Preface

No programming technique solves all problems.
No programming language produces only correct results.
No programmer should start each project from scratch.

Object-oriented programming is the current cure-all — although it has been around for much more then ten years. At the core, there is little more to it then finally applying the good programming principles which we have been taught for more then twenty years. C++ (Eiffel, Oberon-2, Smalltalk ... take your pick) is the New Language because it is object-oriented — although you need not use it that way if you do not want to (or know how to), and it turns out that you can do just as well with plain ANSI-C. Only object-orientation permits code reuse between projects — although the idea of subroutines is as old as computers and good programmers always carried their toolkits and libraries with them.

This book is not going to praise object-oriented programming or condemn the Old Way. We are simply going to use ANSI-C to discover how object-oriented programming is done, what its techniques are, why they help us solve bigger problems, and how we harness generality and program to catch mistakes earlier. Along the way we encounter all the jargon — classes, inheritance, instances, linkage, methods, objects, polymorphisms, and more — but we take it out of the realm of magic and see how it translates into the things we have known and done all along.

I had fun discovering that ANSI-C is a full-scale object-oriented language. To share this fun you need to be reasonably fluent in ANSI-C to begin with — feeling comfortable with structures, pointers, prototypes, and function pointers is a must. Working through the book you will encounter all the newspeak — according to Orwell and Webster a language ''designed to diminish the range of thought'' — and I will try to demonstrate how it merely combines all the good programming principles that you always wanted to employ into a coherent approach. As a result, you may well become a more proficient ANSI-C programmer.

The first six chapters develop the foundations of object-oriented programming with ANSI-C. We start with a careful information hiding technique for abstract data types, add generic functions based on dynamic linkage and inherit code by judicious lengthening of structures. Finally, we put it all together in a class hierarchy that makes code much easier to maintain.

Programming takes discipline. Good programming takes a lot of discipline, a large number of principles, and standard, defensive ways of doing things right. Programmers use tools. Good programmers make tools to dispose of routine tasks once and for all. Object-oriented programming with ANSI-C requires a fair amount of immutable code — names may change but not the structures. Therefore, in chapter seven we build a small preprocessor to create the boilerplate required. It looks like yet another new object-oriented dialect language (*yanoodl* perhaps?) but it should not be viewed as such — it gets the dull parts out of the way and lets us concentrate on the creative aspects of problem solving with better techniques. *ooc*

(sorry) is pliable: we have made it, we understand it and can change it, and it writes the ANSI-C code just like we would.

The following chapters refine our technology. In chapter eight we add dynamic type checking to catch our mistakes earlier on. In chapter nine we arrange for automatic initialization to prevent another class of bugs. Chapter ten introduces delegates and shows how classes and callback functions cooperate to simplify, for example, the constant chore of producing standard main programs. More chapters are concerned with plugging memory leaks by using class methods, storing and loading structured data with a coherent strategy, and disciplined error recovery through a system of nested exception handlers.

Finally, in the last chapter we leave the confines of ANSI-C and implement the obligatory mouse-operated calculator, first for *curses* and then for the X Window System. This example neatly demonstrates how elegantly we can design and implement using objects and classes, even if we have to cope with the idiosyncrasies of foreign libraries and class hierarchies.

Each chapter has a summary where I try to give the more cursory reader a rundown on the happenings in the chapter and their importance for future work. Most chapters suggest some exercises; however, they are not spelled out formally, because I firmly believe that one should experiment on one's own. Because we are building the techniques from scratch, I have refrained from making and using a massive class library, even though some examples could have benefited from it. If you want to understand object-oriented programming, it is more important to first master the techniques and consider your options in code design; dependence on somebody else's library for your developments should come a bit later.

An important part of this book is the enclosed source floppy — it has a DOS file system containing a single shell script to create all the sources arranged by chapter. There is a *ReadMe* file — consult it before you say *make*. It is also quite instructive to use a program like *diff* and trace the evolution of the root classes and *ooc* reports through the later chapters.

The techniques described here grew out of my disenchantment with C++ when I needed object-oriented techniques to implement an interactive programming language and realized that I could not forge a portable implementation in C++. I turned to what I knew, ANSI-C, and I was perfectly able to do what I had to. I have shown this to a number of people in courses and workshops and others have used the methods to get their jobs done. It would have stopped there as my footnote to a fad, if Brian Kernighan and my publishers, Hans-Joachim Niclas and John Wait, had not encouraged me to publish the notes (and in due course to reinvent it all once more). My thanks go to them and to all those who helped with and suffered through the evolution of this book. Last not least I thank my family — and no, object-orientation will not replace sliced bread.

Hollage, October 1993

Axel-Tobias Schreiner

Contents

Preface 5

1 Abstract Data Types — Information Hiding 1

1.1 Data Types 1
1.2 Abstract Data Types 1
1.3 An Example — *Set* 2
1.4 Memory Management 3
1.5 *Object* 3
1.6 An Application 4
1.7 An Implementation — *Set* 4
1.8 Another Implementation — *Bag* 7
1.9 Summary 9
1.10 Exercises 9

2 Dynamic Linkage — Generic Functions 11

2.1 Constructors and Destructors 11
2.2 Methods, Messages, Classes and Objects 12
2.3 Selectors, Dynamic Linkage, and Polymorphisms 13
2.4 An Application 16
2.5 An Implementation — *String* 17
2.6 Another Implementation — *Atom* 18
2.7 Summary 20
2.8 Exercises 20

3 Programming Savvy — Arithmetic Expressions 21

3.1 The Main Loop 21
3.2 The Scanner 22
3.3 The Recognizer 23
3.4 The Processor 23
3.5 Information Hiding 24
3.6 Dynamic Linkage 25
3.7 A Postfix Writer 26
3.8 Arithmetic 28
3.9 Infix Output 28
3.10 Summary 29

4 Inheritance — Code Reuse and Refinement 31

4.1 A Superclass — *Point* 31
4.2 Superclass Implementation — *Point* 32
4.3 Inheritance — *Circle* 33
4.4 Linkage and Inheritance 35
4.5 Static and Dynamic Linkage 36
4.6 Visibility and Access Functions 37
4.7 Subclass Implementation — *Circle* 39

4.8 Summary . . . 40
4.9 Is It or Has It? — Inheritance vs. Aggregates . . . 42
4.10 Multiple Inheritance . . . 42
4.11 Exercises . . . 43

5 Programming Savvy — Symbol Table . . . 45

5.1 Scanning Identifiers . . . 45
5.2 Using Variables . . . 45
5.3 The Screener — *Name* . . . 47
5.4 Superclass Implementation — *Name* . . . 48
5.5 Subclass Implementation — *Var* . . . 50
5.6 Assignment . . . 51
5.7 Another Subclass — Constants . . . 52
5.8 Mathematical Functions — *Math* . . . 52
5.9 Summary . . . 55
5.10 Exercises . . . 55

6 Class Hierarchy — Maintainability . . . 57

6.1 Requirements . . . 57
6.2 Metaclasses . . . 58
6.3 Roots — *Object* and *Class* . . . 59
6.4 Subclassing — *Any* . . . 60
6.5 Implementation — *Object* . . . 62
6.6 Implementation — *Class* . . . 63
6.7 Initialization . . . 65
6.8 Selectors . . . 65
6.9 Superclass Selectors . . . 66
6.10 A New Metaclass — *PointClass* . . . 68
6.11 Summary . . . 70

7 The *ooc* Preprocessor — Enforcing a Coding Standard . . . 73

7.1 *Point* Revisited . . . 73
7.2 Design . . . 78
7.3 Preprocessing . . . 79
7.4 Implementation Strategy . . . 80
7.5 *Object* Revisited . . . 82
7.6 Discussion . . . 84
7.7 An Example — *List*, *Queue*, and *Stack* . . . 85
7.8 Exercises . . . 89

8 Dynamic Type Checking — Defensive Programming . . . 91

8.1 Technique . . . 91
8.2 An Example — *list* . . . 92
8.3 Implementation . . . 94
8.4 Coding Standard . . . 94
8.5 Avoiding Recursion . . . 98
8.6 Summary . . . 100
8.7 Exercises . . . 101

9 Static Construction — Self-Organization 103
9.1 Initialization 103
9.2 Initializer Lists — *munch* 104
9.3 Functions for Objects 106
9.4 Implementation 107
9.5 Summary 109
9.6 Exercises 110

10 Delegates — Callback Functions 111
10.1 Callbacks 111
10.2 Abstract Base Classes 111
10.3 Delegates 113
10.4 An Application Framework — *Filter* 114
10.5 The *respondsTo* Method 117
10.6 Implementation 119
10.7 Another application — *sort* 122
10.8 Summary 123
10.9 Exercises 124

11 Class Methods — Plugging Memory Leaks 125
11.1 An Example 125
11.2 Class Methods 127
11.3 Implementing Class Methods 128
11.4 Programming Savvy — A Classy Calculator 131
11.5 Summary 140
11.6 Exercises 141

12 Persistent Objects — Storing and Loading Data Structures 143
12.1 An Example 143
12.2 Storing Objects — *puto()* 148
12.3 Filling Objects — *geto()* 150
12.4 Loading Objects — *retrieve()* 151
12.5 Attaching Objects — *value* Revisited 153
12.6 Summary 156
12.7 Exercises 157

13 Exceptions — Disciplined Error Recovery 159
13.1 Strategy 159
13.2 Implementation — *Exception* 161
13.3 Examples 163
13.4 Summary 165
13.5 Exercises 166

14 Forwarding Messages — A GUI Calculator 167
14.1 The Idea 167
14.2 Implementation 168
14.3 Object-Oriented Design by Example 171
14.4 Implementation — *lc* 174

14.5 A Character-Based Interface — *curses* . . . 179
14.6 A Graphical Interface — *Xt* . . . 182
14.7 Summary . . . 188
14.8 Exercises . . . 189

A ANSI-C Programming Hints . . . 191

A.1 Names and Scope . . . 191
A.2 Functions . . . 191
A.3 Generic Pointers — *void ** . . . 192
A.4 *const* . . . 193
A.5 *typedef* and *const* . . . 194
A.6 Structures . . . 194
A.7 Pointers to Functions . . . 195
A.8 Preprocessor . . . 196
A.9 Verification — *assert.h* . . . 196
A.10 Global Jumps — *setjmp.h* . . . 196
A.11 Variable Argument Lists — *stdarg.h* . . . 197
A.12 Data Types — *stddef.h* . . . 198
A.13 Memory Management — *stdlib.h* . . . 198
A.14 Memory Functions — *string.h* . . . 198

B The *ooc* Preprocessor — Hints on *awk* Programming . . . 199

B.1 Architecture . . . 199
B.2 File Management — *io.awk* . . . 200
B.3 Recognition — *parse.awk* . . . 200
B.4 The Database . . . 201
B.5 Report Generation — *report.awk* . . . 202
B.6 Line Numbering . . . 203
B.7 The Main Program — *main.awk* . . . 204
B.8 Report Files . . . 204
B.9 The *ooc* Command . . . 205

C Manual . . . 207

C.1 Commands . . . 207
C.2 Functions . . . 214
C.3 Root Classes . . . 214
C.4 GUI Calculator Classes . . . 218

Bibliography . . . 223

1
Abstract Data Types
Information Hiding

1.1 Data Types

Data types are an integral part of every programming language. ANSI-C has **int**, **double** and **char** to name just a few. Programmers are rarely content with what's available and a programming language normally provides facilities to build new data types from those that are predefined. A simple approach is to form aggregates such as arrays, structures, or unions. Pointers, according to C. A. R. Hoare ''a step from which we may never recover,'' permit us to represent and manipulate data of essentially unlimited complexity.

What exactly is a data type? We can take several points of view. A data type is a set of values — **char** typically has 256 distinct values, **int** has many more; both are evenly spaced and behave more or less like the natural numbers or integers of mathematics. **double** once again has many more values, but they certainly do not behave like mathematics' real numbers.

Alternatively, we can define a data type as a set of values plus operations to work with them. Typically, the values are what a computer can represent, and the operations more or less reflect the available hardware instructions. **int** in ANSI-C does not do too well in this respect: the set of values may vary between machines, and operations like arithmetic right shift may behave differently.

More complicated examples do not fare much better. Typically we would define an element of a linear list as a structure

```
typedef struct node {
    struct node * next;
    ... information ...
} node;
```

and for the operations we specify function headers like

```
node * head (node * elt, const node * tail);
```

This approach, however, is quite sloppy. Good programming principles dictate that we conceal the representation of a data item and declare only the possible manipulations.

1.2 Abstract Data Types

We call a data type *abstract*, if we do not reveal its representation to the user. At a theoretical level this requires us to specify the properties of the data type by mathematical axioms involving the possible operations. For example, we can remove an element from a queue only as often as we have added one previously, and we retrieve the elements in the same order in which they were added.

Abstract data types offer great flexibility to the programmer. Since the representation is not part of the definition, we are free to choose whatever is easiest or most efficient to implement. If we manage to distribute the necessary information correctly, use of the data type and our choice of implementation are totally independent.

Abstract data types satisfy the good programming principles of *information hiding* and *divide and conquer*. Information such as the representation of data items is given only to the one with a need to know: to the implementer and not to the user. With an abstract data type we cleanly separate the programming tasks of implementation and usage: we are well on our way to decompose a large system into smaller modules.

1.3 An Example — *Set*

So how do we implement an abstract data type? As an example we consider a set of elements with the operations *add*, *find*, and *drop*.* They all apply to a set and an element and return the element added to, found in, or removed from a set. *find* can be used to implement a condition *contains* which tells us whether an element is already contained in a set.

Viewed this way, set is an abstract data type. To declare what we can do with a set, we start a header file *Set.h*:

```
#ifndef SET_H
#define SET_H

extern const void * Set;

void * add (void * set, const void * element);
void * find (const void * set, const void * element);
void * drop (void * set, const void * element);
int contains (const void * set, const void * element);

#endif
```

The preprocessor statements protect the declarations: no matter how many times we include *Set.h*, the C compiler only sees the declarations once. This technique of protecting header files is so standard, that the GNU C preprocessor recognizes it and does not even access such a file when its protecting symbol is defined.

Set.h is complete, but is it useful? We can hardly reveal or assume less: **Set** will have to somehow represent the fact, that we are working with sets; **add()** takes an element, adds it to a set, and returns whatever was added or already present in the set; **find()** looks for an element in a set and returns whatever is present in the set or a null pointer; **drop()** locates an element, removes it from a set, and returns whatever was removed; **contains()** converts the result of **find()** into a truth value.

---

* Unfortunately, *remove* is an ANSI-C library function to remove a file. If we used this name for a set function, we could no longer include *stdio.h*.

The generic pointer **void *** is used throughout. On the one hand it makes it impossible to discover what a set looks like, but on the other hand it permits us to pass virtually anything to **add()** and the other functions. Not everything will behave like a set or an element — we are sacrificing type security in the interest of information hiding. However, we will see in chapter 8 that this approach can be made completely secure.

1.4 Memory Management

We may have overlooked something: how does one obtain a set? **Set** is a pointer, not a type defined by **typedef**; therefore, we cannot define local or global variables of type **Set**. Instead, we are only going to use pointers to refer to sets and elements, and we declare source and sink of all data items in *new.h*:

```
void * new (const void * type, ...);
void delete (void * item);
```

Just like *Set.h* this file is protected by a preprocessor symbol **NEW_H**. The text only shows the interesting parts of each new file, the source diskette contains the complete code of all examples.

new() accepts a descriptor like **Set** and possibly more arguments for initialization and returns a pointer to a new data item with a representation conforming to the descriptor. **delete()** accepts a pointer originally produced by **new()** and recycles the associated resources.

new() and **delete()** presumably are a frontend to the ANSI-C functions **calloc()** and **free()**. If they are, the descriptor has to indicate at least how much memory is required.

1.5 *Object*

If we want to collect anything interesting in a set, we need another abstract data type **Object** described by the header file *Object.h*:

```
extern const void * Object;     /* new(Object); */

int differ (const void * a, const void * b);
```

differ() can compare objects: it returns true if they are not equal and false if they are. This description leaves room for the functionality of **strcmp()**: for some pairs of objects we might choose to return a negative or positive value to specify an ordering.

Real life objects need more functionality to do something useful. For the moment, we restrict ourselves to the bare necessities for membership in a set. If we built a bigger class library, we would see that a set — and in fact everything else — is an object, too. At this point, a lot of functionality results more or less for free.

1.6 An Application

With the header files, i.e., the definitions of the abstract data types, in place we can write an application *main.c*:

```
#include <stdio.h>

#include "new.h"
#include "Object.h"
#include "Set.h"

int main ()
{   void * s = new(Set);
    void * a = add(s, new(Object));
    void * b = add(s, new(Object));
    void * c = new(Object);

    if (contains(s, a) && contains(s, b))
        puts("ok");

    if (contains(s, c))
        puts("contains?");

    if (differ(a, add(s, a)))
        puts("differ?");

    if (contains(s, drop(s, a)))
        puts("drop?");

    delete(drop(s, b));
    delete(drop(s, c));

    return 0;
}
```

We create a set and add two new objects to it. If all is well, we find the objects in the set and we should not find another new object. The program should simply print **ok**.

The call to **differ()** illustrates a semantic point: a mathematical set can only contain one copy of the object **a**; an attempt to add it again must return the original object and **differ()** ought to be false. Similarly, once we remove the object, it should no longer be in the set.

Removing an element not in a set will result in a null pointer being passed to **delete()**. For now, we stick with the semantics of **free()** and require this to be acceptable.

1.7 An Implementation — *Set*

main.c will compile successfully, but before we can link and execute the program, we must implement the abstract data types and the memory manager. If an object stores no information and if every object belongs to at most one set, we can represent each object and each set as small, unique, positive integer values used as indices into an array **heap[]**. If an object is a member of a set, its array element contains the integer value representing the set. Objects, therefore, point to the set containing them.

