces on UNIX and other systems.* X11 controls a terminal with a bitmap screen, a mouse, and a keyboard and provides input and output facilities. *Xlib* is a library of functions implementing a communication protocol between an application program and the X11 server controlling the terminal.

* The standard source for information about X11 programming is the *X Window System* series published by O'Reilly and Associates. Background material for the current section can be found in volume 4, manual pages are in volume 5, ISBN 0-937175-58-7.

X11 programming is quite difficult because application programs are expected to behave responsibly in sharing the facilities of the server. Therefore, there is the *X toolkit*, a small class hierarchy which mostly serves as a foundation for libraries of graphical objects. The toolkit is implemented in C. Toolkit objects are called *wid-gets*.

The base class of the toolkit is **Object**, i.e., we will have to fix our code to avoid that name. Another important class in the toolkit is **ApplicationShell**: a widget from this class usually provides the framework for a program using the X11 server.

The toolkit itself does not contain classes with widgets that are visible on the screen. However, *Xaw*, the *Athena Widgets*, are a generally available, primitive extension of the toolkit class hierarchy which provides enough functionality to demonstrate our calculator.

The widgets of an application program are arranged in a tree. The root of this tree is an **ApplicationShell** object. If we work with *Xaw*, a **Box** or **Form** widget is next, because it is able to control further widgets within its screen area. For our calculator display we can use a **Label** widget from *Xaw*, and a button can be implemented with a **Command** widget.

On the screen, the **Command** widget appears as a frame with a text in it. If the mouse enters or leaves the frame, it changes its visual appearance. If a mouse button is clicked inside the frame, the widget will invoke a *callback function* which must have been registered previously.

So-called *translation tables* connect events such as mouse clicks and key presses on the keyboard to so-called *action functions* connected with the widget at which the mouse currently points. **Command** has action functions which change its visual appearance and cause the callback function to be invoked. These actions are used in the **Command** translation table to implement the reaction to a mouse click. The translation tables of a particular widget can be changed or augmented, i.e., we can decide that the key press **0** influences a particular **Command** widget as if a mouse button had been clicked inside its frame.

So-called *accelerators* are essentially redirections of translation tables from one widget to another. Effectively, if the mouse is inside a **Box** widget, and if we press a key such as **+**, we can redirect this key press from the **Box** widget to some **Command** widget inside the box, and recognize it as if a mouse button had been clicked inside the **Command** widget itself.

To summarize: we will need an **ApplicationShell** widget as a framework; a **Box** or **Form** widget to contain other widgets; a **Label** widget as a display; several **Command** widgets with suitable callback functions for our buttons; and certain magical convolutions permit us to arrange for key presses to cause the same effects as mouse clicks on specific **Command** widgets.

The standard approach is to create classes of our own to communicate with the classes of the toolkit hierarchy. Such classes are usually referred to as *wrappers* for a foreign hierarchy or system. Obviously, the wrappers should be as independent of any other considerations as possible, so that we can use them in arbitrary

toolkit projects. In general, we should wrap everything in the toolkit; but, to keep this book within reason, here is the minimal set for the calculator:

CLASS	DATA	METHODS
Objct		our base class (renamed)
Xt		base class for X toolkit wrappers
	widget	my X toolkit widget
		makeWidget create my widget
		addAllAccelerators
		setLabel change label resource
		addCallback add callback function
		(widget may or may not change)
XtApplicationShell		framework
		mainLoop X11 event loop
XawBox		wraps Athena's Box
XawForm		wraps Athena's Form
XawLabel		wraps Athena's Label
XawCommand		wraps Athena's Command

These classes are very simple to implement. They exist mostly to hide the uglier X11 and toolkit incantation from our own applications. There are some design choices. For example, **setLabel()** could be defined for **XawLabel** rather than for **Xt**, because a new label is only meaningful for **XawLabel** and **XawCommand** but not for **ApplicationShell**, **Box**, or **Form**. However, by defining **setLabel()** for **Xt** we model how the toolkit works: widgets are controlled by so-called *resources* which can be supplied during creation or later by calling **XtSetValues()**. It is up to the widget if it knows a particular resource and decides to act when it receives a new value for it. Being able to send the value is a property of the toolkit as such, not of a particular widget.

Given this foundation, we need only two more kinds of objects: an **XLineOut** receives a string and displays it and an **XButton** sends out text events for mouse clicks or keyboard characters. **XLineOut** is a subclass of **XawLabel** which behaves like a **LineOut**, i.e., which must do something about **gate()**.

Xt.d

```
% Class XLineOut: XawLabel {
%}
```

Xt.dc

```
% XLineOut ctor { // new(XLineOut(), parent, "name", "text")
    struct XLineOut * self =
                                super_ctor(XLineOut(), _self, app);
    const char * text = va_arg(* app, const char *);
    gate(self, text);
    return self;
}
```


Three arguments must be specified when an **XLineOut** is created: the superclass constructor needs a parent **Xt** object which supplies the parent widget for the application’s widget hierarchy; the new widget should be given a name for the qualification of resource names in an application default file; and, finally, the new **XLineOut** is initially set to some text which may imply its size on the screen. The constructor is brave and simply uses **gate()** to send this initial text to itself.

Because **XLineOut** does not descend from **lc**, it cannot respond directly to **gate()**. However, the selector will forward the call; therefore, we overwrite **forward()** to do the work that would be done by a **gate** method if **XLineOut** could have one:

```
% XLineOut forward {
%casts
    if (selector == (Method) gate)
    {   va_arg(* app, void *);
        setLabel((void *) self, va_arg(* app, const void *));
        * (enum react *) result = accept;
    }
    else
        super_forward(XLineOut(), _self, result,
                        selector, name, app);
}
```

We cannot be sure that every call to **XLineOut_forward()** is a **gate()** in disguise. Every **forward** method should check and only respond to expected calls. The others can, of course, be forwarded up along the inheritance chain with **super_forward()**.

Just as for **new()**, **forward()** is declared with a variable argument list; however, the selector can only pass an initialized **va_list** value to the actual method, and the superclass selector must receive such a pointer. To simplify argument list sharing, specifically in deference to the metaclass constructor, *ooc* generates code to pass a pointer to an argument list pointer, i.e., the parameter **va_list * app**.

As a trivial example for forwarding the **gate()** message to an **XLineOut**, here is the *xhello* test program:

```
void main (int argc, char * argv [])
{   void * shell = new(XtApplicationShell(), & argc, argv);
    void * lineOut = new(XLineOut(), shell, 0, "hello, world");

    mainLoop(shell);
}
```

The program displays a window with the text **hello, world** and waits to be killed with a signal. Here, we have not given the widget in the **XLineOut** object an explicit name, because we are not specifying any resources.

XButton is a subclass of **XawCommand** so that we can install a callback function to receive mouse clicks and key presses:

Xt.d

```
% Class XButton: XawCommand {
    void * button;
%}
```

Xt.dc

```
% XButton ctor { // new(XButton(), parent, "name", "text")
    struct XButton * self = super_ctor(XButton(), _self, app);
    const char * text = va_arg(* app, const char *);

    self -> button = new(Button(), text);
    setLabel(self, text);
    addCallback(self, tell, self -> button);
    return self;
}
```

XButton has the same construction arguments as **XLineOut**: a parent **Xt** object, a widget name, and the text to appear on the button. The widget name may well be different from the text, e.g., the operators for our calculator are unsuitable as components in resource path names.

The interesting part is the callback function. We let an **XButton** own a **Button** and arrange for the callback function **tell()** to send a null pointer with **gate()** to it:

```
static void tell (Widget w, XtPointer client_data,
                  XtPointer call_data)
{
    gate(client_data, NULL);
}
```

client_data is registered with the callback function for the widget to pass it in later. We use it to designate the target of **gate()**.

We could actually avoid the internal button, because we could set up **XButton** itself to be wired to some other object; **client_data** could point to a pointer to this target and a pointer to the text, and then **tell()** could send the text directly to the target. It is simpler, however, to reuse the functionality of **Button**, especially, because it opens up the possibility for **XButton** to receive text via a forwarded **gate()** and pass it to the internal **Button** for filtering.

Message forwarding is, of course, the key to **XButton** as well: the internal button is inaccessible, but it needs to respond to a **wire()** that is originally directed at an **XButton**:

```
% XButton forward {
%casts
    if (selector == wire)
        wire(va_arg(* app, void *), self -> button);
    else
        super_forward(XButton(), _self, result,
                       selector, name, app);
}
```

Comparing the two **forward** methods we notice that **forward()** receives **self** with the **const** qualifier but circumvents it in the case of **XLineOut_forward()**. The basic idea is that the implementor of **gate()** must know if this is safe.