This first solution is so simple that we combine all modules into a single file *Set.c*. Sets and objects have the same representation, so **new()** pays no attention to the type description. It only returns an element in **heap[]** with value zero:

```
#if ! defined MANY || MANY < 1
#define MANY    10
#endif

static int heap [MANY];

void * new (const void * type, ...)
{   int * p;                            /* & heap[1..] */

    for (p = heap + 1; p < heap + MANY; ++ p)
        if (! * p)
            break;
    assert(p < heap + MANY);
    * p = MANY;
    return p;
}
```

We use zero to mark available elements of **heap[]**; therefore, we cannot return a reference to **heap[0]** — if it were a set, its elements would contain the index value zero.

Before an object is added to a set, we let it contain the impossible index value **MANY** so that **new()** cannot find it again and we still cannot mistake it as a member of any set.

new() can run out of memory. This is the first of many errors, that ''cannot happen''. We will simply use the ANSI-C macro **assert()** to mark these points. A more realistic implementation should at least print a reasonable error message or use a general function for error handling which the user may overwrite. For our purpose of developing a coding technique, however, we prefer to keep the code uncluttered. In chapter 13 we will look at a general technique for handling exceptions.

delete() has to be careful about null pointers. An element of **heap[]** is recycled by setting it to zero:

```
void delete (void * _item)
{   int * item = _item;

    if (item)
    {   assert(item > heap && item < heap + MANY);
        * item = 0;
    }
}
```

We need a uniform way to deal with generic pointers; therefore, we prefix their names with an underscore and only use them to initialize local variables with the desired types and with the appropriate names.

A set is represented in its objects: each element points to the set. If an element contains **MANY**, it can be added to the set, otherwise, it should already be in the set because we do not permit an object to belong to more than one set.

```
void * add (void * _set, const void * _element)
{   int * set = _set;
    const int * element = _element;

    assert(set > heap && set < heap + MANY);
    assert(* set == MANY);
    assert(element > heap && element < heap + MANY);

    if (* element == MANY)
        * (int *) element = set – heap;
    else
        assert(* element == set – heap);

    return (void *) element;
}
```

assert() takes out a bit of insurance: we would only like to deal with pointers into **heap[]** and the set should not belong to some other set, i.e., its array element value ought to be **MANY**.

The other functions are just as simple. **find()** only looks if its element contains the proper index for the set:

```
void * find (const void * _set, const void * _element)
{   const int * set = _set;
    const int * element = _element;

    assert(set > heap && set < heap + MANY);
    assert(* set == MANY);
    assert(element > heap && element < heap + MANY);
    assert(* element);

    return * element == set – heap ? (void *) element : 0;
}
```

contains() converts the result of **find()** into a truth value:

```
int contains (const void * _set, const void * _element)
{
    return find(_set, _element) != 0;
}
```

drop() can rely on **find()** to check if the element to be dropped actually belongs to the set. If so, we return it to object status by marking it with **MANY**:

```
void * drop (void * _set, const void * _element)
{   int * element = find(_set, _element);

    if (element)
        * element = MANY;
    return element;
}
```

If we were pickier, we could insist that the element to be dropped not belong to another set. In this case, however, we would replicate most of the code of **find()** in **drop()**.

Our implementation is quite unconventional. It turns out that we do not need **differ()** to implement a set. We still need to provide it, because our application uses this function.

```
int differ (const void * a, const void * b)
{
    return a != b;
}
```

Objects differ exactly when the array indices representing them differ, i.e., a simple pointer comparison is sufficient.

We are done — for this solution we have not used the descriptors **Set** and **Object** but we have to define them to keep our C compiler happy:

```
const void * Set;
const void * Object;
```

We did use these pointers in **main()** to create new sets and objects.

1.8 Another Implementation — *Bag*

Without changing the visible interface in *Set.h* we can change the implementation. This time we use dynamic memory and represent sets and objects as structures:

```
struct Set { unsigned count; };
struct Object { unsigned count; struct Set * in; };
```

count keeps track of the number of elements in a set. For an element, **count** records how many times this element has been added to the set. If we decrement **count** each time the element is passed to **drop()** and only remove the element once **count** is zero, we have a *Bag*, i.e., a set where elements have a reference count.

Since we will use dynamic memory to represent sets and objects, we need to initialize the descriptors **Set** and **Object** so that **new()** can find out how much memory to reserve:

```
static const size_t _Set = sizeof(struct Set);
static const size_t _Object = sizeof(struct Object);

const void * Set = & _Set;
const void * Object = & _Object;
```

new() is now much simpler:

```
void * new (const void * type, ...)
{   const size_t size = * (const size_t *) type;
    void * p = calloc(1, size);

    assert(p);
    return p;
}
```

delete() can pass its argument directly to **free()** — in ANSI-C a null pointer may be passed to **free()**.

add() has to more or less believe its pointer arguments. It increments the element's reference counter and the number of elements in the set:

```
void * add (void * _set, const void * _element)
{   struct Set * set = _set;
    struct Object * element = (void *) _element;

    assert(set);
    assert(element);

    if (! element -> in)
        element -> in = set;
    else
        assert(element -> in == set);
    ++ element -> count, ++ set -> count;

    return element;
}
```

find() still checks, if the element points to the appropriate set:

```
void * find (const void * _set, const void * _element)
{   const struct Object * element = _element;

    assert(_set);
    assert(element);

    return element -> in == _set ? (void *) element : 0;
}
```

contains() is based on **find()** and remains unchanged.

If **drop()** finds its element in the set, it decrements the element's reference count and the number of elements in the set. If the reference count reaches zero, the element is removed from the set:

```
void * drop (void * _set, const void * _element)
{   struct Set * set = _set;
    struct Object * element = find(set, _element);

    if (element)
    {   if (-- element -> count == 0)
            element -> in = 0;
        -- set -> count;
    }
    return element;
}
```

We can now provide a new function **count()** which returns the number of elements in a set:

```
unsigned count (const void * _set)
{   const struct Set * set = _set;

    assert(set);
    return set -> count;
}
```

Of course, it would be simpler to let the application read the component **.count** directly, but we insist on not revealing the representation of sets. The overhead of a function call is insignificant compared to the danger of an application being able to overwrite a critical value.

Bags behave differently from sets: an element can be added several times; it will only disappear from the set, once it is dropped as many times as it was added. Our application in section 1.6 added the object **a** twice to the set. After it is dropped from the set once, **contains()** will still find it in the bag. The test program now has the output

```
ok
drop?
```

1.9 Summary

For an abstract data type we completely hide all implementation details, such as the representation of data items, from the application code.

The application code can only access a header file where a descriptor pointer represents the data type and where operations on the data type are declared as functions accepting and returning generic pointers.

The descriptor pointer is passed to a general function **new()** to obtain a pointer to a data item, and this pointer is passed to a general function **delete()** to recycle the associated resources.

Normally, each abstract data type is implemented in a single source file. Ideally, it has no access to the representation of other data types. The descriptor pointer normally points at least to a constant **size_t** value indicating the space requirements of a data item.

1.10 Exercises

If an object can belong to several sets simultaneously, we need a different representation for sets. If we continue to represent objects as small unique integer values, and if we put a ceiling on the number of objects available, we can represent a set as a bitmap stored in a long character string, where a bit selected by the object value is set or cleared depending on the presence of the object in the set.

A more general and more conventional solution represents a set as a linear list of nodes storing the addresses of objects in the set. This imposes no restriction on objects and permits a set to be implemented without knowing the representation of an object.

For debugging it is very helpful to be able to look at individual objects. A reasonably general solution are two functions

```
int store (const void * object, FILE * fp);
int storev (const void * object, va_list ap);
```

store() writes a description of the object to the file pointer. **storev()** uses **va_arg()** to retrieve the file pointer from the argument list pointed to by **ap**. Both functions return the number of characters written. **storev()** is practical if we implement the following function for sets:

```
int apply (const void * set,
    int (* action) (void * object, va_list ap), ...);
```

apply() calls **action()** for each element in **set** and passes the rest of the argument list. **action()** must not change **set** but it may return zero to terminate **apply()** early. **apply()** returns true if all elements were processed.

2
Dynamic Linkage Generic Functions

2.1 Constructors and Destructors

Let us implement a simple string data type which we will later include into a set. For a new string we allocate a dynamic buffer to hold the text. When the string is deleted, we will have to reclaim the buffer.

new() is responsible for creating an object and **delete()** must reclaim the resources it owns. **new()** knows what kind of object it is creating, because it has the description of the object as a first parameter. Based on the parameter, we could use a chain of **if** statements to handle each creation individually. The drawback is that **new()** would explicitly contain code for each data type which we support.

delete(), however, has a bigger problem. It, too, must behave differently based on the type of the object being deleted: for a string the text buffer must be freed; for an object as used in chapter 1 only the object itself has to be reclaimed; and a set may have acquired various chunks of memory to store references to its elements.

We could give **delete()** another parameter: either our type descriptor or the function to do the cleaning up, but this approach is clumsy and error-prone. There is a much more general and elegant way: each object must know how to destroy its own resources. Part of each and every object will be a pointer with which we can locate a clean-up function. We call such a function a *destructor* for the object.

Now **new()** has a problem. It is responsible for creating objects and returning pointers that can be passed to **delete()**, i.e., **new()** must install the destructor information in each object. The obvious approach is to make a pointer to the destructor part of the type descriptor which is passed to **new()**. So far we need something like the following declarations:

```
struct type {
    size_t size;            /* size of an object */
    void (* dtor) (void *); /* destructor */
};

struct String {
    char * text;            /* dynamic string */
    const void * destroy;   /* locate destructor */
};

struct Set {
    ... information ...
    const void * destroy;   /* locate destructor */
};
```

It looks like we have another problem: somebody needs to copy the destructor pointer **dtor** from the type description to **destroy** in the new object and the copy may have to be placed into a different position in each class of objects.

Initialization is part of the job of **new()** and different types require different work — **new()** may even require different arguments for different types:

```
new(Set);                /* make a set */
new(String, "text");    /* make a string */
```

For initialization we use another type-specific function which we will call a *constructor*. Since constructor and destructor are type-specific and do not change, we pass both to **new()** as part of the type description.

Note that constructor and destructor are not responsible for acquiring and releasing the memory for an object itself — this is the job of **new()** and **delete()**. The constructor is called by **new()** and is only responsible for initializing the memory area allocated by **new()**. For a string, this does involve acquiring another piece of memory to store the text, but the space for **struct String** itself is allocated by **new()**. This space is later freed by **delete()**. First, however, **delete()** calls the destructor which essentially reverses the initialization done by the constructor before **delete()** recycles the memory area allocated by **new()**.

2.2 Methods, Messages, Classes and Objects

delete() must be able to locate the destructor without knowing what type of object it has been given. Therefore, revising the declarations shown in section 2.1, we must insist that the pointer used to locate the destructor must be at the beginning of all objects passed to **delete()**, no matter what type they have.

What should this pointer point to? If all we have is the address of an object, this pointer gives us access to type-specific information for the object, such as its destructor function. It seems likely that we will soon invent other type-specific functions such as a function to display objects, or our comparison function **differ()**, or a function **clone()** to create a complete copy of an object. Therefore we will use a pointer to a table of function pointers.

Looking closely, we realize that this table must be part of the type description passed to **new()**, and the obvious solution is to let an object point to the entire type description:

```
struct Class {
    size_t size;
    void * (* ctor) (void * self, va_list * app);
    void * (* dtor) (void * self);
    void * (* clone) (const void * self);
    int (* differ) (const void * self, const void * b);
};
struct String {
    const void * class; /* must be first */
    char * text;
};
```

```
struct Set {
    const void * class; /* must be first */
        ...
};
```

Each of our objects starts with a pointer to its own type description, and through this type description we can locate type-specific information for the object: **.size** is the length that **new()** allocates for the object; **.ctor** points to the constructor called by **new()** which receives the allocated area and the rest of the argument list passed to **new()** originally; **.dtor** points to the destructor called by **delete()** which receives the object to be destroyed; **.clone** points to a copy function which receives the object to be copied; and **.differ** points to a function which compares its object to something else.

Looking down this list, we notice that every function works for the object through which it will be selected. Only the constructor may have to cope with a partially initialized memory area. We call these functions *methods* for the objects. Calling a method is termed a *message* and we have marked the *receiving object* of the message with the parameter name **self**. Since we are using plain C functions, **self** need not be the first parameter.

Many objects will share the same type descriptor, i.e., they need the same amount of memory and the same methods can be applied to them. We call all objects with the same type descriptor a *class*; a single object is called an *instance* of the class. So far a class, an abstract data type, and a set of possible values together with operations, i.e., a data type, are pretty much the same.

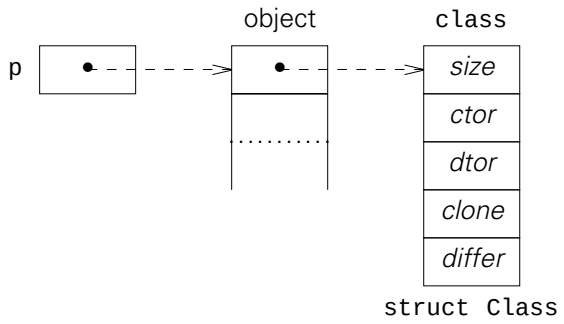
An *object* is an instance of a class, i.e., it has a state represented by the memory allocated by **new()** and the state is manipulated with the methods of its class. Conventionally speaking, an object is a value of a particular data type.

2.3 Selectors, Dynamic Linkage, and Polymorphisms

Who does the messaging? The constructor is called by **new()** for a new memory area which is mostly uninitialized:

```
void * new (const void * _class, ...)
{   const struct Class * class = _class;
    void * p = calloc(1, class -> size);

    assert(p);
    * (const struct Class **) p = class;

    if (class -> ctor)
    {   va_list ap;

        va_start(ap, _class);
        p = class -> ctor(p, & ap);
        va_end(ap);
    }
    return p;
}
```

The existence of the **struct Class** pointer at the beginning of an object is extremely important. This is why we initialize this pointer already in **new()**:



The type description **class** at the right is initialized at compile time. The object is created at run time and the dashed pointers are then inserted. In the assignment

```
* (const struct Class **) p = class;
```

p points to the beginning of the new memory area for the object. We force a conversion of **p** which treats the beginning of the object as a pointer to a **struct Class** and set the argument **class** as the value of this pointer.

Next, if a constructor is part of the type description, we call it and return its result as the result of **new()**, i.e., as the new object. Section 2.6 illustrates that a clever constructor can, therefore, decide on its own memory management.

Note that only explicitly visible functions like **new()** can have a variable parameter list. The list is accessed with a **va_list** variable **ap** which is initialized using the macro **va_start()** from *stdarg.h*. **new()** can only pass the entire list to the constructor; therefore, **.ctor** is declared with a **va_list** parameter and not with its own variable parameter list. Since we might later want to share the original parameters among several functions, we pass the address of **ap** to the constructor — when it returns, **ap** will point to the first argument not consumed by the constructor.

delete() assumes that each object, i.e., each non-null pointer, points to a type description. This is used to call the destructor if any exists. Here, **self** plays the role of **p** in the previous picture. We force the conversion using a local variable **cp** and very carefully thread our way from **self** to its description:

```
void delete (void * self)
{   const struct Class ** cp = self;

    if (self && * cp && (* cp) -> dtor)
        self = (* cp) -> dtor(self);
    free(self);
}
```

The destructor, too, gets a chance to substitute its own pointer to be passed to **free()** by **delete()**. If the constructor decides to cheat, the destructor thus has a chance to correct things, see section 2.6. If an object does not want to be deleted, its destructor would return a null pointer.

All other methods stored in the type description are called in a similar fashion. In each case we have a single receiving object **self** and we need to route the method call through its descriptor:

```
int differ (const void * self, const void * b)
{   const struct Class * const * cp = self;

    assert(self && * cp && (* cp) –> differ);
    return (* cp) –> differ(self, b);
}
```

The critical part is, of course, the assumption that we can find a type description pointer *** self** directly underneath the arbitrary pointer **self**. For the moment at least, we guard against null pointers. We could place a ''magic number'' at the beginning of each type description, or even compare *** self** to the addresses or an address range of all known type descriptions, but we will see in chapter 8 that we can do much more serious checking.

In any case, **differ()** illustrates why this technique of calling functions is called *dynamic linkage* or *late binding*: while we can call **differ()** for arbitrary objects as long as they start with an appropriate type description pointer, the function that actually does the work is determined as late as possible — only during execution of the actual call, not before.

We will call **differ()** a *selector* function. It is an example of a *polymorphic function*, i.e., a function that can accept arguments of different types and act differently on them based on their types. Once we implement more classes which all contain **.differ** in their type descriptors, **differ()** is a *generic function* which can be applied to any object in these classes.

We can view selectors as methods which themselves are not dynamically linked but still behave like polymorphic functions because they let dynamically linked functions do their real work.

Polymorphic functions are actually built into many programming languages, e.g., the procedure **write()** in Pascal handles different argument types differently, and the operator **+** in C has different effects if it is called for integers, pointers, or floating point values. This phenomenon is called *overloading*: argument types and the operator name together determine what the operator does; the same operator name can be used with different argument types to produce different effects.

There is no clear distinction here: because of dynamic linkage, **differ()** behaves like an overloaded function, and the C compiler can make **+** act like a polymorphic function — at least for the built-in data types. However, the C compiler can create different return types for different uses of the operator **+** but the function **differ()** must always have the same return type independent of the types of its arguments.

Methods can be polymorphic without having dynamic linkage. As an example, consider a function **sizeOf()** which returns the size of any object:

```
size_t sizeOf (const void * self)
{   const struct Class * const * cp = self;

    assert(self && * cp);
    return (* cp) –> size;
}
```

All objects carry their descriptor and we can retrieve the size from there. Notice the difference:

```
void * s = new(String, "text");
assert(sizeof s != sizeOf(s));
```

sizeof is a C operator which is evaluated at compile time and returns the number of bytes its argument requires. **sizeOf()** is our polymorphic function which at run time returns the number of bytes of the object, to which the argument points.