Once again, we can test **XButton** with a simple program *xbutton*. This program places an **XLineOut** and an **XButton** into an **XawBox** and another pair into an **XawForm**. Both containers are packed into another **XawBox**:

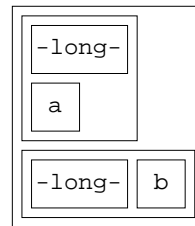
```
void main (int argc, char * argv [])
{
    void * shell = new(XtApplicationShell(), & argc, argv);
    void * box = new(XawBox(), shell, 0);
    void * composite = new(XawBox(), box, 0);
    void * lineOut = new(XLineOut(), composite, 0, "-long-");
    void * button = new(XButton(), composite, 0, "a");

    wire(lineOut, button);
    puto(button, stdout); /* Box will move its children */

    composite = new(XawForm(), box, "form");
    lineOut = new(XLineOut(), composite, "lineOut", "-long-");
    button = new(XButton(), composite, "button", "b");

    wire(lineOut, button);
    puto(button, stdout); /* Form won't move its children */
    mainLoop(shell);
}
```

The result appears approximately as follows on the screen:



Once the button **a** is pressed in the top half, the **XLineOut** receives and displays the text **a**. The Athena **Box** widget used as a container will resize the **Label** widget, i.e., the top box changes to display two squares, each with the text **a** inside. The button with text **b** is contained in an Athena **Form** widget, where the resource

```
*form.button.fromHoriz: lineOut
```

controls the placement. The **Form** widget maintains the appearance of the bottom rectangle even when **b** is pressed and the short text **b** appears inside the **XLineOut**.

The test program demonstrates that **XButton** can operate with mouse clicks and **wire()**; therefore, it is time to wire the calculator *xrun*:

```

void main (int argc, char * argv [])
{
    void * shell = new(XtApplicationShell(), & argc, argv);
    void * form = new(XawForm(), shell, "form");
    void * lineOut = new(XLineOut(), form, "lineOut",
                        ".....");

    void * calc = new(Calc());
    static const char * const cmd [] = { "C", "C",
        "1", "1", "2", "2", "3", "3", "a", "+",
        "4", "4", "5", "5", "6", "6", "s", "-",
        "7", "7", "8", "8", "9", "9", "m", "*",
        "Q", "Q", "0", "0", "t", "=", "d", "/", 0 };
    const char * const * cpp;

    wire(lineOut, calc);
    for (cpp = cmd; * cpp; cpp += 2)
    {
        void * button = new(XButton(), form, cpp[0], cpp[1]);
        wire(calc, button);
    }
    addAllAccelerators(form);
    mainLoop(shell);
}

```

This program is even simpler than the *curses* version, because the table only contains the name and text for each button. The arrangement of the buttons is handled by resources:

```

*form.C.fromHoriz:      lineOut
*form.1.fromVert:       lineOut
*form.2.fromVert:       lineOut
*form.3.fromVert:       lineOut
*form.a.fromVert:       C
*form.2.fromHoriz:      1
*form.3.fromHoriz:      2
*form.a.fromHoriz:      3
...

```

The resource file also contains the accelerators which are loaded by **addAllAccelerators()**:

```

*form.C.accelerators:   <KeyPress>c:      set() notify() unset()
*form.Q.accelerators:   <KeyPress>q:      set() notify() unset()
*form.0.accelerators:   :<KeyPress>0:    set() notify() unset()
...

```

If the resources are in a file *Xapp*, the calculator can, for example, be started with the following Bourne shell command:

```
$ XENVIRONMENT=Xapp xrun
```

14.7 Summary

In this chapter we have looked at an object-oriented design for a simple calculator with a graphical user interface. The CRC design technique summarized at the end of section 14.3 leads to some classes that can be reused unchanged for each of the three solutions.

The first solution tests the actual calculator without a graphical interface. Here, the encapsulation as a class permits an easy test setup. Once the calculator class is functional we can concentrate solely on the idiosyncrasies of the graphics libraries imposed upon us.

Both, *curses* and X11 require that we design some wrapper classes to merge the external library with our class hierarchy. The *curses* example demonstrates that without message forwarding we have to compromise: wrappers that are more likely reusable for the next project do not function too well in conjunction with an existing, application-oriented class hierarchy; wrappers that mesh well with our problem know too much about it to be generally reusable for dealing with *curses*.

The X11 solution shows the convenience of message forwarding. Wrappers just about completely hide the internals of X11 and the toolkit widgets. Problem-oriented classes like **XButton** combine the necessary functionality from the wrappers with the **lc** class developed for our calculator. Message forwarding lets classes like **XButton** function as if they were descendants of **lc**. In this example, message forwarding permits objects to act as if they belonged to two classes at the same time, but we do not incur the overhead and complexity of multiple inheritance as supported in C++.

Message forwarding is quite simple to implement. All that needs to be done is to modify the selector generation in the appropriate *ooc* reports to redirect non-understood selector calls to a new dynamically linked method **forward()** which classes like **XButton** overwrite to receive and possibly redirect forwarded messages.

14.8 Exercises

Obviously, wrapping *curses* into a suitable class hierarchy is an interesting exercise for character terminal aficionados. Similarly, our X11 calculator experiment can be redone with OSF/Motif or another toolkit.

Using accelerators is perhaps not the most natural way to map key presses into input to our calculators. One would probably think of action functions first. However, it turns out that while an action function knows the widget it applies to, it has no reasonable way to get from the widget to our wrapper. Either somebody recompiles the X toolkit with an extra pointer for user data in the **Object** instance record, or we have to subclass some toolkit widgets to provide just such a pointer. Given the pointer, however, we can create a powerful technology based on action functions and our **gate()**.

The idea to **gate()** and **wire()** was more or less lifted from NeXTSTEP. However, in NeXTSTEP a class can have more than one *outlet*, i.e., pointer to another object, and during wiring both, the actual outlet and the method to be used at the receiving end, can be specified.

Comparing sections 5.5 and 11.4, we can see that **Var** should really inherit from **Node** and **Symbol**. Using **forward()**, we could avoid **Val** and its subclasses.

Appendix A

ANSI-C Programming Hints

C was originally defined by Dennis Ritchie in the appendix of [K&R78]. The ANSI-C standard [ANSI] appeared about ten years later and introduced certain changes and extensions. The differences are summarized very concisely in appendix C of [K&R88]. Our style of object-oriented programming with ANSI-C relies on some of the extensions. As an aid to classic C programmers, this appendix explains those innovations in ANSI-C which are important for this book. The appendix is certainly not a definition of the ANSI-C programming language.

A.1 Names and Scope

ANSI-C specifies that names can have almost arbitrary length. Names starting with an underscore are reserved for libraries, i.e., they should not be used in application programs.

Globally defined names can be hidden in a translation unit, i.e., in a source file, by using **static**:

```
static int f (int x) { ... }    only visible in source file
int g;                          visible throughout the program
```

Array names are constant addresses which can be used to initialize pointers even if an array references itself:

```
struct table { struct table * tp; }
v [] = { v, v+1, v+2 };
```

It is not entirely clear how one would code a forward reference to an object which is still to be hidden in a source file. The following appears to be correct:

```
extern struct x object;          forward reference
f() { object = value; }          using the reference
static struct x object;          hidden definition
```

A.2 Functions

ANSI-C permits — but does not require — that the declaration of a function contains parameter declarations right inside the parameter list. If this is done, the function is declared together with the types of its parameters. Parameter names may be specified as part of the function declaration, but this has no bearing on the parameter names used in the function definition.

```
double sqrt ();                  classic version
double sqrt (double);            ANSI-C
double sqrt (double x);          ... with parameter names
int getpid (void);                no parameters, ANSI-C
```

If an ANSI-C function prototype has been introduced, an ANSI-C compiler will try to convert argument values into the types declared for the parameters.

Function definitions may use both variants:

<code>double sqrt (double arg)</code>	ANSI-C
<code>{ ... }</code>	
<code>double sqrt (arg)</code>	classic
<code>double arg;</code>	
<code>{ ... }</code>	

There are exact rules for the interplay between ANSI-C and classic prototypes and definitions; however, the rules are complicated and error-prone. It is best to stick with ANSI-C prototypes and definitions, only.

With the option **-Wall** the GNU-C compiler warns about calls to functions that have not been declared.

A.3 Generic Pointers — **void ***

Every pointer value can be assigned to a pointer variable with type **void *** and vice versa, except for **const** qualifiers. The assignments do not change the pointer value. Effectively, this turns off type checking in the compiler:

<code>int iv [] = { 1, 2, 3 };</code>	
<code>int * ip = iv;</code>	ok, same type
<code>void * vp = ip;</code>	ok, arbitrary to void *
<code>double * dp = vp;</code>	ok, void * to arbitrary

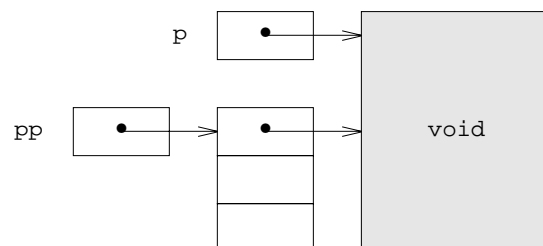
%p is used as a format specification for **printf()** and (theoretically) for **scanf()** to write and read pointer values. The corresponding argument type is **void *** and thus any pointer type:

<code>void * vp;</code>	
<code>printf("%p\n", vp);</code>	display value
<code>scanf("%p", & vp);</code>	read value

Arithmetic operations involving **void *** are not permitted:

<code>void * p, ** pp;</code>	
<code>p + 1</code>	wrong
<code>pp + 1</code>	ok, pointer to pointer

The following picture illustrates this situation:



A.4 **const**

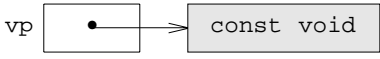
const is a qualifier indicating that the compiler should not permit an assignment. This is quite different from truly constant values. Initialization is allowed; **const** local variables may even be initialized with variable values:

```
int x = 10;
int f () { const int xsave = x; ... }
```

One can always use explicit typecast operations to circumvent the compiler checks:

```
const int cx = 10;
(int) cx = 20;           wrong
* (int *) & cx = 20;     not forbidden
```

These conversions are sometimes necessary when pointer values are assigned:

<code>const void * vp;</code>	
<code>int * ip;</code>	
<code>int * const p = ip;</code>	ok for local variable
<code>vp = ip;</code>	ok, blocks assignment
<code>ip = vp;</code>	wrong, allows assignment
<code>ip = (void *) vp;</code>	ok, brute force
<code>* (const int **) & ip = vp;</code>	ok, overkill
<code>p = ip;</code>	wrong, pointer is blocked
<code>* p = 10;</code>	ok, target is not blocked

const normally binds to the left; however, **const** may be specified before the type name in a declaration:

```
int const v [10];           ten constant elements
const int * const cp = v;   constant pointer to constant value
```

const is used to indicate that one does not want to change a value after initialization or from within a function:

```
char * strcpy (char * target, const char * source);
```

The compiler may place global objects into a write-protected segment if they have been completely protected with **const**. This means, for example, that the components of a structure inherit **const**:

```
const struct { int i; } c;
c.i = 10;           wrong
```

This precludes the dynamic initialization of the following pointer, too:

```
void * const String;
```

It is not clear if a function can produce a **const** result. ANSI-C does not permit this. GNU-C assumes that in this case the function does not cause any side effects and only looks at its arguments and neither at global variables nor at values behind pointers. Calls to this kind of a function can be removed during common subexpression elimination in the compilation process.

Because pointer values to **const** objects cannot be assigned to unprotected pointers, ANSI-C has a strange declaration for **bsearch()**:

```
void * bsearch (const void * key,
               const void * table, size_t nel, size_t width,
               int (* cmp) (const void * key, const void * elt));
```

table[] is imported with **const**, i.e., its elements cannot be modified and a constant table can be passed as an argument. However, the result of **bsearch()** points to a table element and does not protect it from modification.

As a rule of thumb, the parameters of a function should be pointers to **const** objects exactly if the objects will not be modified by way of the pointers. The same applies to pointer variables. The result of a function should (almost) never involve **const**.

A.5 typedef and const

typedef does not define macros. **const** may be bound in unexpected ways in the context of a **typedef**:

```
const struct Class { ... } * p;    protects contents of structure
typedef struct Class { ... } * ClassP;
const ClassP cp;                  contents open, pointer protected
```

How one would protect and pass the elements of a matrix remains a puzzle:

```
main ()
{
    typedef int matrix [10][20];
    matrix a;
    int b [10][20];

    int f (const matrix);
    int g (const int [10][20]);

    f(a);
    f(b);
    g(a);
    g(b);
}
```

There are compilers that do not permit any of the calls...

A.6 Structures

Structures collect components of different types. Structures, components, and variables may all have the same name:

```
struct u { int u; double v; } u;
struct v { double u; int v; } * vp;
```

Structure components are selected with a period for structure variables and with an arrow for pointers to structures:

```
u.u = vp -> v;
```

A pointer to a structure can be declared even if the structure itself has not yet been introduced. A structure may be declared without objects being declared:

```
struct w * wp;
struct w { ... };
```

A structure may contain a structure:

```
struct a { int x; };
struct b { ... struct a y; ... } b;
```

The complete sequence of component names is required for access:

```
b.y.x = 10;
```

The first component of a structure starts right at the beginning of the structure; therefore, structures can be lengthened or shortened:

```
struct a { int x; };
struct c { struct a a; ... } c, * cp = & c;
struct a * ap = & c.a;

assert((void *) ap == (void *) cp);
```

ANSI-C permits neither implicit conversions of pointers to different structures nor direct access to the components of an inner structure:

<code>ap = cp;</code>	wrong
<code>c.x, cp -> x</code>	wrong
<code>cp -> a.x</code>	ok, fully specified
<code>((struct a *) cp) -> x</code>	ok, explicit conversion

A.7 Pointers to Functions

The declaration of a pointer to a function is constructed from the declaration of a function by adding one level of indirection, i.e., a `*` operator, to the function name. Parentheses are used to control precedence:

<code>void * f (void *);</code>	function
<code>void * (* fp) (void *) = f;</code>	pointer to function

These pointers are usually initialized with function names that have been declared earlier. In function calls, function names and pointers are used identically:

<code>int x;</code>	
<code>f (& x);</code>	using a function name
<code>fp (& x);</code>	using a pointer, ANSI-C
<code>(* fp)(& x);</code>	using a pointer, classic

A pointer to a function can be a structure component:

```
struct Class { ...
    void * (* ctor) (void * self, va_list * app);
... } * cp, ** cpp;
```

In a function call, `->` has precedence over the function call, but it has no precedence over dereferencing with `*`, i.e., the parentheses are necessary in the second example:

```
cp -> ctor ( ... );
(* cpp) -> ctor ( ... );
```

A.8 Preprocessor

ANSI-C no longer expands **#define** recursively; therefore, function calls can be hidden or simplified by macros with the same name:

```
#define malloc(type) (type *) malloc(sizeof(type))

int * p = malloc(int);
```

If a macro is defined with parameters, ANSI-C only recognizes its invocation if the macro name appears before a left parenthesis; therefore, macro recognition can be suppressed in a function header by surrounding the function name with an extra set of parentheses:

```
#include <stdio.h>                                defines putchar(ch) as a macro

int (putchar) (int ch) { ... }    name is not replaced
```

Similarly, the definition of a parametrized macro no longer collides with a variable of the same name:

```
#define x(p) (((const struct Object *) (p)) -> x)

int x = 10;                                name is not replaced
```

A.9 Verification — *assert.h*

```
#include <assert.h>

assert( condition );
```

If **condition** is false, this macro call terminates the program with an appropriate error message.

The option **-DNDEBUG** can be specified to most C compilers to remove all calls to **assert()** during compilation. Therefore, **condition** should not contain side effects.

A.10 Global Jumps — *setjmp.h*

```
#include <setjmp.h>

jmp_buf onError;
int val;

if (val = setjmp(onError))
    error handling
else
    first call

...

longjmp(onError, val);
```

These functions are used to effect a global jump from one function back to another function which was called earlier and is still active. Upon the first call, **setjmp()** notes the situation in **jmp_buf** and returns zero. **longjmp()** later returns to this situation; then **setjmp()** returns whatever value was specified as second argument of **longjmp()**; if this value is zero, **setjmp()** will return one.

There are additional conditions: the context from which **setjmp()** was called must still be active; this context cannot be very complicated; variable values are not set back; jumping back to the point from which **longjmp()** was called is not possible; etc. However, recursion levels are handled properly.

A.11 Variable Argument Lists — *stdarg.h*

```
#include <stdarg.h>

void fatal (const char * fmt, ... )
{ va_list ap;
  int code;

  va_start(ap, fmt);           last explicit parameter name
  code = va_arg(ap, int);      next argument value
  vprintf(fmt, ap);
  va_end(ap);                 reinitialize
  exit(code);
}
```

If the parameter list in a function prototype and in a function definition ends with three periods, the function may be called with arbitrarily many arguments. The number of arguments specified in a particular call is not available; therefore, a parameter like the format of **printf()** or a specific trailing argument value must be used to determine when there are no more arguments.

The macros from *stdarg.h* are used to process the argument list. **va_list** is a type to define a variable for traversing the argument list. The variable is initialized by calling **va_start()**; the last explicitly specified parameter name must be used as an argument for initialization. **va_arg()** is a macro which produces the next argument value; it takes the type of this value as an argument. **va_end()** terminates processing of the argument list; following this call, the argument list may be traversed again.

Values of type **va_list** can be passed to other functions. In particular, there are versions of the **printf** functions which accept **va_list** values in place of the usual list of values to be converted.

The values in the variable part of the argument list are subject to the classic conversion rules: integral values are passed as **int** or **long**; floating point values are passed as **double**. The argument type specified as a second argument of **va_arg()** cannot be very complicated — if necessary, **typedef** can be used.

A.12 Data Types — *stddef.h*

stddef.h contains some data type declarations which may differ between platforms or compilers. The types specify the results of certain operations:

<code>size_t</code>	result of <code>sizeof</code>
<code>ptrdiff_t</code>	difference of two pointers

Additionally, there is a macro to compute the distance of a component from the start of a structure:

```
struct s { ... int a; ... };
offsetof(struct s, a)           returns size_t value
```

A.13 Memory Management — *stdlib.h*

```
void * calloc (size_t nel, size_t len);
void * malloc (size_t size);
void * realloc (void * p, size_t size);
void free (void * p);
```

These functions are declared in *stdlib.h*. **calloc()** returns a zeroed memory region with **nel** elements of **len** bytes each. **malloc()** returns an uninitialized memory region with **size** bytes. **realloc()** accepts a memory region allocated by **calloc()** or **malloc()** and makes sure that **size** bytes are available; the area may be lengthened or shortened in place, or it may be copied to a new, larger region. **free()** releases a memory region allocated by the other function; a null pointer may now be used with impunity.

A.14 Memory Functions — *string.h*

In addition to the well-known string functions, *string.h* defines functions for manipulating memory regions with explicit lengths, in particular:

```
void * memcpy (void * to, const void * from, size_t len);
void * memmove (void * to, const void * from, size_t len);
void * memset (void * area, int value, size_t len);
```

memcpy() and **memmove()** both copy a region; source and target may overlap for **memmove()** but not for **memcpy()**. Finally, **memset()** initializes a region with a byte value.

Appendix B

The *ooc* Preprocessor

Hints on *awk* Programming

awk was originally delivered with the Seventh Edition of the UNIX system. Around 1985 the authors extended the language significantly and described the result in [AWK88]. Today, there is a POSIX standard emerging and the new language is available in various implementations, e.g., as *nawk* on System V; as *awk*, adapted from the same sources, with the MKS-Tools for MSDOS; and as *gawk* from the Free Software Foundation (GNU). This appendix assumes a basic knowledge of the (new) *awk* programming language and provides an overview of the implementation of the *ooc* preprocessor. The implementation uses several features of the POSIX standard, and it has been developed with *gawk*.

B.1 Architecture

ooc is implemented as a shell script to load and execute an *awk* program. The shell script facilitates passing *ooc* command arguments to the *awk* program and it permits storing the various modules in a central place.

The *awk* program collects a database of information about classes and methods from the class description files, and produces C code from the database for interface and representation files and for method headers, selectors, parameter import, and initialization in the implementation files. The *awk* program is based on two design concepts: modularisation and report generation.

A module contains a number of functions and a **BEGIN** clause defining the global data maintained by the functions. *awk* does not support information hiding, but the modules are kept in separate files to simplify maintenance. The *ooc* command script can use **AWKPATH** to locate the files in a central place.

All work is done under control of **BEGIN** clauses which *awk* will execute in order of appearance. Consequently, *main.awk* must be loaded last, because it processes the *ooc* command line.

Pattern clauses are not used. They cannot be used for all files anyway, because *ooc* consults for each class description all class description files leading up to it. The algorithm to read lines, remove comments, and glue continuation lines together is implemented in a single function **get()** in *io.awk*. If pattern clauses were used, the same algorithm would have to be replicated in pattern clauses.

The database can be inspected if certain debugging modules are loaded as part of the *awk* program. These debugging modules use pattern clauses for control, i.e., debugging starts once the command line processing of *ooc* has been completed. Debugging statements are entered from standard input and they are executed by the pattern clauses.

Regular output is produced only by interpreting reports. The design goal is that the *awk* program contain as little information about the generated code as possible.

Code generation should be controlled totally by means of changing the report files. Since the *ooc* command script permits substitution of report files, the application programmer may modify all output, at least theoretically, without having to change the *awk* program.

B.2 File Management — *io.awk*

This module is responsible for accessing all files. It maintains **FileStack[]** with name and line number of all open files. **openFile(*fnm*)** pushes **FILENAME** and **FNR** onto this stack and uses **system()** to find out if a file *fnm* can be read. The complete name is then set into **FILENAME** and **FNR** is reset. The function **get()** reads from **FILENAME** and returns a complete input line with no comments and continuations or the special value **EOF** at end of file. This value starts with **%** to simplify certain loops. **closeFile()** closes **FILENAME** and pops the stack.

openFile() implements a search path **OOC_PATH** for all files. This way, reports, class descriptions, and implementations can be stored in a central place for an installation or a project.

io.awk contains two more functions: **error()** to output an error message, and **fatal()** to issue a message and terminate execution with the exit code 1. **error()** also sets the exit code 1 as value of a global variable **status**. Debugging modules will eventually return **status** with an **END** clause.

If *main.awk* contained an **END** clause, *awk* would wait for input after processing all **BEGIN** clauses. Therefore, we set an *awk* variable **debug** from the *ooc* command script to indicate if we have loaded debug modules with pattern clauses. If **debug** is not set, the **BEGIN** clause in *main.awk* is terminated by executing **exit** and passing **status**.

B.3 Recognition — *parse.awk*

parse.awk extracts the database from the class description files. The top level function **load(*desc*)** processes a class description file *desc.d*. Each such file is only read once. The internal function **classDeclaration()** parses a class description; **structDeclarator()** takes care of one line in the representation of a class; **methodDeclaration()** handles a single method declaration; and **declarator()** is called to process each declarator.

All of these functions are quite straightforward. They use **sub()** and **gsub()** to massage input lines for recognition and **split()** to tokenize them. This is insufficient for analyzing a general C declarator; therefore, we limit ourselves to simple declarators where the type precedes the name.

The debugging module *parse.dbg* understands the inputs **classes** and **descriptions** to dump information about all classes and description files, or **all** to do both. For an input like *desc.d* it will load a class description file. Other inputs should be class, description, or method names to dump individual entries in the database.

B.4 The Database

For a class description file, we save the individual lines so that we can copy them to the interface or representation file. Among these lines we need to remember where which classes and metaclasses were defined. The latter information is also required to generate appropriate initializations. Therefore, we produce three arrays: **Pub**[*desc*, *n*] contains lines for the interface file, **Prot**[*desc*, *n*] contains lines for the representation file, and **Dcl**[*desc*, *n*] only records the class and metaclass definitions. For each description name *desc* the index 0 holds the number of lines and the lines are stored with indices from 1 on up. **Dcl**[*desc*, 0] exists exactly, if we have read the description for *desc*. The lines are stored unchanged, we only replace a complete class definition by a line starting with % and containing the metaclass name, if any, and then the class name.

For a class, our database contains its meta- and superclass name, the components of its representation, and the names of its methods. We use a total of six arrays: **Meta**[*class*] contains the metaclass name, **Super**[*class*] contains the superclass name, **Struct**[*class*, *n*] is a list of the component declarator indices, and **Static**[*class*, *n*], **Dynamic**[*class*, *n*], and **Class**[*class*, *n*] contain lists of the various method names. Again, index 0 holds the list length, and the list elements are stored with indices from 1 on up. **Class**[*class*, 0] exists exactly, if we know *class* to be a class or metaclass name.

For a method, we need to remember its name, result, parameter list, linkage, and tag for the **respondsTo()** method. This information is represented with the following six arrays: **Method**[*method*] is the first declarator index; it describes the method name and result type. The parameter declarators follow with ascending indices; **Nparm**[*method*] is the number of parameters. There has to be at least the **self** parameter. **Var**[*method*] is true if *method* permits a variable number of parameters, **Linkage**[*method*] is one of %, %-, or %+ and records in which linkage section the method was declared. **Owner**[*method*] is important for statically linked methods; it contains the class to which the method belongs, i.e., the class of the method's **self** parameter. Finally, **Tag**[*method*] records the default tag of the method for the purposes of **respondsTo()**, and **Tag**[*method*, *class*] holds the actual tag of a *method* for a *class*.

Class representation components and method names and parameters are described as indices into a list of declarators. The list is represented by four arrays: **Name**[*index*] is the name of the declarator, **Const**[*index*] contains the **const** prefix of the declarator, if any. **As**[*index*] is true if @ was used in the declarator, i.e., if the declarator specifies a reference to an object. In this case **Type**[*index*] is either a class name or it is empty if the object is in the owner class of the method. If **As**[*index*] is false, **Type**[*index*] is the type part of the declarator.

Finally, if the global variable **lines** is set, the database contains four more arrays: **Filename**[*name*] and **Fnr**[*name*] indicate where a class or a method was described, **SFilename**[*name*] and **SFnr**[*name*] indicate where a class component was declared. This is used in *report.awk* to implement **#line** stamps.

B.5 Report Generation — *report.awk*

report.awk contains functions to load reports from files and to generate output from reports. This is the only module which prints to standard output; therefore, the tracking of line numbers happens in this module. A simple function **puts()** is available to print a string followed by a newline.

Reports are loaded by calling **loadReports()** with the name of the file to load reports from. To simplify debugging, reports may not be overwritten.

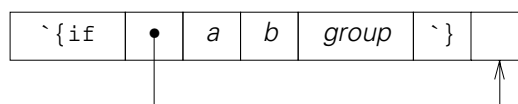
Reports are generated by calling **gen()** with the name of a report. A certain effort is made to avoid emitting leading spaces or more than one empty line in a row: a global variable **newLine** is 0 at the left margin or 1 once we have printed something; an internal function **If()** prints a newline and decrements **newLine** by 1. Spaces are only emitted if **newLine** is 1, i.e., if we are inside a line. Newlines are only emitted if **newLine** is not -1, i.e., if we have not just emitted an empty line.

Report generation is based on a few simple observations: It is cheap to read report lines, use **split()** to take them apart into words separated by single blanks or tabs, and store them in a big array **Token[]**. Another array **Report[]** holds for each report name the index of its first word in **Token[]**. The internal function **endReport()** checks that braces are balanced in each report and terminates each report by an extra closing brace.