2.4 An Application

While we have not yet implemented strings, we are still ready to write a simple test program. *String.h* defines the abstract data type:

```
extern const void * String;
```

All our methods are common to all objects; therefore, we add their declarations to the memory management header file *new.h* introduced in section 1.4:

```
void * clone (const void * self);
int differ (const void * self, const void * b);

size_t sizeOf (const void * self);
```

The first two prototypes declare selectors. They are derived from the corresponding components of **struct Class** by simply removing one indirection from the declarator. Here is the application:

```
#include "String.h"
#include "new.h"

int main ()
{   void * a = new(String, "a"), * aa = clone(a);
    void * b = new(String, "b");

    printf("sizeOf(a) == %u\n", sizeOf(a));
    if (differ(a, b))
        puts("ok");

    if (differ(a, aa))
        puts("differ?");

    if (a == aa)
        puts("clone?");

    delete(a), delete(aa), delete(b);
    return 0;
}
```

We create two strings and make a copy of one. We show the size of a **String** object — not the size of the text controlled by the object — and we check that two different texts result in different strings. Finally, we check that a copy is equal but not identical to its original and we delete the strings again. If all is well, the program will print something like

```
sizeOf(a) == 8
ok
```

2.5 An Implementation — *String*

We implement strings by writing the methods which need to be entered into the type description **String**. Dynamic linkage helps to clearly identify which functions need to be written to implement a new data type.

The constructor retrieves the text passed to **new()** and stores a dynamic copy in the **struct String** which was allocated by **new()**:

```
struct String {
    const void * class; /* must be first */
    char * text;
};

static void * String_ctor (void * _self, va_list * app)
{   struct String * self = _self;
    const char * text = va_arg(* app, const char *);

    self -> text = malloc(strlen(text) + 1);
    assert(self -> text);
    strcpy(self -> text, text);
    return self;
}
```

In the constructor we only need to initialize **.text** because **new()** has already set up **.class**.

The destructor frees the dynamic memory controlled by the string. Since **delete()** can only call the destructor if **self** is not null, we do not need to check things:

```
static void * String_dtor (void * _self)
{   struct String * self = _self;

    free(self -> text), self -> text = 0;
    return self;
}
```

String_clone() makes a copy of a string. Later both, the original and the copy, will be passed to **delete()** so we must make a new dynamic copy of the string's text. This can easily be done by calling **new()**:

```
static void * String_clone (const void * _self)
{   const struct String * self = _self;

    return new(String, self -> text);
}
```

String_differ() is certainly false if we look at identical string objects and it is true if we compare a string with an entirely different object. If we really compare two distinct strings, we try **strcmp()**:

```
static int String_differ (const void * _self, const void * _b)
{   const struct String * self = _self;
    const struct String * b = _b;

    if (self == b)
        return 0;
```

```
    if (! b || b -> class != String)
        return 1;
    return strcmp(self -> text, b -> text);
}
```

Type descriptors are unique — here we use that fact to find out if our second argument really is a string.

All these methods are **static** because they should only be called through **new()**, **delete()**, or the selectors. The methods are made available to the selectors by way of the type descriptor:

```
#include "new.r"

static const struct Class _String = {
    sizeof(struct String),
    String_ctor, String_dtor,
    String_clone, String_differ
};

const void * String = & _String;
```

String.c includes the public declarations in *String.h* and *new.h*. In order to properly initialize the type descriptor, it also includes the private header *new.r* which contains the definition of the representation for **struct Class** shown in section 2.2.

2.6 Another Implementation — *Atom*

To illustrate what we can do with the constructor and destructor interface we implement *atoms*. An atom is a unique string object; if two atoms contain the same strings, they are identical. Atoms are very cheap to compare: **differ()** is true if the two argument pointers differ. Atoms are more expensive to construct and destroy: we maintain a circular list of all atoms and we count the number of times an atom is cloned:

```
struct String {
    const void * class;         /* must be first */
    char * text;
    struct String * next;
    unsigned count;
};

static struct String * ring;    /* of all strings */

static void * String_clone (const void * _self)
{   struct String * self = (void *) _self;

    ++ self -> count;
    return self;
}
```

Our circular list of all atoms is marked in **ring**, extends through the **.next** component, and is maintained by the string constructor and destructor. Before the constructor saves a text it first looks through the list to see if the same text is already stored. The following code is inserted at the beginning of **String_ctor()**:

```
if (ring)
{   struct String * p = ring;

    do
        if (strcmp(p -> text, text) == 0)
        {   ++ p -> count;
            free(self);
            return p;
        }
    while ((p = p -> next) != ring);
}
else
    ring = self;

self -> next = ring -> next, ring -> next = self;
self -> count = 1;
```

If we find a suitable atom, we increment its reference count, free the new string object **self** and return the atom **p** instead. Otherwise we insert the new string object into the circular list and set its reference count to 1.

The destructor prevents deletion of an atom unless its reference count is decremented to zero. The following code is inserted at the beginning of **String_dtor()**:

```
if (-- self -> count > 0)
    return 0;

assert(ring);
if (ring == self)
    ring = self -> next;
if (ring == self)
    ring = 0;
else
{   struct String * p = ring;

    while (p -> next != self)
    {   p = p -> next;
        assert(p != ring);
    }
    p -> next = self -> next;
}
```

If the decremented reference count is positive, we return a null pointer so that **delete()** leaves our object alone. Otherwise we clear the circular list marker if our string is the last one or we remove our string from the list.

With this implementation our application from section 2.4 notices that a cloned string is identical to the original and it prints

```
sizeOf(a) == 16
ok
clone?
```

2.7 Summary

Given a pointer to an object, dynamic linkage lets us find type-specific functions: every object starts with a descriptor which contains pointers to functions applicable to the object. In particular, a descriptor contains a pointer to a constructor which initializes the memory area allocated for the object, and a pointer to a destructor which reclaims resources owned by an object before it is deleted.

We call all objects sharing the same descriptor a class. An object is an instance of a class, type-specific functions for an object are called methods, and messages are calls to such functions. We use selector functions to locate and call dynamically linked methods for an object.

Through selectors and dynamic linkage the same function name will take different actions for different classes. Such a function is called polymorphic.

Polymorphic functions are quite useful. They provide a level of conceptual abstraction: **differ()** will compare any two objects — we need not remember which particular brand of **differ()** is applicable in a concrete situation. A cheap and very convenient debugging tool is a polymorphic function **store()** to display any object on a file descriptor.

2.8 Exercises

To see polymorphic functions in action we need to implement **Object** and **Set** with dynamic linkage. This is difficult for **Set** because we can no longer record in the set elements to which set they belong.

There should be more methods for strings: we need to know the string length, we want to assign a new text value, we should be able to print a string. Things get interesting if we also deal with substrings.

Atoms are much more efficient, if we track them with a hash table. Can the value of an atom be changed?

String_clone() poses an subtle question: in this function **String** should be the same value as **self –> class**. Does it make any difference what we pass to **new()**?

3
Programming Savvy Arithmetic Expressions

Dynamic linkage is a powerful programming technique in its own right. Rather than writing a few functions, each with a big **switch** to handle many special cases, we can write many small functions, one for each case, and arrange for the proper function to be called by dynamic linkage. This often simplifies a routine job and it usually results in code that can be extended easily.

As an example we will write a small program to read and evaluate arithmetic expressions consisting of floating point numbers, parentheses and the usual operators for addition, subtraction, and so on. Normally we would use the compiler generator tools *lex* and *yacc* to build that part of the program which recognizes an arithmetic expression. This book is not about compiler building, however, so just this once we will write this code ourselves.

3.1 The Main Loop

The main loop of the program reads a line from standard input, initializes things so that numbers and operators can be extracted and white space is ignored, calls up a function to recognize a correct arithmetic expression and somehow store it, and finally processes whatever was stored. If things go wrong, we simply read the next input line. Here is the main loop:

```
#include <setjmp.h>

static enum tokens token;   /* current input symbol */

static jmp_buf onError;

int main (void)
{   volatile int errors = 0;
    char buf [BUFSIZ];

    if (setjmp(onError))
        ++ errors;

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

            if (token)
                error("trash after sum");
            process(e);
            delete(e);
        }

    return errors > 0;
}
```

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

    va_start(ap, fmt);
    vfprintf(stderr, fmt, ap), putc('\n', stderr);
    va_end(ap);
    longjmp(onError, 1);
}
```

The error recovery point is defined with **setjmp()**. If **error()** is called somewhere in the program, **longjmp()** continues execution with another return from **setjmp()**. In this case, the result is the value passed to **longjmp()**, the error is counted, and the next input line is read. The exit code of the program reports if any errors were encountered.

3.2 The Scanner

In the main loop, once an input line has been read into **buf[]**, it is passed to **scan()**, which for each call places the next input symbol into the variable **token**. At the end of a line **token** is zero:

```
#include <ctype.h>
#include <errno.h>
#include <stdlib.h>

#include "parse.h"          /* defines NUMBER */

static double number;       /* if NUMBER: numerical value */

static enum tokens scan (const char * buf)
                    /* return token = next input symbol */
{   static const char * bp;

    if (buf)
        bp = buf;           /* new input line */

    while (isspace(* bp))
        ++ bp;
    if (isdigit(* bp) || * bp == '.')
    {   errno = 0;
        token = NUMBER, number = strtod(bp, (char **) & bp);
        if (errno == ERANGE)
            error("bad value: %s", strerror(errno));
    }
    else
        token = * bp ? * bp ++ : 0;
    return token;
}
```

We call **scan()** with the address of an input line or with a null pointer to continue work on the present line. White space is ignored and for a leading digit or decimal point we extract a floating point number with the ANSI-C function **strtod()**. Any other character is returned as is, and we do not advance past a null byte at the end of the input buffer.

The result of **scan()** is stored in the global variable **token** — this simplifies the recognizer. If we have detected a number, we return the unique value **NUMBER** and we make the actual value available in the global variable **number**.

3.3 The Recognizer

At the top level expressions are recognized by the function **sum()** which internally calls on **scan()** and returns a representation that can be manipulated by **process()** and reclaimed by **delete()**.

If we do not use *yacc* we recognize expressions by the method of recursive descent where grammatical rules are translated into equivalent C functions. For example: a sum is a product, followed by zero or more groups, each consisting of an addition operator and another product. A grammatical rule like

```
sum : product { +|– product }...
```

is translated into a C function like

```
void sum (void)
{
    product();
    for (;;)
    {   switch (token) {
        case '+':
        case '–':
            scan(0), product(); continue;
        }
        return;
    }
}
```

There is a C function for each grammatical rule so that rules can call each other. Alternatives are translated into **switch** or **if** statements, iterations in the grammar produce loops in C. The only problem is that we must avoid infinite recursion.

token always contains the next input symbol. If we recognize it, we must call **scan(0)** to advance in the input and store a new symbol in **token**.

3.4 The Processor

How do we process an expression? If we only want to perform simple arithmetic on numerical values, we can extend the recognition functions and compute the result as soon as we recognize the operators and the operands: **sum()** would expect a **double** result from each call to **product()**, perform addition or subtraction as soon as possible, and return the result, again as a **double** function value.

If we want to build a system that can handle more complicated expressions we need to store expressions for later processing. In this case, we can not only perform arithmetic, but we can permit decisions and conditionally evaluate only part of an expression, and we can use stored expressions as user functions within other expressions. All we need is a reasonably general way to represent an expression. The conventional technique is to use a binary tree and store **token** in each node:

```
struct Node {
    enum tokens token;
    struct Node * left, * right;
};
```

This is not very flexible, however. We need to introduce a **union** to create a node in which we can store a numerical value and we waste space in nodes representing unary operators. Additionally, **process()** and **delete()** will contain **switch** statements which grow with every new token which we invent.

3.5 Information Hiding

Applying what we have learned thus far, we do not reveal the structure of a node at all. Instead, we place some declarations in a header file *value.h*:

```
const void * Add;
    ...
void * new (const void * type, ...);
void process (const void * tree);
void delete (void * tree);
```

Now we can code **sum()** as follows:

```
#include "value.h"

static void * sum (void)
{   void * result = product();
    const void * type;

    for (;;)
    {   switch (token) {
        case '+':
            type = Add;
            break;
        case '-':
            type = Sub;
            break;
        default:
            return result;
        }
        scan(0);
        result = new(type, result, product());
    }
}
```

product() has the same architecture as **sum()** and calls on a function **factor()** to recognize numbers, signs, and a sum enclosed in parentheses:

```
static void * sum (void);

static void * factor (void)
{   void * result;

    switch (token) {
    case '+':
        scan(0);
        return factor();
```

```
        case '-':
            scan(0);
            return new(Minus, factor());
        default:
            error("bad factor: '%c' 0x%x", token, token);
        case NUMBER:
            result = new(Value, number);
            break;
        case '(':
            scan(0);
            result = sum();
            if (token != ')')
                error("expecting )");
        }
        scan(0);
        return result;
    }
```

Especially in **factor()** we need to be very careful to maintain the scanner invariant: **token** must always contain the next input symbol. As soon as **token** is consumed we need to call **scan(0)**.

3.6 Dynamic Linkage

The recognizer is complete. *value.h* completely hides the evaluator for arithmetic expressions and at the same time specifies what we have to implement. **new()** takes a description such as **Add** and suitable arguments such as pointers to the operands of the addition and returns a pointer representing the sum:

```
    struct Type {
        void * (* new) (va_list ap);
        double (* exec) (const void * tree);
        void (* delete) (void * tree);
    };

    void * new (const void * type, ...)
    {   va_list ap;
        void * result;

        assert(type && ((struct Type *) type) -> new);

        va_start(ap, type);
        result = ((struct Type *) type) -> new(ap);
        * (const struct Type **) result = type;
        va_end(ap);
        return result;
    }
```

We use dynamic linkage and pass the call to a node-specific routine which, in the case of **Add**, has to create the node and enter the two pointers:

```
    struct Bin {
        const void * type;
        void * left, * right;
    };
```

```
static void * mkBin (va_list ap)
{   struct Bin * node = malloc(sizeof(struct Bin));

    assert(node);
    node -> left = va_arg(ap, void *);
    node -> right = va_arg(ap, void *);
    return node;
}
```

Note that only **mkBin()** knows what node it creates. All we require is that the various nodes start with a pointer for the dynamic linkage. This pointer is entered by **new()** so that **delete()** can reach its node-specific function:

```
void delete (void * tree)
{
    assert(tree && * (struct Type **) tree
        && (* (struct Type **) tree) -> delete);

    (* (struct Type **) tree) -> delete(tree);
}

static void freeBin (void * tree)
{
    delete(((struct Bin *) tree) -> left);
    delete(((struct Bin *) tree) -> right);
    free(tree);
}
```

Dynamic linkage elegantly avoids complicated nodes. **.new()** creates precisely the right node for each type description: binary operators have two descendants, unary operators have one, and a value node only contains the value. **delete()** is a very simple function because each node handles its own destruction: binary operators delete two subtrees and free their own node, unary operators delete only one subtree, and a value node will only free itself. Variables or constants can even remain behind — they simply would do nothing in response to **delete()**.

3.7 A Postfix Writer

So far we have not really decided what **process()** is going to do. If we want to emit a postfix version of the expression, we would add a character string to the **struct Type** to show the actual operator and **process()** would arrange for a single output line indented by a tab:

```
void process (const void * tree)
{
    putchar('\t');
    exec(tree);
    putchar('\n');
}
```

exec() handles the dynamic linkage:

```
static void exec (const void * tree)
{
    assert(tree && * (struct Type **) tree
        && (* (struct Type **) tree) -> exec);

    (* (struct Type **) tree) -> exec(tree);
}
```

and every binary operator is emitted with the following function:

```
static void doBin (const void * tree)
{
    exec(((struct Bin *) tree) -> left);
    exec(((struct Bin *) tree) -> right);
    printf(" %s", (* (struct Type **) tree) -> name);
}
```

The type descriptions tie everything together:

```
static struct Type _Add = { "+", mkBin, doBin, freeBin };
static struct Type _Sub = { "-", mkBin, doBin, freeBin };

const void * Add = & _Add;
const void * Sub = & _Sub;
```

It should be easy to guess how a numerical value is implemented. It is represented as a structure with a **double** information field:

```
struct Val {
    const void * type;
    double value;
};

static void * mkVal (va_list ap)
{   struct Val * node = malloc(sizeof(struct Val));

    assert(node);
    node -> value = va_arg(ap, double);
    return node;
}
```

Processing consists of printing the value:

```
static void doVal (const void * tree)
{
    printf(" %g", ((struct Val *) tree) -> value);
}
```

We are done — there is no subtree to delete, so we can use the library function **free()** directly to delete the value node:

```
static struct Type _Value = { "", mkVal, doVal, free };

const void * Value = & _Value;
```

A unary operator such as **Minus** is left as an exercise.

3.8 Arithmetic

If we want to do arithmetic, we let the execute functions return **double** values to be printed in **process()**:

```
static double exec (const void * tree)
{
    return (* (struct Type **) tree) –> exec(tree);
}

void process (const void * tree)
{
    printf("\t%g\n", exec(tree));
}
```

For each type of node we need one execution function which computes and returns the value for the node. Here are two examples:

```
static double doVal (const void * tree)
{
    return ((struct Val *) tree) –> value;
}

static double doAdd (const void * tree)
{
    return exec(((struct Bin *) tree) –> left) +
            exec(((struct Bin *) tree) –> right);
}

static struct Type _Add = { mkBin, doAdd, freeBin };
static struct Type _Value = { mkVal, doVal, free };

const void * Add = & _Add;
const void * Value = & _Value;
```

3.9 Infix Output

Perhaps the highlight of processing arithmetic expressions is to print them with a minimal number of parentheses. This is usually a bit tricky, depending on who is responsible for emitting the parentheses. In addition to the operator name used for postfix output we add two numbers to the **struct Type**:

```
struct Type {
    const char * name;      /* node's name */
    char rank, rpar;
    void * (* new) (va_list ap);
    void (* exec) (const void * tree, int rank, int par);
    void (* delete) (void * tree);
};
```

.rank is the precedence of the operator, starting with 1 for addition. **.rpar** is set for operators such as subtraction, which require their right operand to be enclosed in parentheses if it uses an operator of equal precedence. As an example consider

```
$ infix
1 + (2 – 3)
    1 + 2 – 3
1 – (2 – 3)
    1 – (2 – 3)
```

This demonstrates that we have the following initialization:

```
static struct Type _Add = {"+", 1, 0, mkBin, doBin, freeBin};
static struct Type _Sub = {"–", 1, 1, mkBin, doBin, freeBin};
```

The tricky part is for a binary node to decide if it must surround itself with parentheses. A binary node such as an addition is given the precedence of its superior and a flag indicating whether parentheses are needed in the case of equal precedence. **doBin()** decides if it will use parentheses:

```
static void doBin (const void * tree, int rank, int par)
{   const struct Type * type = * (struct Type **) tree;

    par = type –> rank < rank
            || (par && type –> rank == rank);

    if (par) putchar('(');
```

If our node has less precedence than its superior, or if we are asked to put up parentheses on equal precedence, we print parentheses. In any case, if our description has **.rpar** set, we require only of our right operand that it put up extra parentheses:

```
    exec(((struct Bin *) tree) –> left, type –> rank, 0);
    printf(" %s ", type –> name);
    exec(((struct Bin *) tree) –> right,
            type –> rank, type –> rpar);
    if (par) putchar(')');
}
```

The remaining printing routines are significantly simpler to write.

3.10 Summary

Three different processors demonstrate the advantages of information hiding. Dynamic linkage has helped to divide a problem into many very simple functions. The resulting program is easily extended — try adding comparisons and an operator like **? :** in C.

4
Inheritance
Code Reuse and Refinement

4.1 A Superclass — *Point*

In this chapter we will start a rudimentary drawing program. Here is a quick test for one of the classes we would like to have:

```
#include "Point.h"
#include "new.h"

int main (int argc, char ** argv)
{   void * p;

    while (* ++ argv)
    {   switch (** argv) {
        case 'p':
            p = new(Point, 1, 2);
            break;
        default:
            continue;
        }
        draw(p);
        move(p, 10, 20);
        draw(p);
        delete(p);
    }
    return 0;
}
```

For each command argument starting with the letter **p** we get a new point which is drawn, moved somewhere, drawn again, and deleted. ANSI-C does not include standard functions for graphics output; however, if we insist on producing a picture we can emit text which Kernighan's *pic* [Ker82] can understand:

```
$ points p
"." at 1,2
"." at 11,22
```

The coordinates do not matter for the test — paraphrasing a commercial and OOspeak: ''the point is the message.''

What can we do with a point? **new()** will produce a point and the constructor expects initial coordinates as further arguments to **new()**. As usual, **delete()** will recycle our point and by convention we will allow for a destructor.

draw() arranges for the point to be displayed. Since we expect to work with other graphical objects — hence the **switch** in the test program — we will provide dynamic linkage for **draw()**.

move() changes the coordinates of a point by the amounts given as arguments. If we implement each graphical object relative to its own reference point, we will be able to move it simply by applying **move()** to this point. Therefore, we should be able to do without dynamic linkage for **move()**.

4.2 Superclass Implementation — *Point*

The abstract data type interface in *Point.h* contains the following:

```
extern const void * Point;          /* new(Point, x, y); */

void move (void * point, int dx, int dy);
```

We can recycle the *new.?* files from chapter 2 except that we remove most methods and add **draw()** to *new.h*:

```
void * new (const void * class, ...);
void delete (void * item);
void draw (const void * self);
```

The type description **struct Class** in *new.r* should correspond to the method declarations in *new.h*:

```
struct Class {
    size_t size;
    void * (* ctor) (void * self, va_list * app);
    void * (* dtor) (void * self);
    void (* draw) (const void * self);
};
```

The selector **draw()** is implemented in *new.c*. It replaces selectors such as **differ()** introduced in section 2.3 and is coded in the same style:

```
void draw (const void * self)
{   const struct Class * const * cp = self;

    assert(self && * cp && (* cp) -> draw);
    (* cp) -> draw(self);
}
```

After these preliminaries we can turn to the real work of writing *Point.c*, the implementation of points. Once again, object-orientation has helped us to identify precisely what we need to do: we have to decide on a representation and implement a constructor, a destructor, the dynamically linked method **draw()** and the statically linked method **move()**, which is just a plain function. If we stick with two-dimensional, Cartesian coordinates, we choose the obvious representation:

```
struct Point {
    const void * class;
    int x, y;               /* coordinates */
};
```

The constructor has to initialize the coordinates **.x** and **.y** — by now absolutely routine:

```
static void * Point_ctor (void * _self, va_list * app)
{   struct Point * self = _self;

    self -> x = va_arg(* app, int);
    self -> y = va_arg(* app, int);
    return self;
}
```

It turns out that we do not need a destructor because we have no resources to reclaim before **delete()** does away with **struct Point** itself. In **Point_draw()** we print the current coordinates in a way which *pic* can understand:

```
static void Point_draw (const void * _self)
{   const struct Point * self = _self;

    printf("\".\" at %d,%d\n", self -> x, self -> y);
}
```

This takes care of all the dynamically linked methods and we can define the type descriptor, where a null pointer represents the non-existing destructor:

```
static const struct Class _Point = {
    sizeof(struct Point), Point_ctor, 0, Point_draw
};

const void * Point = & _Point;
```

move() is not dynamically linked, so we omit **static** to export it from *Point.c* and we do not prefix its name with the class name **Point**:

```
void move (void * _self, int dx, int dy)
{   struct Point * self = _self;

    self -> x += dx, self -> y += dy;
}
```

This concludes the implementation of points in *Point.?* together with the support for dynamic linkage in *new.?*.

4.3 Inheritance — *Circle*

A circle is just a big point: in addition to the center coordinates it needs a radius. Drawing happens a bit differently, but moving only requires that we change the coordinates of the center.

This is where we would normally crank up our text editor and perform source code reuse. We make a copy of the implementation of points and change those parts where a circle is different from a point. **struct Circle** gets an additional component:

```
    int rad;
```

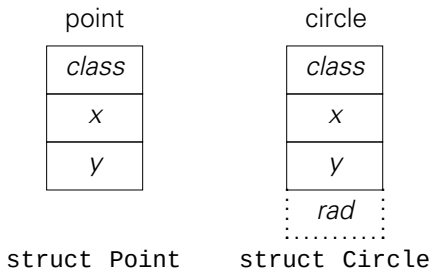
This component is initialized in the constructor

```
    self -> rad = va_arg(* app, int);
```

and used in **Circle_draw()**:

```
printf("circle at %d,%d rad %d\n",
    self -> x, self -> y, self -> rad);
```

We get a bit stuck in **move()**. The necessary actions are identical for a point and a circle: we need to add the displacement arguments to the coordinate components. However, in one case, **move()** works on a **struct Point**, and in the other case, it works on a **struct Circle**. If **move()** were dynamically linked, we could provide two different functions to do the same thing, but there is a much better way. Consider the layout of the representations of points and circles:



The picture shows that every circle begins with a point. If we derive **struct Circle** by adding to the end of **struct Point**, we can pass a circle to **move()** because the initial part of its representation looks just like the point which **move()** expects to receive and which is the only thing that **move()** can change. Here is a sound way to make sure the initial part of a circle always looks like a point:

```
struct Circle { const struct Point _; int rad; };
```

We let the derived structure start with a copy of the base structure that we are extending. Information hiding demands that we should never reach into the base structure directly; therefore, we use an almost invisible underscore as its name and we declare it to be **const** to ward off careless assignments.

This is all there is to *simple inheritance*: a *subclass* is derived from a *superclass* (or *base class*) merely by lengthening the structure that represents an object of the superclass.

Since representation of the subclass object (a circle) starts out like the representation of a superclass object (a point), the circle can always pretend to be a point — at the initial address of the circle's representation there really is a point's representation.

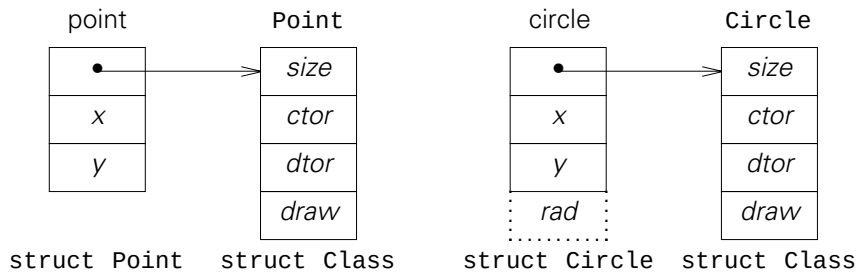
It is perfectly sound to pass a circle to **move()**: the subclass inherits the methods of the superclass because these methods only operate on that part of the subclass' representation that is identical to the superclass' representation for which the methods were originally written. Passing a circle as a point means converting from a **struct Circle *** to a **struct Point ***. We will refer to this as an *up-cast* from a subclass to a superclass — in ANSI-C it can only be accomplished with an explicit conversion operator or through intermediate **void *** values.

It is usually unsound, however, to pass a point to a function intended for circles such as **Circle_draw()**: converting from a **struct Point *** to a **struct Circle *** is only permissible if the point originally was a circle. We will refer to this as a *down-cast* from a superclass to a subclass — this requires explicit conversions or **void *** values, too, and it can only be done to pointers to objects that were in the subclass to begin with.

4.4 Linkage and Inheritance

move() is not dynamically linked and does not use a dynamically linked method to do its work. While we can pass points as well as circles to **move()**, it is not really a polymorphic function: **move()** does not act differently for different kinds of objects, it always adds arguments to coordinates, regardless of what else might be attached to the coordinates.

The situation is different for a dynamically linked method like **draw()**. Let us look at the previous picture again, this time with the type descriptions shown explicitly:



When we up-cast from a circle to a point, we do not change the state of the circle, i.e., even though we look at the circle's **struct Circle** representation as if it were a **struct Point**, we do not change its contents. Consequently, the circle viewed as a point still has **Circle** as a type description because the pointer in its **.class** component has not changed. **draw()** is a selector function, i.e., it will take whatever argument is passed as **self**, proceed to the type description indicated by **.class**, and call the draw method stored there.

A subclass inherits the statically linked methods of its superclass — those methods operate on the part of the subclass object which is already present in the superclass object. A subclass can choose to supply its own methods in place of the dynamically linked methods of its superclass. If inherited, i.e., if not overwritten, the superclass' dynamically linked methods will function just like statically linked methods and modify the superclass part of a subclass object. If overwritten, the subclass' own version of a dynamically linked method has access to the full representation of a subclass object, i.e., for a circle **draw()** will invoke **Circle_draw()** which can consider the radius when drawing the circle.

4.5 Static and Dynamic Linkage

A subclass inherits the statically linked methods of its superclass and it can choose to inherit or overwrite the dynamically linked methods. Consider the declarations for **move()** and **draw()**:

```
void move (void * point, int dx, int dy);
void draw (const void * self);
```

We cannot discover the linkage from the two declarations, although the implementation of **move()** does its work directly, while **draw()** is only the selector function which traces the dynamic linkage at runtime. The only difference is that we declare a statically linked method like **move()** as part of the abstract data type interface in *Point.h*, and we declare a dynamically linked method like **draw()** with the memory management interface in *new.h*, because we have thus far decided to implement the selector function in *new.c*.

Static linkage is more efficient because the C compiler can code a subroutine call with a direct address, but a function like **move()** cannot be overwritten for a subclass. Dynamic linkage is more flexible at the expense of an indirect call — we have decided on the overhead of calling a selector function like **draw()**, checking the arguments, and locating and calling the appropriate method. We could forgo the checking and reduce the overhead with a macro* like

```
#define draw(self) \
    ((* (struct Class **) self) -> draw (self))
```

but macros cause problems if their arguments have side effects and there is no clean technique for manipulating variable argument lists with macros. Additionally, the macro needs the declaration of **struct Class** which we have thus far made available only to class implementations and not to the entire application.

Unfortunately, we pretty much decide things when we design the superclass. While the function calls to the methods do not change, it takes a lot of text editing, possibly in a lot of classes, to switch a function definition from static to dynamic linkage and vice versa. Beginning in chapter 7 we will use a simple preprocessor to simplify coding, but even then linkage switching is error-prone.

In case of doubt it is probably better to decide on dynamic rather than static linkage even if it is less efficient. Generic functions can provide a useful conceptional abstraction and they tend to reduce the number of function names which we need to remember in the course of a project. If, after implementing all required classes, we discover that a dynamically linked method was never overwritten, it is a lot less trouble to replace its selector by its single implementation, and even waste its slot in **struct Class**, than to extend the type description and correct all the initializations.

---

* In ANSI-C macros are not expanded recursively so that a macro may hide a function by the same name.

4.6 Visibility and Access Functions

We can now attempt to implement **Circle_draw()**. Information hiding dictates that we use three files for each class based on a ''need to know'' principle. *Circle.h* contains the abstract data type interface; for a subclass it includes the interface file of the superclass to make declarations for the inherited methods available:

```
#include "Point.h"

extern const void * Circle;     /* new(Circle, x, y, rad) */
```

The interface file *Circle.h* is included by the application code and for the implementation of the class; it is protected from multiple inclusion.

The representation of a circle is declared in a second header file, *Circle.r*. For a subclass it includes the representation file of the superclass so that we can derive the representation of the subclass by extending the superclass:

```
#include "Point.r"

struct Circle { const struct Point _; int rad; };
```

The subclass needs the superclass representation to implement inheritance: **struct Circle** contains a **const struct Point**. The point is certainly not constant — **move()** will change its coordinates — but the **const** qualifier guards against accidentally overwriting the components. The representation file *Circle.r* is only included for the implementation of the class; it is protected from multiple inclusion.

Finally, the implementation of a circle is defined in the source file *Circle.c* which includes the interface and representation files for the class and for object management:

```
#include "Circle.h"
#include "Circle.r"
#include "new.h"
#include "new.r"

static void Circle_draw (const void * _self)
{   const struct Circle * self = _self;

    printf("circle at %d,%d rad %d\n",
        self -> _.x, self -> _.y, self -> rad);
}
```

In **Circle_draw()** we have read point components for the circle by invading the subclass part with the ''invisible name'' _. From an information hiding perspective this is not such a good idea. While reading coordinate values should not create major problems we can never be sure that in other situations a subclass implementation is not going to cheat and modify its superclass part directly, thus potentially playing havoc with its invariants.

Efficiency dictates that a subclass reach into its superclass components directly. Information hiding and maintainability require that a superclass hide its own representation as best as possible from its subclasses. If we opt for the latter, we should provide *access functions* for all those components of a superclass which a subclass is allowed to look at, and *modification functions* for those components, if any, which the subclass may modify.

Access and modification functions are statically linked methods. If we declare them in the representation file for the superclass, which is only included in the implementations of subclasses, we can use macros, because side effects are no problem if a macro uses each argument only once. As an example, in *Point.r* we define the following access macros:*

```
#define x(p)    (((const struct Point *)(p)) -> x)
#define y(p)    (((const struct Point *)(p)) -> y)
```

These macros can be applied to a pointer to any object that starts with a **struct Point**, i.e., to objects from any subclass of our points. The technique is to up-cast the pointer into our superclass and reference the interesting component there. **const** in the cast blocks assignments to the result. If **const** were omitted

```
#define x(p)        (((struct Point *)(p)) -> x)
```

a macro call **x(p)** produces an l-value which can be the target of an assignment. A better modification function would be the macro definition

```
#define set_x(p,v)  (((struct Point *)(p)) -> x = (v))
```

which produces an assignment.

Outside the implementation of a subclass we can only use statically linked methods for access and modification functions. We cannot resort to macros because the internal representation of the superclass is not available for the macros to reference. Information hiding is accomplished by not providing the representation file *Point.r* for inclusion into an application.

The macro definitions demonstrate, however, that as soon as the representation of a class is available, information hiding can be quite easily defeated. Here is a way to conceal **struct Point** much better. Inside the superclass implementation we use the normal definition:

```
struct Point {
    const void * class;
    int x, y;               /* coordinates */
};
```

For subclass implementations we provide the following opaque version:

```
struct Point {
  const char _ [ sizeof( struct {
    const void * class;
    int x, y;               /* coordinates */
  })];
};
```

This structure has the same size as before, but we can neither read nor write the components because they are hidden in an anonymous interior structure. The catch is that both declarations must contain identical component declarations and this is difficult to maintain without a preprocessor.

---

* In ANSI-C, a parametrized macro is only expanded if the macro name appears before a left parenthesis. Elsewhere, the macro name behaves like any other identifier.

4.7 Subclass Implementation — *Circle*

We are ready to write the complete implementation of circles, where we can choose whatever techniques of the previous sections we like best. Object-orientation prescribes that we need a constructor, possibly a destructor, **Circle_draw()**, and a type description **Circle** to tie it all together. In order to exercise our methods, we include *Circle.h* and add the following lines to the switch in the test program in section 4.1:

```
case 'c':
    p = new(Circle, 1, 2, 3);
    break;
```

Now we can observe the following behavior of the test program:

```
$ circles p c
"." at 1,2
"." at 11,22
circle at 1,2 rad 3
circle at 11,22 rad 3
```

The circle constructor receives three arguments: first the coordinates of the circle's point and then the radius. Initializing the point part is the job of the point constructor. It consumes part of the argument list of **new()**. The circle constructor is left with the remaining argument list from which it initializes the radius.

A subclass constructor should first let the superclass constructor do that part of the initialization which turns plain memory into the superclass object. Once the superclass constructor is done, the subclass constructor completes initialization and turns the superclass object into a subclass object.

For circles this means that we need to call **Point_ctor()**. Like all dynamically linked methods, this function is declared **static** and thus hidden inside *Point.c*. However, we can still get to the function by means of the type descriptor **Point** which is available in *Circle.c*:

```
static void * Circle_ctor (void * _self, va_list * app)
{   struct Circle * self =
        ((const struct Class *) Point) -> ctor(_self, app);

    self -> rad = va_arg(* app, int);
    return self;
}
```

It should now be clear why we pass the address **app** of the argument list pointer to each constructor and not the **va_list** value itself: **new()** calls the subclass constructor, which calls its superclass constructor, and so on. The supermost constructor is the first one to actually do something, and it gets first pick at the left end of the argument list passed to **new()**. The remaining arguments are available to the next subclass and so on until the last, rightmost arguments are consumed by the final subclass, i.e., by the constructor directly called by **new()**.

Destruction is best arranged in the exact opposite order: **delete()** calls the subclass destructor. It should destroy its own resources and then call its direct superclass destructor which can destroy the next set of resources and so on. Construc-

tion happens superclass before subclass, destruction happens in reverse, subclass before superclass, circle part before point part. Here, however, nothing needs to be done.

We have worked on **Circle_draw()** before. We use visible components and code the representation file *Point.r* as follows:

```
struct Point {
    const void * class;
    int x, y;               /* coordinates */
};

#define x(p)    (((const struct Point *)(p)) -> x)
#define y(p)    (((const struct Point *)(p)) -> y)
```

Now we can use the access macros for **Circle_draw()**:

```
static void Circle_draw (const void * _self)
{   const struct Circle * self = _self;

    printf("circle at %d,%d rad %d\n",
        x(self), y(self), self -> rad);
}
```

move() has static linkage and is inherited from the implementation of points. We conclude the implementation of circles by defining the type description which is the only globally visible part of *Circle.c*:

```
static const struct Class _Circle = {
    sizeof(struct Circle), Circle_ctor, 0, Circle_draw
};

const void * Circle = & _Circle;
```

While it looks like we have a viable strategy of distributing the program text implementing a class among the interface, representation, and implementation file, the example of points and circles has not exhibited one problem: if a dynamically linked method such as **Point_draw()** is not overwritten in the subclass, the subclass type descriptor needs to point to the function implemented in the superclass. The function name, however, is defined **static** there, so that the selector cannot be circumvented. We shall see a clean solution to this problem in chapter 6. As a stopgap measure, we would avoid the use of **static** in this case, declare the function header only in the subclass implementation file, and use the function name to initialize the type description for the subclass.

4.8 Summary

The objects of a superclass and a subclass are similar but not identical in behavior. Subclass objects normally have a more elaborate state and more methods — they are specialized versions of the superclass objects.

We start the representation of a subclass object with a copy of the representation of a superclass object, i.e., a subclass object is represented by adding components to the end of a superclass object.

A subclass inherits the methods of a superclass: because the beginning of a subclass object looks just like a superclass object, we can up-cast and view a pointer to a subclass object as a pointer to a superclass object which we can pass to a superclass method. To avoid explicit conversions, we declare all method parameters with **void *** as generic pointers.

Inheritance can be viewed as a rudimentary form of polymorphism: a superclass method accepts objects of different types, namely objects of its own class and of all subclasses. However, because the objects all pose as superclass objects, the method only acts on the superclass part of each object, and it would, therefore, not act differently on objects from different classes.

Dynamically linked methods can be inherited from a superclass or overwritten in a subclass — this is determined for the subclass by whatever function pointers are entered into the type description. Therefore, if a dynamically linked method is called for an object, we always reach the method belonging to the object's true class even if the pointer was up-casted to some superclass. If a dynamically linked method is inherited, it can only act on the superclass part of a subclass object, because it does not know of the existence of the subclass. If a method is overwritten, the subclass version can access the entire object, and it can even call its corresponding superclass method through explicit use of the superclass type description.

In particular, constructors should call superclass constructors back to the ultimate ancestor so that each subclass constructor only deals with its own class' extensions to its superclass representation. Each subclass destructor should remove the subclass' resources and then call the superclass destructor and so on to the ultimate ancestor. Construction happens from the ancestor to the final subclass, destruction takes place in the opposite order.

Our strategy has a glitch: in general we should not call dynamically linked methods from a constructor because the object may not be initialized completely. **new()** inserts the final type description into an object before the constructor is called. Therefore, if a constructor calls a dynamically linked method for an object, it will not necessarily reach the method in the same class as the constructor. The safe technique would be for the constructor to call the method by its internal name in the same class, i.e., for points to call **Points_draw()** rather then **draw()**.

To encourage information hiding, we implement a class with three files. The interface file contains the abstract data type description, the representation file contains the structure of an object, and the implementation file contains the code of the methods and initializes the type description. An interface file includes the superclass interface file and is included for the implementation as well as any application. A representation file includes the superclass representation file and is only included for the implementation.

Components of a superclass should not be referenced directly in a subclass. Instead, we can either provide statically linked access and possibly modification methods for each component, or we can add suitable macros to the representation file of the superclass. Functional notation makes it much simpler to use a text edi-

tor or a debugger to scan for possible information leakage or corruption of invariants.

4.9 Is It or Has It? — Inheritance vs. Aggregates

Our representation of a circle contains the representation of a point as the first component of **struct Circle**:

```
struct Circle { const struct Point _; int rad; };
```

However, we have voluntarily decided not to access this component directly. Instead, when we want to inherit we cast up from **Circle** back to **Point** and deal with the initial **struct Point** there.

There is a another way to represent a circle: it can contain a point as an aggregate. We can handle objects only through pointers; therefore, this representation of a circle would look about as follows:

```
struct Circle2 { struct Point * point; int rad; };
```

This circle does not look like a point anymore, i.e., it cannot inherit from **Point** and reuse its methods. It can, however, apply point methods to its point component; it just cannot apply point methods to itself.

If a language has explicit syntax for inheritance, the distinction becomes more apparent. Similar representations could look as follows in C++:

```
struct Circle : Point { int rad; }; // inheritance

struct Circle2 {
    struct Point point; int rad;    // aggregate
};
```

In C++ we do not necessarily have to access objects only as pointers.

Inheritance, i.e., making a subclass from a superclass, and aggregates, i.e., including an object as component of some other object, provide very similar functionality. Which approach to use in a particular design can often be decided by the *is-it-or-has-it?* test: if an object of a new class *is* just like an object of some other class, we should use inheritance to implement the new class; if an object of a new class *has* an object of some other class as part of its state, we should build an aggregate.

As far as our points are concerned, a circle is just a big point, which is why we used inheritance to make circles. A rectangle is an ambiguous example: we can describe it through a reference point and the side lengths, or we can use the endpoints of a diagonal or even three corners. Only with a reference point is a rectangle some sort of fancy point; the other representations lead to aggregates. In our arithmetic expressions we could have used inheritance to get from a unary to a binary operator node, but that would substantially violate the test.

4.10 Multiple Inheritance

Because we are using plain ANSI-C, we cannot hide the fact that inheritance means including a structure at the beginning of another. Up-casting is the key to reusing a

superclass method on objects of a subclass. Up-casting from a circle back to a point is done by casting the address of the beginning of the structure; the value of the address does not change.

If we include two or even more structures in some other structure, and if we are willing to do some address manipulations during up-casting, we could call the result *multiple inheritance*: an object can behave as if it belonged to several other classes. The advantage appears to be that we do not have to design inheritance relationships very carefully — we can quickly throw classes together and inherit whatever seems desirable. The drawback is, obviously, that there have to be address manipulations during up-casting before we can reuse methods of the superclasses.

Things can actually get quite confusing very quickly. Consider a text and a rectangle, each with an inherited reference point. We can throw them together into a button — the only question is if the button should inherit one or two reference points. C++ permits either approach with rather fancy footwork during construction and up-casting.

Our approach of doing everything in ANSI-C has a significant advantage: it does not obscure the fact that inheritance — multiple or otherwise — always happens by inclusion. Inclusion, however, can also be accomplished as an aggregate. It is not at all clear that multiple inheritance does more for the programmer than complicate the language definition and increase the implementation overhead. We will keep things simple and continue with simple inheritance only. Chapter 14 will show that one of the principal uses of multiple inheritance, library merging, can often be realized with aggregates and message forwarding.

4.11 Exercises

Graphics programming offers a lot of opportunities for inheritance: a point and a side length defines a square; a point and a pair of offsets defines a rectangle, a line segment, or an ellipse; a point and an array of offset pairs defines a polygon or even a spline. Before we proceed to all of these classes, we can make smarter points by adding a text, together with a relative position, or by introducing color or other viewing attributes.

Giving **move()** dynamic linkage is difficult but perhaps interesting: locked objects could decide to keep their point of reference fixed and move only their text portion.

Inheritance can be found in many more areas: sets, bags, and other collections such as lists, stacks, queues, etc. are a family of related data types; strings, atoms, and variables with a name and a value are another family.

Superclasses can be used to package algorithms. If we assume the existence of dynamically linked methods to compare and swap elements of a collection of objects based on some positive index, we can implement a superclass containing a sorting algorithm. Subclasses need to implement comparison and swapping of their objects in some array, but they inherit the ability to be sorted.

5
Programming Savvy Symbol Table

Judicious lengthening of a structure, and thus, sharing the functionality of a base structure, can help to avoid cumbersome uses of **union**. Especially in combination with dynamic linkage, we obtain a uniform and perfectly robust way of dealing with diverging information. Once the basic mechanism is in place, a new extended structure can be easily added and the basic code reused.

As an example, we will add keywords, constants, variables, and mathematical functions to the little calculator started in chapter 3. All of these objects live in a symbol table and share the same basic name searching mechanism.

5.1 Scanning Identifiers

In section 3.2 we implemented the function **scan()** which accepts an input line from the main program and hands out one input symbol per call. If we want to introduce keywords, named constants etc., we need to extend **scan()**. Just like floating point numbers, we extract alphanumeric strings for further analysis:

```
#define ALNUM   "ABCDEFGHIJKLMNOPQRSTUVWXYZ" \
                "abcdefghijklmnopqrstuvwxyz" \
                "_" "0123456789"

static enum tokens scan (const char * buf)
{   static const char * bp;
    ...
    if (isdigit(* bp) || * bp == '.')
        ...
    else if (isalpha(* bp) || * bp == '_')
    {   char buf [BUFSIZ];
        int len = strspn(bp, ALNUM);

        if (len >= BUFSIZ)
            error("name too long: %-.10s...", bp);

        strncpy(buf, bp, len), buf[len] = '\0', bp += len;
        token = screen(buf);
    }
    ...
```

Once we have an identifier we let a new function **screen()** decide what its **token** value should be. If necessary, **screen()** will deposit a description of the symbol in a global variable **symbol** which the parser can inspect.

5.2 Using Variables

A variable participates in two operations: its value is used as an operand in an expression, or the value of an expression is assigned to it. The first operation is a simple extension to the **factor()** part of the recognizer shown in section 3.5.

```
static void * factor (void)
{   void * result;
    ...
    switch (token) {
    case VAR:
        result = symbol;
        break;
    ...
```

VAR is a unique value which **screen()** places into **token** when a suitable identifier is found. Additional information about the identifier is placed into the global variable **symbol**. In this case **symbol** contains a node to represent the variable as a leaf in the expression tree. **screen()** either finds the variable in the symbol table or uses the description **Var** to create it.

Recognizing an assignment is a bit more complicated. Our calculator is comfortable to use if we permit two kinds of statements with the following syntax:

```
asgn : sum
     | VAR = asgn
```

Unfortunately, **VAR** can also appear at the left end of a **sum**, i.e., it is not immediately clear how to recognize C-style embedded assignment with our technique of recursive descent.* Because we want to learn how to deal with keywords anyway, we settle for the following grammar:

```
stmt : sum
     | LET VAR = sum
```

This is translated into the following function:

```
static void * stmt (void)
{   void * result;

    switch (token) {
    case LET:
        if (scan(0) != VAR)
            error("bad assignment");
        result = symbol;
        if (scan(0) != '=')
            error("expecting =");
        scan(0);
        return new(Assign, result, sum());
    default:
        return sum();
    }
}
```

In the main program we call **stmt()** in place of **sum()** and our recognizer is ready to handle variables. **Assign** is a new type description for a node which computes the value of a sum and assigns it to a variable.

---

* There is a trick: simply try for a sum. If on return the next input symbol is **=** the sum must be a leaf node for a variable and we can build the assignment.

5.3 The Screener — *Name*

An assignment has the following syntax:

```
stmt : sum
     | LET VAR = sum
```

LET is an example of a keyword. In building the screener we can still decide what identifier will represent **LET**: **scan()** extracts an identifier from the input line and passes it to **screen()** which looks in the symbol table and returns the appropriate value for **token** and, at least for a variable, a node in **symbol**.

The recognizer discards **LET** but it installs the variable as a leaf node in the tree. For other symbols, such as the name of a mathematical function, we may want to apply **new()** to whatever symbol the screener returns in order to get a new node for our tree. Therefore, our symbol table entries should, for the most part, have the same functions with dynamic linkage as our tree nodes.

For a keyword, a **Name** needs to contain the input string and the token value. Later we want to inherit from **Name**; therefore, we define the structure in a representation file *Name.r*:

```
struct Name {           /* base structure */
    const void * type;  /* for dynamic linkage */
    const char * name;  /* may be malloc–ed */
    int token;
};
```

Our symbols never die: it does not matter if their names are constant strings for predefined keywords or dynamically stored strings for user defined variables — we will not reclaim them.

Before we can find a symbol, we need to enter it into the symbol table. This cannot be handled by calling **new(Name, ...)**, because we want to support more complicated symbols than **Name**, and we should hide the symbol table implementation from them. Instead, we provide a function **install()** which takes a **Name** object and inserts it into the symbol table. Here is the symbol table interface file *Name.h*:

```
extern void * symbol;   /* –> last Name found by screen() */

void install (const void * symbol);
int screen (const char * name);
```

The recognizer must insert keywords like **LET** into the symbol table before they can be found by the screener. These keywords can be defined in a constant table of structures — it makes no difference to **install()**. The following function is used to initialize recognition:

```
#include "Name.h"
#include "Name.r"

static void initNames (void)
{   static const struct Name names [] = {
        { 0, "let", LET },
        0 };
    const struct Name *  np;
```

```
        for (np = names; np -> name; ++ np)
            install(np);
    }
```

Note that **names[]**, the table of keywords, need not be sorted. To define **names[]** we use the representation of **Name**, i.e., we include *Name.r*. Since the keyword **LET** is discarded, we provide no dynamically linked methods.

5.4 Superclass Implementation — *Name*

Searching for symbols by name is a standard problem. Unfortunately, the ANSI standard does not define a suitable library function to solve it. **bsearch()** — binary search in a sorted table — comes close, but if we insert a single new symbol we would have to call **qsort()** to set the stage for further searching.

UNIX systems are likely to provide two or three function families to deal with growing tables. **lsearch()** — linear search of an array and adding at the end(!) — is not entirely efficient. **hsearch()** — a hash table for structures consisting of a text and an information pointer — maintains only a single table of fixed size and imposes an awkward structure on the entries. **tsearch()** — a binary tree with arbitrary comparison and deletion — is the most general family but quite inefficient if the initial symbols are installed from a sorted sequence.

On a UNIX system, **tsearch()** is probably the best compromise. The source code for a portable implementation with binary threaded trees can be found in [Sch87]. However, if this family is not available, or if we cannot guarantee a random initialization, we should look for a simpler facility to implement. It turns out that a careful implementation of **bsearch()** can very easily be extended to support insertion into a sorted array:

```
void * binary (const void * key,
    void * _base, size_t * nelp, size_t width,
    int (* cmp) (const void * key, const void * elt))
{   size_t nel = * nelp;
#define base    (* (char **) & _base)
    char * lim = base + nel * width, * high;

    if (nel > 0)
    {   for (high = lim - width; base <= high; nel >>= 1)
        {   char * mid = base + (nel >> 1) * width;
            int c = cmp(key, mid);

            if (c < 0)
                high = mid - width;
            else if (c > 0)
                base = mid + width, -- nel;
            else
                return (void *) mid;
        }
```

Up to here, this is the standard binary search in an arbitrary array. **key** points to the object to be found; **base** initially is the start address of a table of ***nelp** elements,

each with **width** bytes; and **cmp** is a function to compare **key** to a table element. At this point we have either found a table element and returned its address, or **base** is now the address where **key** should be in the table. We continue as follows:

```
            memmove(base + width, base, lim – base);
        }
        ++ *nelp;
        return memcpy(base, key, width);
    #undef  base
    }
```

memmove() shifts the end of the array out of the way* and **memcpy()** inserts **key**. We assume that there is room beyond the array and we record through **nelp** that we have added an element — **binary()** differs from the standard function **bsearch()** only in requiring the address rather than the value of the variable containing the number of elements in the table.

Given a general means of search and entry, we can easily manage our symbol table. First we need to compare a key to a table element:

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

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

As a key, we pass only the address of a pointer to the text of an input symbol. The table elements are, of course, **Name** structures, and we look only at their **.name** component.

Searching or entering is accomplished by calling **binary()** with suitable parameters. Since we do not know the number of symbols in advance, we make sure that there is always room for the table to expand:

```
    static struct Name ** search (const char ** name)
    {   static const struct Name ** names;  /* dynamic table */
        static size_t used, max;

        if (used >= max)
        {   names = names
                ? realloc(names, (max *= 2) * sizeof * names)
                : malloc((max = NAMES) * sizeof * names);
            assert(names);
        }
        return binary(name, names, & used, sizeof * names, cmp);
    }
```

NAMES is a defined constant with the initial allotment of table entries; each time we run out, we double the size of the table.

search() takes the address of a pointer to the text to be found and returns the address of the table entry. If the text could not be found in the table, **binary()** has

---

* **memmove()** copies bytes even if source and target area overlap; **memcpy()** does not, but it is more efficient.

inserted the key — i.e., only the pointer to the text, not a **struct Name** — into the table. This strategy is for the benefit of **screen()**, which only builds a new table element if an identifier from the input is really unknown:

```
int screen (const char * name)
{   struct Name ** pp = search(& name);

    if (* pp == (void *) name)  /* entered name */
        * pp = new(Var, name);
    symbol = * pp;
    return (* pp) -> token;
}
```

screen() lets **search()** look for the input symbol to be screened. If the pointer to the text of the symbol is entered into the symbol table, we need to replace it by an entry describing the new identifier.

For **screen()**, a new identifier must be a variable. We assume that there is a type description **Var** which knows how to construct **Name** structures describing variables and we let **new()** do the rest. In any case, we let **symbol** point to the symbol table entry and we return its **.token** value.