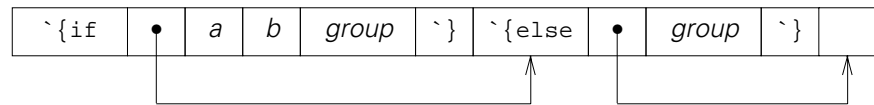
If a single blank or tab character is used as the split character, and if we emit a single blank for an empty word, a report closely resembles the generated output: two blanks represent one blank in the output. Generation is reasonably efficient if we can quickly identify words to be left unchanged (they do not start with a back quote) and if we have a fast way to search for replacements (they are stored in an array **Replace[]** indexed by the words to be replaced). Elements of **Replace[]** are mostly set by functions defined in *parse.awk* which look at the database: **setClass()**, **setMethod()**, and **setDeclarator()** set the information described in the table at the end of the manual page of *ooc* in section C.1.

Groups are simple to implement. When reading the report lines, after each word starting with **{**, i.e., at the beginning of each group, we store the index of the word following the matching **}**, i.e., the index past the end of the group. This requires maintaining a stack of open braces, but the stack can be stored in the same places where we later store the indices of the corresponding closing braces.

During execution, we run the report generator recursively. The contents of a group are interpreted by a call to **genGroup()** that returns at the closing brace. For a loop we can issue as many calls as necessary, and eventually we continue execution by following the index to the position past the group. At the global level, each report is terminated by one extra closing brace token. **{if** groups are just as easy:



If the comparison works out, we recursively execute *group*. In any case we continue execution past the group. For an **{else** we have the following arrangement:



If the comparison works out, we recursively execute its group. Afterwards, we can follow both index values or we can arrange for an **{else** group to always be skipped when it is encountered directly. If the comparison fails, and if the index after **{if** points to **{else**, we position to the **{else** and recursively execute its group. Afterwards we follow the index as usual.

The termination token **}** can contain arbitrary text following the brace. However, there are two special cases. The loop over the parameters of a method calls **genGroup()** with an extra parameter **more** set to 1 as long as there are some method parameters yet to be processed. If **more** is set, **genGroup()** emits a comma and a space for the termination token **},**. This simplifies generating parameter lists.

The other special termination token is **}n** which results in an extra newline if anything was output for the group. **genGroup()** returns a truth value indicating whether it was terminated by the token **}n** or not. Functions such as **genLoopMethods()**, which drive a loop over calls to **genGroup()**, return the value of **genGroup()** if the loop was entered and false otherwise. Finally, **genGroup()** will emit the extra newline exactly if the loop function returns true, i.e., if the loop was entered and terminated by **}n**. This simplifies block arrangements in the generated code.

The debugging module *report.dbg* accepts a filename like *c.rep* and loads the reports from the file. Given a valid report name, it will symbolically display the report. Given **all** or **reports**, it will show all reports.

B.6 Line Numbering

A preprocessor should output **#line** stamps so that the C compiler can relate its error messages to the original source files. Unfortunately, *ooc* consults several input files to produce certain output lines, i.e., there appears to be no implicit relationship between class description files, source files, and an output line. Moreover, if report files are formatted to be legible they tend to generate many blank lines which in turn could result in many **#line** stamps.

We compromise and only generate a **#line** stamp if a report requests it. The stamp can be based on a class, method, or structure component name, or it can record the current input file name and line number. The current input file position is available as **FILENAME** and **FNR** from the *io.awk* module. The other positions have been recorded by *parse.awk*. A function **genLineStamp()** in *report.awk* collects the required information and produces the **#line** stamp.

We could optimize by counting the output lines — all the information is available in *report.awk*. However, experience indicates that this slows *ooc* down considerably. A few extra **#line** stamps or newlines will not delay compilation very much.

The entire feature is only enabled by setting the global variable **lines** to a nonzero value. This is under control of an option **-l** passed to the *ooc* command script.

B.7 The Main Program — *main.awk*

The **BEGIN** clause in *main.awk* should be the last one executed. It processes each argument in **ARGV[]** and deletes it from the array. A name like *c.rep* is interpreted as a report file to be loaded with **loadReports()**. A name like *Object.dc* is an implementation to be preprocessed. **-dc**, **-h**, and **-r** result in reports by these names to be interpreted. Any other argument should be a class name for which the description is loaded with **load()**; the name is set as replacement for **`desc**. Such an argument must precede most of the other arguments, because **`desc** is remembered for report generation.

load() recursively loads all class descriptions back to the root class. If the *awk* variable **makefile** is set by the *ooc* command script, the report **-M** is generated for each class name specified as an argument. This normally produces lines for a *makefile* to indicate how class description files depend on each other. However, *ooc* cannot detect that as a result of preprocessing an implementation file *class.c* depends on the class description file *class.d* in addition to the file *class.dc*. This dependency must be added to a *makefile* separately.

main.awk contains two functions. **preprocess()** takes care of the preprocessing of an implementation file. It generates the report **include** at the beginning of the implementation file. It calls **methodHeader()** for the various ways in which a method header can be requested, and it generates the reports **casts** and **init** for the preprocessor statements **%casts** and **%init**.

methodHeader() generates the report **methodHeader** and it records the method definition in the database: **Links[class, n]** is the list of method names defined for a class and **Tags[method, class]** is the actual tag defined for a method in a class. These lists are used in the initialization report.

B.8 Report Files

Reports are stored in several files to simplify maintenance. *h.rep* and *r.rep* contain the reports for the interface and representation files. *c.rep* contains the reports for preprocessing an implementation file. There are two versions of each of these files, one for the root class, and one for all other classes. *m.rep* contains the report for the *makefile* option **-M** and *dc.rep* contains the report for **-dc**. Three other files, *etc.rep*, *header.rep*, and *va.rep*, contain reports that are called from more than one other file.

Dividing the reports among several files according to command line options has the advantage that we can check the command line in the *ooc* command script and only load those files which are really needed. Checking is a lot cheaper than loading and searching through many unused reports.

With `{if}` groups and report calls through `%` we can produce more or less convoluted solutions. The basic idea was to make things easier to read and more efficient by duplicating some decisions in the reports called by the **selector** report and by separating the reports for the root class and the other classes. As *ooc* evolves through the chapters, we modify some reports anyway.

B.9 The *ooc* Command

ooc can load an arbitrary number of reports and descriptions, output several interface and representation files, and suggest or preprocess various implementation files, all in one run. This is a consequence of the modular implementation. However, *ooc* is a genuine filter, i.e., it will read files as directed by the command line, but it will only write to standard output. If several outputs are produced in one run, they would have to be split and written to different files by a postprocessor based on *awk* or *csplit*. Here are some typical invocations of *ooc*:

```
$ ooc -R Object -h > Object.h          # root class
$ ooc -R Object -r > Object.r
$ ooc -R Object Object.dc > Object.c

$ ooc Point -h > Point.h                # other class
$ ooc -M Point Circle >> makefile       # dependencies
$ echo 'Point.c: Point.d' >> makefile
$ ooc Circle -dc > Circle.dc            # start an implementation
$ ooc Circle -dc | ooc Circle - > Circle.c # fake...
```

If *ooc* is called without arguments, it produces the following usage description:

```
$ ooc
usage:  ooc [option ...] [report ...] description target ...

options:  -d          arrange for debugging
          -l          make #line stamps
          -Dnm=val    define val for `nm` (one word)
          -M          make dependency for each description
          -R          process root description
          -7 -8 ...   versions for book chapters
report:   report.rep  load alternative report file
description: class    load class description file
targets:  -dc         make thunks for last 'class'
          -h         make interface for last 'class'
          -r         make representation for last 'class'
          -          preprocess stdin for last 'class'
          source.dc   preprocess source for last 'class'
```

It should be noted that if any report file is loaded, the standard reports are not loaded. The way to replace only a single standard report file is to provide a file by the same name earlier on **OOCPATH**.

The *ooc* command script needs to be reviewed during installation. It contains **AWKPATH**, the path for the *awk* processor to locate the modules, and **OOCPATH** to locate the reports. This last variable is set to look in a standard place as a last resort; if *ooc* is called with **OOCPATH** already defined, this value is prefixed to the standard place.

To speed things up, the command script checks the entire command line and loads only the necessary report files. If *ooc* is not used correctly, the script emits the usage description shown above. Otherwise *awk* is executed by the same process.

Appendix C Manual

This appendix contains UNIX manual pages describing the final version of *ooc* and some classes developed in this book.

C.1 Commands

munch — produce class list

nm -p object... archive... | **munch**

munch reads a Berkeley-style *nm*(1) listing from standard input and produces as standard output a C source file defining a null-terminated array **classes[]** with pointers to the class functions found in each *object* and *archive*. The array is sorted by class function names.

A class function is any name that appears with type **T** and, preceded with an underscore, with type **b**, **d**, or **s**.

This is a hack to simplify retrieval programs. The compatible effect of option **-p** in Berkeley and System V *nm* is quite a surprise.

Because HP/UX *nm* does not output static symbols, *munch* is not very useful on this system.

ooc — preprocessor for object-oriented coding in ANSI C

ooc [option ...] [report ...] description target ...

ooc is an *awk* program which reads class descriptions and performs the routine coding tasks necessary to do object-oriented coding in ANSI C. Code generated by *ooc* is controlled by reports which may be changed. This manual page describes the effects of the standard reports.

description is a class name. *ooc* loads a class description file with the name *description.d* and recursively class description files for all superclasses back to the root class. If **-h** or **-r** is specified as a *target*, a C header file for the public interface or the private representation of *description* is written to standard output. If *source.dc* or **-** is specified as a *target*, **#include** statements for the *description* header files are written to standard output and *source.dc* or standard input is read, preprocessed, and copied to standard output. If **-dc** is specified as a *target*, a source skeleton for *description* is written to standard output, which contains all possible methods.

The output is produced by report generation from standard report files. If *file.rep* is specified as a *report*, the standard files are not loaded.

There are some global *options* to control *ooc*:

- D***name*[=*value*]
defines *value* or an empty string as replacement for *`name*. The *name* should be a single word. *ooc* predefines **GNUC** with value **0**.
- d**
arranges for debugging to follow normal processing. Debugging commands are read from standard input: *class.d* loads a class description file; *report.rep* loads a report file; a description, report, class, or method name produces a dump of the appropriate information; and **all**, **classes**, **descriptions**, or **reports** dump all information in the respective category.
- I**
produces **#line** stamps as directed by the reports.
- M**
produces a *makefile* dependency line between each *description* and its superclass description files.
- R**
must be specified if the root class is processed. Other standard reports are loaded in this case.

Lexical Conventions

All input lines are processed as follows: first, a comment is removed; next, lines are glued together as long as they end with a backslash; finally, trailing white space is removed.

A comment extends from *//* to the end of a line. It is removed together with preceding white space before glueing takes place.

In glueing, the backslash marks the contact point and is removed. All white space around the contact point is replaced with a single space.

Identifiers significant to *ooc* follow the conventions of C, except that they may not use underscore characters. The underscore is used to avoid clashes between *ooc*'s and the user's code.

Declarators significant to *ooc* are simplified relative to C. They may start with **const** and the type information must precede the name. The type information may use ***** but no parentheses. In general, an arbitrary declarator can be adapted for *ooc* by introducing a type name with **typedef**.

A line starting with **%%** acts as end of file.

Class Description File

The class description file has the following format:

```
header
% meta class {
  components
```

```

%
    methods with static linkage
%-
    methods with dynamic linkage
%+
    class methods
%}
...

```

header is arbitrary information which is copied to standard output if the interface file is produced. Information following **%prot** is copied to standard output if the representation file is produced.

components are C structure component declarations with one declarator per line. They are copied into the **struct** generated in the representation file for the class. They also determine the order of the construction parameters for the root metaclass.

The first set of *methods* has static linkage, i.e., they are functions with at least one object as a parameter; the second set has dynamic linkage and has an object as a parameter for which the method is selected; the third set are class methods, i.e., they have a class as a parameter for which the method is selected. The selection object is always called **self**. The method declarations define C function headers, selectors, and information for the metaclass constructor.

The class header line **% meta class {** has one of three forms. The first form is used to introduce the root class only:

```

% meta class {
    class is the root class, indicated by the fact that it has no superclass. The
    superclass is then defined to be the root class itself. meta should be intro-
    duced later as the root metaclass, indicated by the fact that it has itself as
    metaclass.

```

```

% meta class: super {
    class is a new class with meta as its metaclass and super as its superclass.
    This would also be used to introduce the root metaclass, which has itself as
    metaclass and the root class as superclass. If super is undefined, ooc will
    recursively (but only once) load the class description file super.d and then
    super and meta must have been defined so that class can be defined. If
    this form of the class header is used, only methods with static linkage can
    be introduced.

```

```

% meta: supermeta class: super {
    This additionally defines meta as a new metaclass with supermeta as its
    superclass. If super is undefined, ooc will recursively (but only once) load
    the class description file super.d and then super and supermeta must have
    been defined so that meta and class can be defined.

```

A method declaration line has the following form, where braces indicate zero or more occurrences and brackets indicate an optional item:


```
[ tag : ] declarator ( declarator { , declarator } [ , ... ] );
```

The optional *tag* is an identifier involved in locating a method with **respondsTo()**. The first *declarator* introduces the method name and result type, the remaining *declarators* introduce parameter names and types. Exactly one parameter name must be **self** to indicate the receiver of the method call.

A *declarator* is a simplified C declarator as described above, but there are two special cases:

_name

introduces *name* as the declarator name. The type is a pointer to an instance of the current class or to the class for which a dynamically linked method is overwritten. Such a pointer will be dereferenced by **%casts** as *name* within a method. Therefore, **self** must be introduced as **_self**, where **self** is the dereferenced object or class for class methods and **_self** is the raw pointer.

class @ name

introduces *name* as a pointer to an instance of *class*. Such a pointer will not be dereferenced but it will be checked by **%casts**.

The result type of a method can employ *class @*. In this case, the result type is generated as a pointer to a **struct class** which is useful when implementing methods, and which cannot be used other than for assignments to **void *** in application code. The result type should be **void *** for constructors and similar methods to emphasize the generic aspects of construction.

Preprocessing

Subject to the lexical conventions described above, an implementation file *source.dc* is copied to standard output. Lines starting with **%** are preprocessed as follows:

% class method {

This is replaced by a C function header for *method*; the header is declared **static** with the name *class_method*, unless *method* has static linkage. In the latter case, *class* is optional. *ooc* checks in all cases that the method can be specified for *class*. Function names are remembered as necessary for initialization of the description of *class* and the corresponding metaclass if any. There can be an optional tag preceding *class* unless *method* has static linkage.

%casts

This is replaced by definitions of local variables to securely dereference parameter pointers to objects in the current class. For statically linked methods this is followed by checks to verify the parameters pointing to objects of other classes. **%casts** should be used where local variables can be defined; for statically linked methods it must be the last definition. Note that null pointers flunk the checks and terminate the calling program.

%init

This should be near the end of the implementation file. If the *description* introduced a new metaclass, a constructor for the metaclass, selectors for the class, and initializations for the metaclass are generated. In either case, an initialization for the class is generated.

If a method *m* does not have static linkage, there are two selectors: *m* with the same parameters as the method selecting the method defined for **self**, and **super_m** with an explicit class description as an additional first parameter. The second selector is used to pass a method call to the superclass of the class where the method is defined.

If a dynamically linked or class method has a variable argument list, the selector passes **va_list * app** to the actual method.

If a selector recognizes that it cannot be applied to its object, it calls **forward** and passes its object, a pointer to a result area, or a null pointer, its own address, its name as a string, and its entire argument list. **forward** should be a dynamically linked method in the root class; it can be used to forward a message from one object to another.

Tags

respondsTo() is a method in the root class which takes an object and a tag, i.e., a C string containing an identifier, and returns either a null pointer or a selector which will accept the object and other parameters and call the method corresponding to the tag.

The tag under which a class or dynamically linked method can be found is defined as follows. The default is either the method name or *tag* in the method header in the class description file:

```
[ tag : ] declarator ( declarator { , declarator } [ , ... ] );
```

The method header in the implementation may overwrite the tag:

```
% mtag: class method {
```

The effective tag is *mtag* if specified, or *tag* if not. If *mtag* or *tag* is empty but the colon is specified, **respondsTo()** cannot find the method.

Report File

ooc uses report files containing all code fragments which *ooc* will generate. Names such as **app** for an argument list pointer can be changed in the report file. Only **self** is built into *ooc* itself.

A report file contains one or more reports. The usual lexical conventions apply. Each report is preceded by a line starting with **%** and containing the report name which may be enclosed by white space. The report name is arbitrary text but it must be unique.

A report consists of lines of words separated by single blanks or tabs, called spaces. An empty word results between any two adjacent spaces or if a space starts or ends a line.

An empty word, not at the beginning of an output line, is printed as a blank. In particular, this means that two successive spaces in a report represent a single blank to be printed. Any word not starting with a back quote ` is printed as is.

A word starting with `% causes a report to be printed. The report name is the remainder of the word.

`#line followed by a word causes a line stamp to be printed if option **-l** is specified; the phrase is ignored otherwise. If the word is a class, method, or class component name, the line stamp refers to its position in a class description file. Otherwise, and in particular for empty words, the line stamp refers to the current input file position.

A word starting with `{ starts a group. The group is terminated with a word starting with `}. All other words starting with a back quote ` are replaced during printing. Some replacements are globally defined, others are set up by certain groups. A table of replacements follows at the end of this section.

Groups are either loops over parts of the database collected by *ooc* or they are conditionals based on a comparison. Words inside a group are printed under control of the loop or the comparison. Afterwards, printing continues with the word following the group. Groups can be nested, but that may not make sense for some parts of the database. Here is a table of words starting a loop:

`{%	static methods for the current `class
`{%-	dynamic methods for the current `class
`{%+	class methods for the current `class
`{()	parameters for the current `method
`{dcl	class headers in the `desc description file
`{pub	public lines in the `desc description file
`{prot	protected lines in the `desc description file
`{links <i>class</i>	dynamic and class methods defined for <i>class</i>
`{struct <i>class</i>	components for <i>class</i>
`{super	`desc and all its superclasses back to `root

A loop is terminated with a word starting with `}. If the terminating word is `}, in the loop over parameters, and if the loop will continue for more parameters, a comma followed by a blank is printed for this word. If the terminating word is `}n and if the group has produced any output, a newline is printed for this word. Otherwise, nothing is printed for termination.

A conditional group starts with **`{if** or **`{ifnot** followed by two words. The words are replaced if necessary and compared. If they are equal, the group starting with **`{if** is executed; if they are not equal, the group starting with **`{ifnot** is executed. If either group is not executed and if it is followed by a group starting with **`{else**, this group is executed. Otherwise the **`{else** group is skipped.

In general it is best if the **`}** terminating the **`{if** group immediately precedes **`{else** on the same line of the report.

Here is a table of replaced words together with the events that set up the replacements:

set up globally

<code>`</code>	no text (empty string)
<code>``</code>	<code>`</code> (back quote)
<code>`t`</code>	tab
<code>`n`</code>	newline (limited to one blank line)

set up once class descriptions are loaded

<code>`desc`</code>	last <i>description</i> from command line
<code>`root`</code>	root class' name
<code>`metaroot`</code>	root's metaclass name

set up for a class`% %- %+ `{dcl `{prot `{pub `{super`

<code>`class`</code>	class' name
<code>`super`</code>	class' superclass name
<code>`meta`</code>	class' metaclass name
<code>`supermeta`</code>	metaclass' superclass name

set up for a method`{% `{%- `{%+ `{links class`

<code>`method`</code>	method's name
<code>`result`</code>	method's result type
<code>`linkage`</code>	method's linkage: <code>%</code> , <code>%-</code> , or <code>%+</code>
<code>`tag`</code>	method's tag
<code>`,`</code>	, ... if variable arguments are declared, empty if not
<code>`_last`</code>	last parameter's name if variable arguments, undefined if not

set up for a declarator`{() `{struct class`

<code>`name`</code>	name in declarator
<code>`const`</code>	const followed by a blank, if so declared
<code>`type`</code>	void * for objects, declared type otherwise
<code>`_`</code>	<code>_</code> if used in declaration, empty otherwise
<code>`cast`</code>	object's class name, empty otherwise

set up for lines from the description file`{dcl `{prot `{pub`

<code>`class`</code>	set up for a class description, empty otherwise
<code>`line`</code>	line's text if not class description, undefined otherwise
<code>`newmeta`</code>	1 if new metaclass is defined, 0 if not

A *description* on the command line of *ooc* sets up for a class. Requesting a method header in a source file sets up for a class and a method. The loops `{dcl`, `{prot`, and `{pub` set up for lines from a class description file. The loops `{%`, `{%-`, `{%+`, and `{links class` set up for a method. The loop `{()` sets up for a parameter declarator. The loop `{struct class` sets up for the declarator of a component of a class. The loop `{super` runs from *description* through all its superclasses.

Environment

OOC_PATH is a colon-separated list of paths. If a file name does not contain path delimiters, *ooc* looks for the file (class descriptions, sources, and report files) by

prefixing each entry of **OOCPATH** to the required file name. By default, **OOCPATH** consists of the working directory and a standard place.

FILES	<i>class.d</i>	description for <i>class</i>
	<i>class.dc</i>	implementation for <i>class</i>
	<i>report.rep</i>	report file
	<i>AWKPATH/*.awk</i>	modules
	<i>AWKPATH/*.dbg</i>	debugger modules
	<i>OOCPATH/c.rep</i>	implementation file reports
	<i>OOCPATH/dc.rep</i>	implementation thunks report
	<i>OOCPATH/etc.rep</i>	common reports
	<i>OOCPATH/h.rep</i>	interface file report
	<i>OOCPATH/header.rep</i>	common reports
	<i>OOCPATH/m.rep</i>	<i>makefile</i> dependency report
	<i>OOCPATH/r.rep</i>	representation file reports
	<i>OOCPATH/va.rep</i>	common reports
	<i>OOCPATH/[chr]-R.rep</i>	root class versions

The C preprocessor is applied to the output of *ooc*, not to the input, i.e., conditional compilation should not be applied to *ooc* controls.

C.2 Functions

retrieve — get object from file

void * retrieve (FILE * fp)

retrieve() returns an object read in from the input stream represented by *fp*. At end of file or in case of an error, **retrieve()** returns a null pointer.

retrieve() requires a sorted table of class function pointers that can be produced with *munch*(1). Once the class description has been located, **retrieve()** applies the method **geto** to an area obtained with **allocate**.

SEE ALSO *munch*(1), *Object*(3)

C.3 Root Classes

intro — introduction to the root classes

Object	Class
Exception	

Object(3) is the root class; **Class**(3) is the root metaclass. Most of the methods defined for **Object** are used in the standard reports for *ooc*(1), i.e., they cannot be changed without changing the reports.

Exception(3) manages a stack of exception handlers. This class is not mandatory for working with *ooc*.

Class Class: Object - root metaclass

Object

Class**new(Class(), name, superclass, size, selector, tag, method, ... , 0);****Object @ allocate (const self)****const Class @ super (const self)****const char * nameOf (const self)**

A metaclass object describes a class, i.e., it contains the class *name*, a pointer to the class' *super* class description, the *size* of an object in the class, and information about all dynamically linked methods which can be applied to objects of the class. This information consists of a pointer to the *selector* function, a *tag* string for the **respondsTo** method (which may be empty), and a pointer to the actual *method* function for objects of the class.

A metaclass is a collection of metaclass objects which all contain the same variety of method informations, where, of course, each metaclass object may point to different methods. A metaclass description describes a metaclass.

Class is the root metaclass. There is a metaclass object **Class** which describes the metaclass **Class**. Every other metaclass *X* is described by some other metaclass object *X* which is a member of **Class**.

The metaclass **Class** contains a metaclass object **Object** which describes the root class **Object**. A new class *Y*, which has the same dynamically bound methods as the class **Object**, is described by a metaclass object *Y*, which is a member of **Class**.

A new class *Z*, which has more dynamically bound methods than **Object**, requires a metaclass object *Z*, which is a member of a new metaclass *M*. This new metaclass has a metaclass description *M*, which is a member of **Class**.

The **Class** constructor is used to build new class description objects like *Y* and metaclass description objects like *M*. The *M* constructor is used to build new class description objects like *Z*. The *Y* constructor builds objects which are members of class *Y*, and the *Z* constructor builds objects in class *Z*.

allocate reserves memory for an object of its argument class and installs this class as the class description pointer. Unless overwritten, **new** calls **allocate** and applies **ctor** to the result. **retrieve** calls **allocate** and applies **geto** to the result.

super returns the superclass from a class description.

nameOf returns the name from a class description.

The **Class** constructor **ctor** handles method inheritance. Only information about overwritten methods needs to be passed to **new**. The information consists of the address of the selector to locate the method, a tag string which may be empty, and the address of the new method. The method information tuples may appear in any order of methods; zero in place of a *selector* terminates the list.

delete, **dtor**, and **geto** are inoperative for class descriptions.

Class descriptions are only accessed by means of functions which initialize the description during the first call.

SEE ALSO `ooc(1)`, `retrieve(2)`

Class **Exception: Object** — manage a stack of exception handlers

Object

Exception

new(Exception());

int catch (self)

void cause (int number)

Exception is a class for managing a stack of exception handlers. After it is armed with **catch**, the newest **Exception** object can receive a nonzero exception number sent with **cause()**.

ctor pushes the new **Exception** object onto the global exception stack, **dtor** removes it. These calls must be balanced.

catch arms its object for receiving an exception number. Once the number is sent, **catch** will return it. This function is implemented as a macro with **setjmp(3)** and is subject to the same restrictions; in particular, the function containing the call to **catch** must still be active when the exception number is sent.

Other methods should generally not be applied to an **Exception** object.

SEE ALSO `setjmp(3)`

Class **Object** — root class

Object

Class

new(Object());

typedef void (* Method) ();

const void * classOf (const self)

size_t sizeOf (const self)

int isA (const self, const Class @ class)

int isOf (const self, const Class @ class)

void * cast (const Class @ class, const self)

Method respondsTo (const self, const char * tag)

%—

void * ctor (self, va_list * app)

void delete (self)

void * dtor (self)

int puto (const self, FILE * fp)

void * geto (self, FILE * fp)

void forward (self, void * result, Method selector, const char * name, ...)

%+

Object @ new (const self, ...)

Object is the root class; all classes and metaclasses have **Object** as their ultimate superclass. Metaclasses have **Class** as their penultimate superclass.

classOf returns the class description of an object; **sizeOf** returns the size in bytes.

isA returns true if an object is described by a specific class description, i.e., if it belongs to that class. **isA** is false for null pointers. **isOf** returns true, if an object belongs to a specific class or has it as a superclass. **isOf** is false for null pointers and true for any object and the class **Object**.

cast checks if its second argument is described, directly or ultimately, by the first. If not, and in particular for null pointers, the calling program is terminated. **cast** normally returns its second argument unchanged; for efficiency, **cast** could be replaced by a macro.

respondsTo returns zero or a method selector corresponding to a tag for some object. If the result is not null, the object with other arguments as appropriate can be passed to this selector.

ctor is the constructor. It receives the additional arguments from **new**. It should first call **super_ctor**, which may use up part of the argument list, and then handle its own initialization from the rest of the argument list.

Unless overwritten, **delete** destroys an object by calling **dtor** and sending the result to **free(3)**. Null pointers may not be passed to **delete**.

dtor is responsible for reclaiming resources acquired by the object. It will normally call **super_dtor** and let it determine its result. If a null pointer is returned, **delete** will effectively not reclaim the space for the object.

puto writes an ASCII representation of an object to a stream. It will normally call **puto** for the superclass so that the output starts with the class name. The representation must be designed so that **geto** can retrieve all but the class name from the stream and place the information into the area passed as first argument. **geto** works just like **ctor** and will normally let the superclass **geto** handle the part written by the superclass **puto**.

forward is called by a selector if it cannot be applied to an object. The method can be overwritten to forward messages.

Unless overwritten, **new** calls **allocate** and passes the result to **ctor** together with its remaining arguments.

SEE ALSO ooc(1), retrieve(2), Class(3)

C.4 GUI Calculator Classes

intro — introduction to the calculator application

Objct	Class
Event	
lc	lcClass
Button	
Calc	
Crt	
CButton	
CLineOut	
LineOut	
Mux	
List	ListClass
Xt	
XawBox	
XawCommand	
XButton	
XawForm	
XawLabel	
XLineOut	
XtApplicationShell	

Object(3) is the root class. **Object** needs to be renamed as **Objct** because the original name is used by X11.

Event(4) is a class to represent input data such as key presses or mouse clicks.

lc(4) is the base class to represent objects that can receive, process, and send events. **Button** converts incoming events to events with definite text values. **Calc** processes texts and sends a result on. **LineOut** displays an incoming text. **Mux** tries to send an incoming event to one of several objects.

Crt(4) is a class to work with the *curses* terminal screen function package. It sends position events for a cursor and text events for other key presses. **CButton** implements **Button** on a *curses* screen. **CLineOut** implements **LineOut**.

List manages a list of objects and is taken from chapter 7.

Xt(4) is a class to work with the X Toolkit. The subclasses wrap toolkit and Athena widgets. **XButton** implements a **Button** with a **Command** widget. **XLineOut** implements a **LineOut** with a **Label** widget.

SEE ALSO *curses*(3), *X*(1)

lcClass Crt: lc — input/output objects for curses

Objct
lc

Crt

CButton
CLineout

```
new(Crt());
new(CButton(), "text", y, x);
new(CLineOut(), y, x, len);

void makeWindow (self, int rows, int cols, int x, int y)
void addStr (self, int y, int x, const char * s)
void crtBox (self)
```

A **Crt** object wraps a *curses*(3) window. *curses* is initialized when the first **Crt** object is created.

Crt_gate() is the event loop: it monitors the keyboard; it implements a *vi*-style cursor move for the keys **hjkl**, and possibly, for the arrow keys; if *return* is pressed, it sends an **Event** object with *kind* 1 and an array with column and row position; if *control-D* is pressed, **gate** returns **reject**; any other key is sent on as an **Event** object with a string containing the key character.

A **CLineOut** object implements a **LineOut** object on a *curses* screen. Incoming strings should not exceed *len* bytes.

A **CButton** object implements a **Button** object on a *curses* screen. If it receives a matching text, it sends it. Additionally, if it receives a position event, e.g., from a **Crt** object, and if the coordinates are within its window, it sends its text on.

SEE ALSO Event(4)

Class Event: Object — input item

Objct

Event

```
new(Event(), kind, data);

int kind (self)
void * data (self)
```

An **Event** object represents an input item such as a piece of text, a mouse click, etc.

kind is zero if *data* is a static string. *kind* is not zero if *data* is a pointer. In particular, a mouse click can be represented with *kind* 1 and *data* pointing to an array with two integer coordinates.

SEE ALSO lc(4)

lcClass: Class lc: Object — basic input/output/transput objects

```

Object
  lc
    Button
    Calc
    LineOut
    Mux

new(lc());
new(Button(), "text");
new(Calc());
new(LineOut());
new(Mux());

%—
void wire (Object @ to, self)
enum { reject, accept } gate (self, const void * item)

```

An **lc** object has an output pin and an input action. **wire()** connects the output to some other object. If an **lc** object is sent a data *item* with **gate()**, it will perform some action and send some result to its output pin; some **lc** objects only create output and others only consume input. **gate()** returns **accept** if the receiver accepts the data.

lc is a base class. Subclasses overwrite **gate()** to implement their own processing. **lc_gate()** takes *item* and uses **gate()** to send it on to the output pin, i.e., a subclass will use **super_gate()** to send something to its output pin.

A **Button** object contains a text which is sent out in response to certain inputs. It expects an **Event** object as input. If the **Event** contains a matching text or a null pointer or other data, the **Button** accepts the input and sends its own text on. A non-matching text is rejected.

Button is designed as a base class. Subclasses should match mouse positions, etc., and use **super_gate()** to send out the appropriate text.

A **Calc** object receives a string, computes a result, and sends the current result on as a string. The first character of the input string is processed: digits are assembled into a non-negative decimal number; **+**, **-**, *****, and **/** perform arithmetic operations on two operands; **=** completes an arithmetic operation; **C** resets the calculator; and **Q** quits the application. The calculator is a simple, precedence-free, finite state machine: the first set of digits defines a first operand; the first operator is saved; more digits define another operand; if another operator is received, the saved operator is executed and the new operator is saved. Invalid inputs are accepted and silently ignored.

A **LineOut** object accepts a string and displays it.

A **Mux** object can be wired to a list of outputs. It sends its input to each of these outputs until one of them accepts the input. The list is built and searched in order of the **wire()** calls.

SEE ALSO Crt(4), Event(4), Xt(4)

Class Xt: Object — input/output objects for X11

Object

Xt

```

XawBox
XawCommand
  XButton
XawForm
XawLabel
  XLineOut
XtApplicationShell

```

```

new(Xt());
new(XtApplicationShell(), & argc, argv);
new(XawBox(), parent, "name");
new(XawCommand(), parent, "name");
new(XawForm(), parent, "name");
new(XawLabel(), parent, "name");
new(XButton(), parent, "name", "label");
new(XLineOut(), parent, "name", "label");

void * makeWidget (self, WidgetClass wc, va_list * app)
void addAllAccelerators (self)
void setLabel (self, const char * label)
void addCallback (self, XtCallbackProc fun, XtPointer data)

void mainLoop (self)

```

An **Xt** object wraps a widget from the X toolkit. **makeWidget()** is used to create and install the widget in the hierarchy; it takes a *parent* **Xt** object and a widget *name* from the argument list pointer to which *app* points. **addAllAccelerators()** is used to install the accelerators below the **Xt** object. **setLabel()** sets the **label** resource. **addCallback()** adds a callback function to the **callback** list.

An **XtApplicationShell** object wraps an application shell widget from the X toolkit. When it is created, the shell widget is also created and X toolkit options are removed from the main program argument list passed to **new()**. The application main loop is **mainLoop()**.

XawBox, **XawCommand**, **XawForm**, and **XawLabel** objects wrap the corresponding Athena widgets. When they are created, the widgets are also created. **setLabel()** is accepted by **XawCommand** and **XawLabel**. A callback function can be registered with an **XawCommand** object by **addCallback()**.

An **XButton** object is a **Button** object implemented with an **XawCommand** object. It forwards **wire()** to its internal **Button** object and it sets a callback to **gate()** to this button so that it sends its *text* on if **notify()** is executed, i.e., if the button is clicked. Accelerators can be used to arrange for other calls to **notify()**.

An **XLineOut** object is a **LineOut** object implemented with an **XawLabel** object. It forwards **gate()** to itself to receive and display a string. If permitted by the parent widget, its widget will change its size according to the string.

SEE ALSO Event(4)

Bibliography

- [ANSI] *American National Standard for Information Systems — Programming Language C X3.159-1989.*
- [AWK88] A. V. Aho, B. W. Kernighan und P. J. Weinberger *The awk Programming Language* Addison-Wesley 1988, ISBN 0-201-07981-X.
- [Bud91] T. Budd *An Introduction to Object-Oriented Programming* Prentice Hall 1991, ISBN 0-201-54709-0.
- [Ker82] B. W. Kernighan "pic — A Language for Typesetting Graphics" *Software — Practice and Experience* January 1982.
- [K&P84] B. W. Kernighan and R. Pike *The UNIX Programming Environment* Prentice Hall 1984, ISBN 0-13-937681-X.
- [K&R78] B. W. Kernighan and D. M. Ritchie *The C Programming Language* Prentice Hall 1978, ISBN 0-13-110163-3.
- [K&R88] B. W. Kernighan and D. M. Ritchie *The C Programming Language* Second Edition, Prentice Hall 1988, ISBN 0-13-110362-8.
- [Sch87] A. T. Schreiner *UNIX Sprechstunde* Hanser 1987, ISBN 3-446-14894-9.
- [Sch90] A. T. Schreiner *Using C with curses, lex, and yacc* Prentice Hall 1990, ISBN 0-13-932864-5.