```
void install (const void * np)
{   const char * name = ((struct Name *) np) -> name;
    struct Name ** pp = search(& name);

    if (* pp != (void *) name)
        error("cannot install name twice: %s", name);
    * pp = (struct Name *) np;
}
```

install() is a bit simpler. We accept a **Name** object and let **search()** find it in the symbol table. **install()** is supposed to deal only with new symbols, so we should always be able to enter the object in place of its name. Otherwise, if **search()** really finds a symbol, we are in trouble.

5.5 Subclass Implementation — *Var*

screen() calls **new()** to create a new variable symbol and returns it to the recognizer which inserts it into an expression tree. Therefore, **Var** must create symbol table entries that can act like nodes, i.e., when defining **struct Var** we need to extend a **struct Name** to inherit the ability to live in the symbol table and we must support the dynamically linked functions applicable to expression nodes. We describe the interface in *Var.h*:

```
const void * Var;
const void * Assign;
```

A variable has a name and a value. If we evaluate an arithmetic expression, we need to return the **.value** component. If we delete an expression, we must not delete the variable node, because it lives in the symbol table:

```
struct Var {    struct Name _; double value; };

#define value(tree) (((struct Var *) tree) -> value)
```

```
static double doVar (const void * tree)
{
    return value(tree);
}

static void freeVar (void * tree)
{
}
```

As discussed in section 4.6 the code is simplified by providing an access function for the value.

Creating a variable requires allocating a **struct Var**, inserting a dynamic copy of the variable name, and the token value **VAR** prescribed by the recognizer:

```
static void * mkVar (va_list ap)
{   struct Var * node = calloc(1, sizeof(struct Var));
    const char * name = va_arg(ap, const char *);
    size_t len = strlen(name);

    assert(node);
    node -> _.name = malloc(len+1);
    assert(node -> _.name);
    strcpy((void *) node -> _.name, name);
    node -> _.token = VAR;
    return node;
}

static struct Type _Var = { mkVar, doVar, freeVar };

const void * Var = & _Var;
```

new() takes care of inserting the type description **Var** into the node before the symbol is returned to **screen()** or to whoever wants to use it.

Technically, **mkVar()** is the constructor for **Name**. However, only variable names need to be stored dynamically. Because we decided that in our calculator the constructor is responsible for allocating an object, we cannot let the **Var** constructor call a **Name** constructor to maintain the **.name** and **.token** components — a **Name** constructor would allocate a **struct Name** rather than a **struct Var**.

5.6 Assignment

Assignment is a binary operation. The recognizer guarantees that we have a variable as a left operand and a sum as a right operand. Therefore, all we really need to implement is the actual assignment operation, i.e., the function dynamically linked into the **.exec** component of the type description:

```
#include "value.h"
#include "value.r"

static double doAssign (const void * tree)
{
    return value(left(tree)) = exec(right(tree));
}
```

```
static struct Type _Assign = { mkBin, doAssign, freeBin };
const void * Assign = & _Assign;
```

We share the constructor and destructor for **Bin** which, therefore, must be made global in the implementation of the arithmetic operations. We also share **struct Bin** and the access functions **left()** and **right()**. All of this is exported with the interface file *value.h* and the representation file *value.r*. Our own access function **value()** for **struct Var** deliberately permits modification so that assignment is quite elegant to implement.

5.7 Another Subclass — Constants

Who likes to type the value of π or other mathematical constants? We take a clue from Kernighan and Pike's *hoc* [K&P84] and predefine some constants for our calculator. The following function needs to be called during the initialization of the recognizer:

```
void initConst (void)
{   static const struct Var constants [] = {    /* like hoc */
        { &_Var, "PI",    CONST,  3.14159265358979323846 },
        ...
        0 };
    const struct Var * vp;
    for (vp = constants; vp -> _.name; ++ vp)
        install(vp);
}
```

Variables and constants are almost the same: both have names and values and live in the symbol table; both return their value for use in an arithmetic expression; and both should not be deleted when we delete an arithmetic expression. However, we should not assign to constants, so we need to agree on a new token value **CONST** which the recognizer accepts in **factor()** just like **VAR**, but which is not permitted on the left hand side of an assignment in **stmt()**.

5.8 Mathematical Functions — *Math*

ANSI-C defines a number of mathematical functions such as **sin()**, **sqrt()**, **exp()**, etc. As another exercise in inheritance, we are going to add library functions with a single **double** parameter and a **double** result to our calculator.

These functions work pretty much like unary operators. We could define a new type of node for each function and collect most of the functionality from **Minus** and the **Name** class, but there is an easier way. We extend **struct Name** into **struct Math** as follows:

```
struct Math { struct Name _;
    double (* funct) (double);
};
#define funct(tree) (((struct Math *) left(tree)) -> funct)
```

In addition to the function name to be used in the input and the token for recognition we store the address of a library function like **sin()** in the symbol table entry.

During initialization we call the following function to enter all the function descriptions into the symbol table:

```
#include <math.h>

void initMath (void)
{   static const struct Math functions [] = {
        { &_Math, "sqrt",  MATH,  sqrt },
        ...
        0 };

    const struct Math * mp;

    for (mp = functions; mp -> _.name; ++ mp)
        install(mp);
}
```

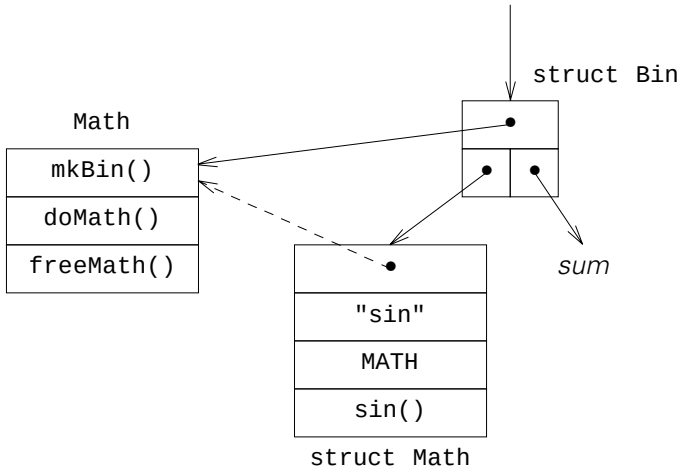
A function call is a factor just like using a minus sign. For recognition we need to extend our grammar for factors:

```
factor : NUMBER
    | - factor
    | ...
    | MATH ( sum )
```

MATH is the common token for all functions entered by **initMath()**. This translates into the following addition to **factor()** in the recognizer:

```
static void * factor (void)
{   void * result;
    ...
    switch (token) {
    case MATH:
    {   const struct Name * fp = symbol;

        if (scan(0) != '(')
            error("expecting (");
        scan(0);
        result = new(Math, fp, sum());
        if (token != ')')
            error("expecting )");
        break;
    }
```

symbol first contains the symbol table element for a function like **sin()**. We save the pointer and build the expression tree for the function argument by calling **sum()**. Then we use **Math**, the type description for the function, and let **new()** build the following node for the expression tree:



We let the left side of a binary node point to the symbol table element for the function and we attach the argument tree at the right. The binary node has **Math** as a type description, i.e., the methods **doMath()** and **freeMath()** will be called to execute and delete the node, respectively.

The **Math** node is still constructed with **mkBin()** because this function does not care what pointers it enters as descendants. **freeMath()**, however, may only delete the right subtree:

```
static void freeMath (void * tree)
{
    delete(right(tree));
    free(tree);
}
```

If we look carefully at the picture, we can see that execution of a **Math** node is very easy. **doMath()** needs to call whatever function is stored in the symbol table element accessible as the left descendant of the binary node from which it is called:

```
#include <errno.h>

static double doMath (const void * tree)
{   double result = exec(right(tree));

    errno = 0;
    result = funct(tree)(result);
    if (errno)
        error("error in %s: %s",
            ((struct Math *) left(tree)) -> _.name,
            strerror(errno));
    return result;
}
```

The only problem is to catch numerical errors by monitoring the **errno** variable declared in the ANSI-C header file *errno.h*. This completes the implementation of mathematical functions for the calculator.

5.9 Summary

Based on a function **binary()** for searching and inserting into a sorted array, we have implemented a symbol table containing structures with a name and a token value. Inheritance permitted us to insert other structures into the table without changing the functions for search and insertion. The elegance of this approach becomes apparent once we consider a conventional definition of a symbol table element for our purposes:

```
struct {
    const char * name;
    int token;
    union {                     /* based on token */
        double value;
        double (* funct) (double);
    } u;
};
```

For keywords, the **union** is unnecessary. User defined functions would require a much more elaborate description, and referencing parts of the **union** is cumbersome.

Inheritance permits us to apply the symbol table functionality to new entries without changing existing code at all. Dynamic linkage helps in many ways to keep the implementation simple: symbol table elements for constants, variables, and functions can be linked into the expression tree without fear that we delete them inadvertently; an execution function concerns itself only with its own arrangement of nodes.

5.10 Exercises

New keywords are necessary to implement things like **while** or **repeat** loops, **if** statements, etc. Recognition is handled in **stmt()**, but this is, for the most part, only a problem of compiler construction, not of inheritance. Once we have decided on the type of statement, we will build node types like **While**, **Repeat**, or **IfElse**, and the keywords in the symbol table need not know of their existence.

A bit more interesting are functions with two arguments like **atan2()** in the mathematical library of ANSI-C. From the point of view of the symbol table, the functions are handled just like simple functions, but for the expression tree we need to invent a new node type with three descendants.

User defined functions pose a really interesting problem. This is not too hard if we represent a single parameter by **$** and use a node type **Parm** to point back to the function entry in the symbol table where we can temporarily store the argument value as long as we do not permit recursion. Functions with parameter names and several parameters are more difficult, of course. However, this is a good exercise to investigate the benefits of inheritance and dynamic linkage. We shall return to this problem in chapter 11.

6
Class Hierarchy Maintainability

6.1 Requirements

Inheritance lets us evolve general data types into more specialized ones and spares us recoding basic functionality. Dynamic Linkage helps us repair the shortcomings that a more general data type might have. What we still need is a clean global organization to simplify maintaining a larger system of classes:

(1) all dynamic links have to point to the correct methods — e.g., a constructor should not be inserted in the wrong place in a class description;

(2) we need a coherent way to add, remove, or change the order of dynamically linked methods for a superclass while guaranteeing correct inheritance to its subclasses;

(3) there should be no loopholes such as missing dynamic links or undefined methods;

(4) if we inherit a dynamically linked method, the implementation of the superclass from which we inherit must remain absolutely unchanged, i.e., inheritance must be possible using binary information only;

(5) different sets of classes should be able to have different sets of dynamically linked methods — e.g., only **Point** and **Circle** from chapter 4, but not the sets from chapter 1 or the expression nodes from chapter 3 and 5, have a use for a **draw()** method.

Mostly, this list indicates that maintaining dynamic linkage is difficult and error-prone — if we cannot substantially improve the situation we may well have created a white elephant.

So far we have worked with a single list of dynamically linked methods, regardless of whether or not it made sense for a particular class. The list was defined as **struct Class** and it was included wherever dynamic linkage needed to be initialized. Thanks to function prototypes, ANSI-C will check that function names like **Point_ctor** fit the slots in the class description, where they are used as static initializers. (1) above is only a problem if several methods have type compatible interfaces or if we change **struct Class** and do a sloppy recompilation.

Item (2), changing **struct Class**, sounds like a nightmare — we need to manually access every class implementation to update the static initialization of the class description, and we can easily forget to add a new method in some class, thus causing problem (3).

We had an elegant way to add assignment to the calculator in section 5.6: we changed the source code and made the dynamically linked methods for binary nodes from section 3.6 public so that we could reuse them as initializers for the **Assign** description, but this clearly violates requirement (4).

If maintaining a single **struct Class** sounds like a challenge already, (5) above suggests that we should have different versions of **struct Class** for different sets of classes! The requirement is perfectly reasonable, however: every class needs a constructor and a destructor; for points, circles, and other graphical objects we add drawing facilities; atoms and strings need comparisons; collections like sets, bags, or lists have methods to add, find, and remove objects; and so on.

6.2 Metaclasses

It turns out that requirement (5) does not compound our problems — it actually points the way to solving them. Just like a circle adds information to a point, so do the class descriptions for points and circles together add information — a polymorphic **draw()** — to the class description for both of these two classes.

Put differently: As long as two classes have the same dynamically linked methods, albeit with different implementations, they can use the same **struct Class** to store the links — this is the case for **Point** and **Circle**. Once we add another dynamically linked method, we need to lengthen **struct Class** to provide room for the new link — this is how we get from a class with only a constructor and a destructor to a class like **Point** with a **.draw** component thrown in.

Lengthening structures is what we called inheritance, i.e., we discover that class descriptions with the same set of methods form a class, and that there is inheritance among the classes of class descriptions!

We call a class of class descriptions a *metaclass*. A metaclass behaves just like a class: **Point** and **Circle**, the descriptions for all points and all circles, are two objects in a metaclass **PointClass**, because they can both describe how to draw. A metaclass has methods: we can ask an object like **Point** or **Circle** for the size of the objects, points or circles, that it describes, or we could ask the object **Circle** if **Point**, indeed, describes the superclass of the circles.

Dynamically linked methods can do different things for objects from different classes. Does a metaclass need dynamically linked methods? The destructor in **PointClass** would be called as a consequence of **delete(Point)** or **delete(Circle)**, i.e., when we try to eliminate the class description for points or circles. This destructor ought to return a null pointer because it is clearly not a good idea to eliminate a class description. A metaclass constructor is much more useful:

```
Circle = new(PointClass,     /* ask the metaclass */
    "Circle",                /* to make a class description */
    Point,                   /* with this superclass, */
    sizeof(struct Circle),   /* this size for the objects, */
    ctor, Circle_ctor,       /* this constructor, */
    draw, Circle_draw,       /* and this drawing method. */
    0);                      /* end of list */
```

This call should produce a class description for a class whose objects can be constructed, destroyed, and drawn. Because drawing is the new idea common to all class descriptions in **PointClass**, it seems only reasonable to expect that the **PointClass** constructor would at least know how to deposit a link to a drawing method in the new description.

Even more is possible: if we pass the superclass description **Point** to the **PointClass** constructor, it should be able to first copy all the inherited links from **Point** to **Circle** and then overwrite those which are redefined for **Circle**. This, however, completely solves the problem of binary inheritance: when we create **Circle** we only specify the new methods specific to circles; methods for points are implicitly inherited because their addresses can be copied by the **PointClass** constructor.

6.3 Roots — *Object* and *Class*

Class descriptions with the same set of methods are the objects of a metaclass. A metaclass as such is a class and, therefore, has a class description. We must assume that the class descriptions for metaclasses once again are objects of meta (metameta?) classes, which in turn are classes and ...

It seems unwise to continue this train of thought. Instead, let us start with the most trivial objects imaginable. We define a class **Object** with the ability to create, destroy, compare, and display objects.

Interface *Object.h*:

```
extern const void * Object;     /* new(Object); */

void * new (const void * class, ...);
void delete (void * self);

int differ (const void * self, const void * b);
int puto (const void * self, FILE * fp);
```

Representation *Object.r*:

```
struct Object {
    const struct Class * class; /* object's description */
};
```

Next we define the representation for the class description for objects, i.e., the structure to which the component **.class** in **struct Object** for our trivial objects points. Both structures are needed in the same places, so we add to *Object.h*:

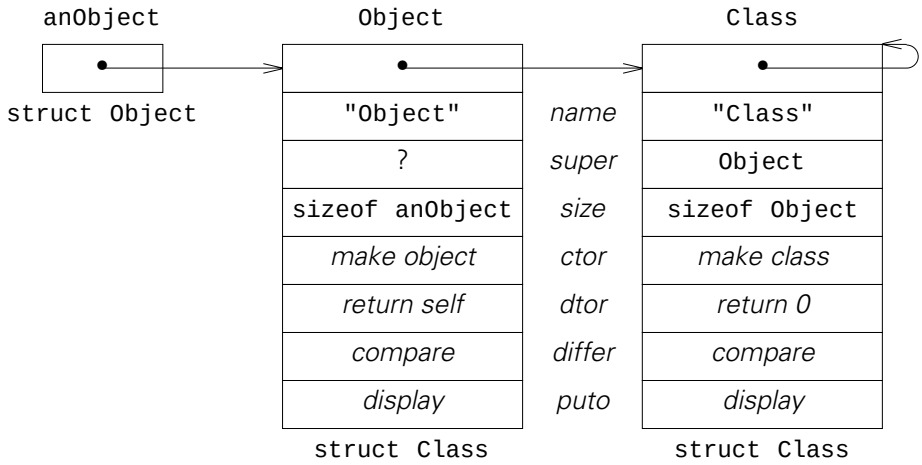
```
extern const void * Class;  /* new(Class, "name", super, size
                                          sel, meth, ... 0); */
```

and to *Object.r*:

```
struct Class {
    const struct Object _;          /* class' description */
    const char * name;              /* class' name */
    const struct Class * super;     /* class' super class */
    size_t size;                    /* class' object's size */
    void * (* ctor) (void * self, va_list * app);
    void * (* dtor) (void * self);
    int (* differ) (const void * self, const void * b);
    int (* puto) (const void * self, FILE * fp);
};
```

struct Class is the representation of each element of the first metaclass **Class**. This metaclass is a class; therefore, its elements point to a class description. Pointing to a class description is exactly what an **Object** can do, i.e., **struct Class** extends **struct Object**, i.e., **Class** is a subclass of **Object**!

This does not cause grief: objects, i.e., instances of the class **Object**, can be created, destroyed, compared, and displayed. We have decided that we want to create class descriptions, and we can write a destructor that silently prevents that a class description is destroyed. It may be quite useful to be able to compare and display class descriptions. However, this means that the metaclass **Class** has the same set of methods, and therefore the same type of description, as the class **Object**, i.e., the chain from objects to their class description and from there to the description of the class description ends right there. Properly initialized, we end up with the following picture:



The question mark indicates one rather arbitrary decision: does **Object** have a superclass or not? It makes no real difference, but for the sake of uniformity we define **Object** to be its own superclass, i.e., the question mark in the picture is replaced by a pointer to **Object** itself.

6.4 Subclassing — *Any*

Given the descriptions **Class** and **Object**, we can already make new objects and even a new subclass. As an example, consider a subclass **Any** which claims that all its objects are equal to any other object, i.e., **Any** overwrites **differ()** to always return zero. Here is the implementation of **Any**, and a quick test, all in one file *any.c*:

```
#include "Object.h"

static int Any_differ (const void * _self, const void * b)
{
    return 0;   /* Any equals anything... */
}
```

```
int main ()
{   void * o = new(Object);
    const void * Any =
                    new(Class, "Any", Object, sizeOf(o),
                        differ, Any_differ,
                        0);
    void * a = new(Any);

    puto(Any, stdout);
    puto(o, stdout);
    puto(a, stdout);

    if (differ(o, o) == differ(a, a))
        puts("ok");

    if (differ(o, a) != differ(a, o))
        puts("not commutative");

    delete(o), delete(a);
    delete(Any);

    return 0;
}
```

If we implement a new class we need to include the interface of its superclass. **Any** has the same representation as **Object** and the class is so simple that we do not even need to include the superclass representation file. The class description **Any** is created by requesting a new instance from its metaclass **Class** and constructing it with the new class name, the superclass description, and the size of an object of the new class:

```
const void * Any =
                    new(Class, "Any", Object, sizeOf(o),
                        differ, Any_differ,
                        0);
```

Additionally, we specify exactly those dynamically linked methods, which we overwrite for the new class. The method names can appear in any order, each is preceded by its selector name. A zero terminates the list.

The program generates one instance **o** of **Object** and one instance **a** of **Any**, and displays the new class description and the two instances. Either instance cannot differ from itself, so the program prints **ok**. The method **differ()** has been overwritten for **Any**; therefore, we get different results if we compare **o** to **a**, and vice versa:

```
$ any
Class at 0x101fc
Object at 0x101f4
Any at 0x10220
ok
not commutative
Any: cannot destroy class
```

Clearly, we should not be able to delete a class description. This error is already detected during compilation, because **delete()** does not accept a pointer to an area protected with **const**.

6.5 Implementation — *Object*

Implementing the **Object** class is straightforward: the constructor and destructor return **self**, and **differ()** checks if its two argument pointers are equal. Defining these trivial implementations is very important, however: we use a single tree of classes and make **Object** the ultimate superclass of every other class; if a class does not overwrite a method such as **differ()** it inherits it from **Object**, i.e., every class has at least a rudimentary definition for every dynamically linked method already applicable to **Object**.

This is a general safety principle: whenever we introduce a new dynamically linked method, we will immediately implement it for its first class. In this fashion we can never be caught selecting a totally undefined method. A case in point is the **puto()** method for **Object**:

```
static int Object_puto (const void * _self, FILE * fp)
{   const struct Class * class = classOf(_self);

    return fprintf(fp, "%s at %p\n", class -> name, _self);
}
```

Every object points to a class description and we have stored the class name with the description. Therefore, for any object we can at least display the class name and the address of the object. The first three lines of output from the trivial test program in section 6.4 indicate that we have not bothered to overwrite this method for **Class** or **Any**.

puto() relies on an access function **classOf()** which does some safety checks and returns the class descriptor for an object:

```
const void * classOf (const void * _self)
{   const struct Object * self = _self;

    assert(self && self -> class);
    return self -> class;
}
```

Similarly, we can ask an object for its size* — remember that, technically, an object is a plain **void *** in ANSI-C:

```
size_t sizeOf (const void * _self)
{   const struct Class * class = classOf(_self);

    return class -> size;
}
```

It is debatable if we should ask the object for the size, or if we should only ask it for the class and then explicitly ask the class for the size. If we implement **sizeOf()** for

* The spelling is likely to be error-prone, but I just could not resist the pun. Inventing good method names is an art.

objects, we cannot apply it to a class description to get the corresponding object size — we will get the size of the class description itself. However, practical use indicates that defining **sizeOf()** for objects is preferable. In contrast, **super()** is a statically linked method which returns the superclass of a class, not of an object.

6.6 Implementation — *Class*

Class is a subclass of **Object**, so we can simply inherit the methods for comparison and display. The destructor returns a null pointer to keep **delete()** from actually reclaiming the space occupied by a class description:

```
static void * Class_dtor (void * _self)
{   struct Class * self = _self;

    fprintf(stderr, "%s: cannot destroy class\n", self->name);
    return 0;
}
```

Here is the access function to get the superclass from a class description:

```
const void * super (const void * _self)
{   const struct Class * self = _self;

    assert(self && self -> super);
    return self -> super;
}
```

The only difficult part is the implementation of the **Class** constructor because this is where a new class description is initialized, where inheritance takes place, and where our four basic methods can be overwritten. We recall from section 6.4 how a new class description is created:

```
const void * Any =
                new(Class, "Any", Object, sizeOf(o),
                    differ, Any_differ,
                    0);
```

This means that our **Class** constructor receives the name, superclass, and object size for a new class description. We start by transferring these from the argument list:

```
static void * Class_ctor (void * _self, va_list * app)
{   struct Class * self = _self;

    self -> name = va_arg(* app, char *);
    self -> super = va_arg(* app, struct Class *);
    self -> size = va_arg(* app, size_t);

    assert(self -> super);
```

self cannot be a null pointer because we would not have otherwise found this method. **super**, however, could be zero and that would be a very bad idea.

The next step is inheritance. We must copy the constructor and all other methods from the superclass description at **super** to our new class description at **self**:

```
const size_t offset = offsetof(struct Class, ctor);
...
memcpy((char *) self + offset, (char *) self -> super
                + offset, sizeOf(self -> super) - offset);
```

Assuming that the constructor is the first method in **struct Class**, we use the ANSI-C macro **offsetof()** to determine where our copy is to start. Fortunately, the class description at **super** is subclassed from **Object** and has inherited **sizeOf()** so we can compute how many bytes to copy.

While this solution is not entirely foolproof, it seems to be the best compromise. Of course, we could copy the entire area at **super** and store the new name etc. afterwards; however, we would still have to rescue the **struct Object** at the beginning of the new class description, because **new()** has already stored the class description's class description pointer there.

The last part of the **Class** constructor is responsible for overwriting whatever methods have been specified in the argument list to **new()**. ANSI-C does not let us assign function pointers to and from **void ***, so a certain amount of casting is necessary:

```
{
    typedef void (* voidf) ();  /* generic function pointer */
    voidf selector;
    va_list ap = * app;

    while ((selector = va_arg(ap, voidf)))
    {   voidf method = va_arg(ap, voidf);

        if (selector == (voidf) ctor)
            * (voidf *) & self -> ctor = method;
        else if (selector == (voidf) dtor)
            * (voidf *) & self -> dtor = method;
        else if (selector == (voidf) differ)
            * (voidf *) & self -> differ = method;
        else if (selector == (voidf) puto)
            * (voidf *) & self -> puto = method;
    }

    return self;
}}
```

As we shall see in section 6.10, this part of the argument list is best shared among all class constructors so that the selector/method pairs may be specified in any order. We accomplish this by no longer incrementing *** app**; instead we pass a copy **ap** of this value to **va_arg()**.

Storing the methods in this fashion has a few consequences: If no class constructor is interested in a selector, a selector/method pair is silently ignored, but at least it is not added to a class description where it does not belong. If a method does not have the proper type, the ANSI-C compiler will not detect the error because the variable argument list and our casting prevent type checks. Here we rely on the programmer to match the selector to the method supplied with it, but they must be specified as a pair and that should result in a certain amount of plausibility.

6.7 Initialization

Normally we obtain a class description by sending **new()** to a metaclass description. In the case of **Class** and **Object** we would issue the following calls:

```
const void * Object = new(Class,
        "Object", Object, sizeof(struct Object),
        ctor, Object_ctor,
        dtor, Object_dtor,
        differ, Object_differ,
        puto, Object_puto,
        0);

const void * Class = new(Class,
        "Class", Object, sizeof(struct Class),
        ctor, Class_ctor,
        dtor, Class_dtor,
        0);
```

Unfortunately, either call relies on the other already having been completed. Therefore, the implementation of **Class** and **Object** in *Object.c* requires static initialization of the class descriptions. This is the only point where we explicitly initialize a **struct Class**:

```
static const struct Class object [] = {
    { { object + 1 },
      "Object", object, sizeof(struct Object),
      Object_ctor, Object_dtor, Object_differ, Object_puto
    },
    { { object + 1 },
      "Class", object, sizeof(struct Class),
      Class_ctor, Class_dtor, Object_differ, Object_puto
    }
};

const void * Object = object;
const void * Class = object + 1;
```

An array name is the address of the first array element and can already be used to initialize components of the elements. We fully parenthesize this initialization in case **struct Object** is changed later on.

6.8 Selectors

The job of a selector function is unchanged from chapter 2: One argument **_self** is the object for dynamic linkage. We verify that it exists and that the required method exists for the object. Then we call the method and pass all arguments to it; therefore, the method can assume that **_self** is a proper object for it. Finally, we return the result value of the method, if any, as the result of the selector.

Every dynamically linked method must have a selector. So far, we have hidden calls to the constructor and the destructor behind **new()** and **delete()**, but we still need the function names **ctor** and **dtor** for the selector/method pairs passed to the **Class** constructor. We may later decide to bind **new()** and **delete()** dynamically; therefore, it would not be a good idea to use their names in place of **ctor** and **dtor**.

We have introduced a common superclass **Object** for all our classes and we have given it some functionality that simplifies implementing selector functions. **classOf()** inspects an object and returns a non-zero pointer to its class description. This permits the following implementation for **delete()**:

```
void delete (void * _self)
{
    if (_self)
        free(dtor(_self));
}

void * dtor (void * _self)
{   const struct Class * class = classOf(_self);

    assert(class -> dtor);
    return class -> dtor(_self);
}
```

new() must be implemented very carefully but it works similarly:

```
void * new (const void * _class, ...)
{   const struct Class * class = _class;
    struct Object * object;
    va_list ap;

    assert(class && class -> size);
    object = calloc(1, class -> size);
    assert(object);
    object -> class = class;
    va_start(ap, _class);
    object = ctor(object, & ap);
    va_end(ap);
    return object;
}
```

We verify the class description and we make sure that we can create a zero-filled object. Then we initialize the class description of the object and we are ready to let the normal selector **ctor()** find and execute the constructor:

```
void * ctor (void * _self, va_list * app)
{   const struct Class * class = classOf(_self);

    assert(class -> ctor);
    return class -> ctor(_self, app);
}
```

There is perhaps a bit too much checking going on, but we have a uniform and robust interface.

6.9 Superclass Selectors

Before a subclass constructor performs its own initialization, it is required to call the superclass constructor. Similarly, a subclass destructor must call its superclass destructor after it has completed its own resource reclamation. When we are implementing selector functions, we should also supply selectors for the superclass calls:

```
void * super_ctor (const void * _class,
                  void * _self, va_list * app)
{   const struct Class * superclass = super(_class);

    assert(_self && superclass -> ctor);
    return superclass -> ctor(_self, app);
}

void * super_dtor (const void * _class, void * _self)
{   const struct Class * superclass = super(_class);

    assert(_self && superclass -> dtor);
    return superclass -> dtor(_self);
}
```

These selectors should only be called by a subclass implementation; therefore, we include their declarations into the representation file and not into the interface file. To be on the safe side, we supply superclass selectors for all dynamically linked methods, i.e., every selector has a corresponding superclass selector. This way, every dynamically linked method has a simple way to call its superclass method.

Actually, there is a subtle trap luring. Consider how a method of an arbitrary class **X** would call its superclass method. This is the correct way:

```
static void * X_method (void * _self, va_list * app)
{   void * p = super_method(X, _self, app);
    ...
```

Looking at the superclass selectors shown above we see that **super_method()** in this case calls

```
super(X) -> method(_self, app);
```

i.e., the method in the superclass of the class **X** for which we just defined **X_method()**. The same method is still reached even if some subclass **Y** inherited **X_method()** because the implementation is independent of any future inheritance.

The following code for **X_method()** may look more plausible, but it will break once the method is inherited:

```
static void * X_method (void * _self, va_list * app)
{   void * p =  /* WRONG */
        super_method(classOf(_self), _self, app);
    ...
```

The superclass selector definition now produces

```
super(classOf(_self)) -> method(_self, app);
```

If **_self** is in class **X**, we reach the same method as before. However, if **_self** is in a subclass **Y** of **X** we get

```
super(Y) -> method(_self, app);
```

and that is still **X_method()**, i.e., instead of calling a superclass method, we get stuck in a sequence of recursive calls!

6.10 A New Metaclass — *PointClass*

Object and **Class** are the root of our class hierarchy. Every class is a subclass of **Object** and inherits its methods, every metaclass is a subclass of **Class** and cooperates with its constructor. **Any** in section 6.4 has shown how a simple subclass can be made by replacing dynamically linked methods of its superclass and, possibly, defining new statically linked methods.

We now turn to building classes with more functionality. As an example we connect **Point** and **Circle** to our class hierarchy. These classes have a new dynamically linked method **draw()**; therefore, we need a new metaclass to accommodate the link. Here is the interface file *Point.h*:

```
#include "Object.h"

extern const void * Point;          /* new(Point, x, y); */

void draw (const void * self);
void move (void * point, int dx, int dy);

extern const void * PointClass;     /* adds draw */
```

The subclass always includes the superclass and defines a pointer to the class description and to the metaclass description if there is a new one. Once we introduce metaclasses, we can finally declare the selector for a dynamically linked method where it belongs: in the same interface file as the metaclass pointer.

The representation file *Point.r* contains the object structure **struct Point** with its access macros as before, and it contains the superclass selectors together with the structure for the metaclass:

```
#include "Object.r"

struct Point { const struct Object _;   /* Point : Object */
    int x, y;               /* coordinates */
};

#define x(p)    (((const struct Point *)(p)) -> x)
#define y(p)    (((const struct Point *)(p)) -> y)

void super_draw (const void * class, const void * self);

struct PointClass {
    const struct Class _;           /* PointClass : Class */
    void (* draw) (const void * self);
};
```

The implementation file *Point.c* contains **move()**, **Point_draw()**, **draw()**, and **super_draw()**. These methods are written as before; we saw the technique for the superclass selector in the previous section. The constructor must call the superclass constructor:

```
static void * Point_ctor (void * _self, va_list * app)
{   struct Point * self = super_ctor(Point, _self, app);

    self -> x = va_arg(* app, int);
    self -> y = va_arg(* app, int);
    return self;
}
```

One new idea in this file is the constructor for the metaclass. It calls the superclass constructor to perform inheritance and then uses the same loop as **Class_ctor()** to overwrite the new dynamically linked method **draw()**:

```
static void * PointClass_ctor (void * _self, va_list * app)
{   struct PointClass * self
                    = super_ctor(PointClass, _self, app);
    typedef void (* voidf) ();
    voidf selector;
    va_list ap = * app;

    while ((selector = va_arg(ap, voidf)))
    {   voidf method = va_arg(ap, voidf);

        if (selector == (voidf) draw)
            * (voidf *) & self -> draw = method;
    }

    return self;
}
```

Note that we share the selector/method pairs in the argument list with the superclass constructor: **ap** takes whatever **Class_ctor()** returns in *** app** and starts the loop from there.

With this constructor in place we can dynamically initialize the new class descriptions: **PointClass** is made by **Class** and then **Point** is made with the class description **PointClass**:

```
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);
}
```

Writing the initialization is straightforward: we specify the class names, inheritance relationships, and the size of the object structures, and then we add selector/method pairs for all dynamically linked methods defined in the file. A zero completes each argument list.

In chapter 9 we will perform this initialization automatically. For now, **initPoint()** is added to the interface in *Point.h* and the function must definitely be called before we can make points or subclasses. The function is interlocked so that it can be called more than once — it will produce exactly one class description **PointClass** and **Point**.

As long as we call **initPoint()** from **main()** we can reuse the test program *points* from section 4.1 and we get the same output:

```
$ points p
"." at 1,2
"." at 11,22
```

Circle is a subclass of **Point** introduced in chapter 4. In adding it to the class hierarchy, we can remove the ugly code in the constructor shown in section 4.7:

```
static void * Circle_ctor (void * _self, va_list * app)
{   struct Circle * self = super_ctor(Circle, _self, app);

    self -> rad = va_arg(* app, int);
    return self;
}
```

Of course, we need to add an initialization function **initCircle()** to be called from **main()** before circles can be made:

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

Because **Circle** depends on **Point**, we call on **initPoint()** before we initialize **Circle**. All of these functions do their real work only once, and we can call them in any order as long as we take care of the interdependence inside the function itself.

6.11 Summary

Objects point to their class descriptions which, for the most part, contain pointers to dynamically linked methods. Class descriptions with the same set of method pointers constitute a metaclass — class descriptions are objects, too. A metaclass, again, has a class description.

Things remain finite because we start with a trivial class **Object** and with a first metaclass **Class** which has **Object** as a superclass. If the same set of methods — constructor, destructor, comparison, and display — can be applied to objects and class descriptions, then the metaclass description **Class** which describes the class description **Object** also describes itself.

A metaclass constructor fills a class description and thus implements binary inheritance, the destructor returns zero to protect the class description from being destroyed, the display function could show method pointers, etc. Two class descriptions are the same if and only if their addresses are equal.

If we add dynamically linked methods such as **draw()**, we need to start a new metaclass, because its constructor is the one to insert the method address into a class description. The metaclass description always uses **struct Class** and is, therefore, created by a call

```
PointClass = new(Class, ...
            ctor, PointClass_ctor,
            0);
```

Once the metaclass description exists, we can create class descriptions in this metaclass and insert the new method:

```
Point = new(PointClass, ...
            draw, Point_draw,
            ...
            0);
```

These two calls must be executed exactly once, before any objects in the new class can be created. There is a standard way to write all metaclass constructors so that the selector/method pairs can be specified in any order. More classes in the same metaclass can be created just by sending **new()** to the metaclass description.

Selectors are also written in a standard fashion. It is a good idea to decide on a discipline for constructors and destructors to always place calls along the superclass chain. To simplify coding, we provide superclass selectors with the same arguments as selectors; an additional first parameter must be specified as the class for which the method calling the superclass selector is defined. Superclass selectors, too, are written according to a standard pattern.

A coherent style of verifications makes the implementations smaller and more robust: selectors verify the object, its class, and the existence of a method; superclass selectors should additionally verify the new class argument; a dynamically linked method is only called through a selector, i.e., it need not verify its object. A statically linked method is no different from a selector: it must verify its argument object.

Let us review the meaning of two basic components of objects and class descriptions and our naming conventions. Every class eventually has **Object** as a superclass. Given a pointer *p* to an object in an arbitrary subclass of **Object**, the component *p*–**>class** points to the class description for the object. Assume that the pointer *C* points to the same class description and that *C* is the class name published in the interface file *C.h*. Then the object at *p* will be represented with a **struct** *C*. This explains why in section 6.3 **Class–>class** has to point to **Class** itself: the object to which **Class** points is represented by a **struct Class**.

Every class description must begin with a **struct Class** to store things like the class name or a pointer to the superclass description. Now let *C* point to a class description and let *C*–**>super** point to the same class description as the pointer *S* published in the interface file *S.h*, i.e., *S* is the superclass of *C*. In this case, **struct** *C* must start with **struct** *S*. This explains why in section 6.3 **Class–>super** has to point to **Object**: we decided that a **struct Class** starts with a **struct Object**.

The only exception to this rule is the fact that **Object–>super** has the value **Object** although it was pointed out in section 6.3 that this was a rather arbitrary decision.

7
The *ooc* Preprocessor Enforcing a Coding Standard

Looking over the last chapter, it seems that we have solved the big problem of cleaning up class maintenance by introducing another big problem: we now have an awful number of conventions about how certain functions have to be written (most notably a metaclass constructor) and which additional functions must be provided (selectors, superclass selectors, and initializations). We also have rules for defensive coding, i.e., argument checking, but the rules are not uniform: we should be paranoid in selectors and statically linked methods but we can be more trusting in dynamically linked methods. If we should decide to change our rules at a later date, we will likely have to revise a significant amount of rather standard code — a repetitive and error-prone process.

In this chapter we will look at the design of a preprocessor *ooc* which helps us to stick to the conventions developed in the last chapter. The preprocessor is simple enough to be implemented in a few days using *awk* [AWK88], and it enables us to follow (and later redesign) our coding conventions. *ooc* is documented with a manual page in appendix C, the implementation is detailed in appendix B, and the complete source code is available as part of the sources to this book.

ooc is certainly not intended to introduce a new programming language — we are still working with ANSI-C and the output from *ooc* is exactly what we would write by hand as well.

7.1 *Point* Revisited

We want to engineer a preprocessor *ooc* which helps us maintain our classes and coding standards. The best way to design such a preprocessor is to take a typical, existing example class, and see how we can simplify our implementation effort using reasonable assumptions about what a preprocessor can do. In short, let us ''play'' preprocessor for a while.

Point from chapter 4 and section 6.10 is a good example: it is not the root class of our system, it requires a new metaclass, and it has a few, typical methods. From now on we will use italics and refer to it as *Point* to emphasize that it only serves as a model for what our preprocessor has to handle.

We start with a more or less self-explanatory class description that we can easily understand and that is not too hard for an *awk* based preprocessor to read:

```
% PointClass: Class  Point: Object {  // header
    int x;                           // object components
    int y;
%                                    // statically linked
    void move (_self, int dx, int dy);
%-                                   // dynamically linked
    void draw (const _self);
%}
```

Boldface in this class description indicates items that *ooc* recognizes; the regular line printer font is used for items which the preprocessor reads here and reproduces elsewhere. Comments start with **//** and extend to the end of a line; lines can be continued with a backslash.

Here we describe a new class *Point* as a subclass of *Object*. The objects have new components *x* and *y*, both of type *int*. There is a statically linked method *move()* that can change its object using the other parameters. We also introduce a new dynamically linked method *draw()*; therefore, we must start a new metaclass *PointClass* by extending the meta superclass *Class*. The object argument of *draw()* is **const**, i.e., it cannot be changed.

If we do not have new dynamically linked methods, the description is even simpler. Consider **Circle** as a typical example:

```
% PointClass  Circle: Point {          // header
    int rad;                           // object component
%}                                     // no static methods
```

These simple, line-oriented descriptions contain enough information so that we can completely generate interface files. Here is a pattern to suggest how *ooc* would create *Point.h*:

```
#ifndef Point_h
#define Point_h

#include "Object.h"

extern const void * Point;

for all methods in %
  void move (void * self, int dx, int dy);

if there is a new metaclass
  extern const void * PointClass;

  for all methods in %-
    void draw (const void * self);

void initPoint (void);

#endif
```

Boldface marks parts of the pattern common to all interface files. The regular typeface marks information which *ooc* must read in the class description and insert into the interface file. Parameter lists are manipulated a bit: **_self** or **const _self** are converted to suitable pointers; other parameters can be copied directly.

Parts of the pattern are used repeatedly, e.g., for all methods with a certain linkage or for all parameters of a method. Other parts of the pattern depend on conditions such as a new metaclass being defined. This is indicated by italics and indentation.

The class description also contains enough information to produce the representation file. Here is a pattern to generate *Point.r*:

```
#ifndef Point_r
#define Point_r

#include "Object.r"

struct Point { const struct Object _;
    for all components
      int x;
      int y;
};

if there is a new metaclass
  struct PointClass { const struct Class _;
      for all methods in %-
        void (* draw) (const void * self);
  };

  for all methods in %-
    void super_draw (const void * class, const void * self);

#endif
```

The original file can be found in section 6.10. It contains definitions for two access macros **x()** and **y()**. So that *ooc* can insert them into the representation file, we adopt the convention that a class description file may contain extra lines in addition to the class description itself. These lines are copied to an interface file or, if they are preceded by a line with **%prot**, to a representation file. **prot** refers to protected information — such lines are available to the implementations of a class and its subclasses but not to an application using the class.

The class description contains enough information so that *ooc* can generate a significant amount of the implementation file as well. Let us look at the various parts of *Point.c* as an example:

```
#include "Point.h"                    // include
#include "Point.r"
```

First, the implementation file includes the interface and representation files.

```
                                      // method header
void move (void * _self, int dx, int dy) {
    for all parameters                // importing objects
      if parameter is a Point
        struct Point * self = _self;

    for all parameters                // checking objects
      if parameter is an object
        assert(_self);

  ...                                 // method body
```

For statically linked methods we can check that they are permitted for the class before we generate the method header. With a loop over the parameters we can initialize local variables from all those parameters which refer to objects in the class to which the method belongs, and we can protect the method against null pointers.

```
                                        // method header
static void Point_draw (const void * _self) {
    for all parameters                  // importing objects
      if parameter is a Point
        const struct Point * self = _self;

  ...                                   // method body
```

For dynamically linked methods we also check, generate headers, and import objects. The pattern can be a bit different to account for the fact that the selector should have already checked that the objects are what they pretend to be.

There are a few problems, however. As a subclass of *Object*, our class *Point* may overwrite a dynamically linked method such as *ctor()* that first appeared in *Object*. If *ooc* is to generate all method headers, we have to read all superclass descriptions back to the root of the class tree. From the superclass name *Object* in the class description for *Point* we have to be able to find the class description file for the superclass. The obvious solution is to store a description for *Object* in a file with a related name such as *Object***.d**.

```
static void * Point_ctor (void * _self, va_list * app) {
  ...
```

Another problem concerns the fact that *Point_ctor()* calls the superclass selector, and, therefore, does not need to import the parameter objects like *Point_draw()* did. It is probably a good idea if we have a way to tell *ooc* each time whether or not we want to import and check objects.

```
if there is a new metaclass
  for all methods in %-
    void draw (const void * _self) {  // selector
        const struct PointClass * class = classOf(_self);

        assert(class -> draw);
        class -> draw(self);
    }
                                      // superclass selector
    void super_draw (const void * class, const void * _self) {
        const struct PointClass * superclass = super(class);

        assert(_self && superclass -> draw);
        superclass -> draw(self);
    }
```

If the class description defines a new metaclass, we can completely generate the selectors and superclass selectors for all new dynamically linked methods. If we want to, in each selector we can use loops over the parameters and check that there are no null pointers posing as objects.

```
if there is a new metaclass
                                        // metaclass constructor
  static void * PointClass_ctor (void * _self, va_list * app) {
  {   struct PointClass * self =
                              super_ctor(PointClass, _self, app);
      typedef void (* voidf) ();
      voidf selector;
      va_list ap = * app;

      while ((selector = va_arg(ap, voidf)))
      {   voidf method = va_arg(ap, voidf);

          for all methods in %-
            if (selector == (voidf) draw)
            {   * (voidf *) & self -> draw = method;
                continue;
            }
      }
      return self;
  }
```

With a loop over the method definitions in the class description we can completely generate the metaclass constructor. Somehow, however, we will have to tell *ooc* the appropriate method header to use for *ctor()* — a different project might decide on different conventions for constructors.

```
const void * Point;                     // class descriptions
if there is a new metaclass
  const void * PointClass;

void initPoint (void)                   // initialization
{
    if there is a new metaclass
      if (! PointClass)
          PointClass = new(Class, "PointClass",
                           Class, sizeof(struct PointClass),
                           ctor, PointClass_ctor,
                           (void *) 0);

    if (! Point)
        Point = new(PointClass, "Point",
                    Object, sizeof(struct Point),
                    for all overwritten methods
                      ctor, Point_ctor,
                      draw, Point_draw,
                    (void *) 0);
}
```

The initialization function depends on a loop over all dynamically linked methods overwritten in the implementation.

7.2 Design

Let us draw a few conclusions from the experiment of preprocessing *Point*. We started with a fairly simple, line-oriented class description:

```
interface lines                                  // arbitrary
%prot
representation lines

% metaclass[: metasuper]  class: super {         // header
    ...                                          // object components
%                                                // statically linked
    type name ([const] _self, ...);
    ...
%-                                               // dynamically linked
    type name ([const] _self, ...);
    ...
%}
```

The only difficulty is that we need to untangle the parameter lists and split type and name information in each declarator. A cheap solution is to demand that **const**, if any, precede the type and that the type completely precede the name.* We also recognize the following special cases:

`_self`	message target in the current class
_name	other object in the current class
class `@` *name*	object in other class

All of these can be preceded by **const**. Objects in the current class are dereferenced when they are imported.

The file name for a class description is the class name followed by **.d** so that *ooc* can locate superclass descriptions. We do not have to worry much about the metaclass: either a class has the same metaclass as its superclass, or a class has a new metaclass and the superclass of this metaclass is the metaclass of the class' superclass. Either way, if we read the superclass description, we have sufficient information to cope with the metaclass.

Once *ooc* has digested a class description, it has enough information to generate the interface and the representation file. It is wise to design a command as a filter, i.e., to require explicit i/o redirection to create a file, so we arrive at the following typical invocations for our preprocessor:

```
$ ooc Point –h > Point.h     # interface file
$ ooc Point –r > Point.r     # representation file
```

The implementation file is more difficult. Somehow *ooc* must find out about method bodies and whether or not parameters should be checked and imported. If we add the method bodies to the class description file, we keep things together, but we cause a lot more processing: during each run *ooc* has to load class description files back to the root class, however, method bodies are only interesting in the outermost description file. Additionally, implementations are more likely to change

* If necessary, this can always be arranged with a **typedef**.

than interfaces; if method bodies are kept with the class description, *make* will recreate the interface files every time we change a method body, and this probably leads to lots of unnecessary recompilations.*

A cheap solution would be to let *ooc* read a class description and produce a skeleton implementation file containing all *possible* method headers for a class together with the new selectors, if any, and the initialization code. This requires a loop from the class back through its superclasses to generate headers and fake bodies for all the dynamically linked methods which we encounter along the way. *ooc* would be invoked like this:

```
$ ooc Point -dc > skeleton  # starting an implementation
```

This provides a useful starting point for a new implementation but it is difficult to maintain: The *skeleton* will contain all possible methods. The basic idea is to erase those which we do not need. However, every method will appear twice: once with a header and a fake body and once in the argument list to create the class description. It is hard, but absolutely essential, to keep this synchronized.

ooc is supposed to be a development and maintenance tool. If we change a method header in a class description it is only reasonable to expect that *ooc* propagates the change to all implementation files. With the skeleton approach we can only produce a new skeleton and manually edit things back together — not entirely a pleasing prospect!

If we do not want to store method bodies within a class description and if we require that *ooc* change an existing implementation file, we are led to design *ooc* as a preprocessor. Here is a typical invocation:

```
$ ooc Point Point.dc > Point.c      # preprocessing
```

ooc loads the class description for *Point* and then reads the implementation file *Point.dc* and writes a preprocessed version of this file to standard output. We can even combine this with the skeleton approach described above, as long as we create the skeleton with preprocessing statements rather than as an immutable C file.

7.3 Preprocessing

What preprocessing statements should *ooc* provide? Looking over our experiment with *Point* in section 7.1 there are three areas where *ooc* can help: given a method name, it knows the method header; given a method, it can check and import the object parameters; somewhere it can produce selectors, the metaclass constructor, and the initialization function as required. Experimenting once again with *Point*, the following implementation file *Point.dc* seems reasonable:

```
% move {
%casts
    self -> x += dx, self -> y += dy;
}
```

* *yacc* has a similar problem with the header file *y.tab.h*. The standard approach is to duplicate this file, rewrite the copy only if it is new, and use the copy in *makefile* rules. See [K&P84].

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

    self -> x = va_arg(* app, int);
    self -> y = va_arg(* app, int);
    return self;
}

% Point draw {
%casts
    printf("\".\" at %d,%d\n", x(self), y(self));
}

%init
```

Boldface indicates what *ooc* finds interesting:

`%` *method* `{`	header for statically linked method
`%` *class method* `{`	header to overwrite dynamically linked method
`%casts`	import object parameters
`%init`	create selectors and initialization code

For a method with static linkage, we already know the class for which it is declared. However, the class may be specified anyway, and if we should decide to change the method's linkage later, we do not need to edit the source again.

There is a question whether we should not require that the method header be spelled out by the programmer, parameter list and all. While this would make the implementation file easier to read, it is harder to maintain if a method definition is changed. It is also (marginally) harder to parse.

7.4 Implementation Strategy

We know what *ooc* has to do. How do we write the preprocessor? For reasons of efficiency, we may eventually have to resort to a platform like *lex* and *yacc*, but a first implementation is a lot cheaper if we use a string programming language like *awk* or *perl*. If this works out, we know the necessary algorithms and data structures, and it should be easy to transcribe them into a more efficient implementation; we might even use something like an *awk* to C translator to do the job. Above all, however, with a cheap first implementation we can verify that our idea is feasible and convenient in the first place.

ooc parses the class descriptions and builds a database. Given the database, command line options decide what will be generated. As we have seen, generation can be based on some sort of pattern with words to be printed directly and words to be replaced with information from the database. Our patterns contained loops, however, so it seems that a pattern really is an *awk* function with control structures and variables.

A first implementation of *ooc* actually worked this way, but it proved to be difficult to change. There is a much better way: a simple report language with text replacement, loops, and conditionals, which is used to express a pattern and which is interpreted by *ooc* to generate output.

The implementation is explained in more detail in appendix B. The report language is defined as part of the *ooc* manual in appendix C. The report language has about twenty five replacements, ten loops, two conditionals, and a way to call a report as part of another report. As an example, here is how selectors could be generated:

```
`{if `newmeta 1
  `{%–
    `result  `method  ( `{() `const `type  `_ `name `}, )  { `n
`t      const  struct  `meta  *  class  =  classOf(self); `n
        `%casts
`t      assert(class  –>  `method ); `n
`t      `{ifnot `result void return `} \
            class  –>  `method ( `{() `name `}, ); `n
    } `n
  `}
`}
```

Within reports, *ooc* finds all those words interesting that start with a back quote; groups start with **`{** and end with **`}** and are either loops or conditionals; a report call starts with **`%**; all other words starting with a back quote are replaced with information from the database.

`{if** takes the next two words and executes the rest of the group if the words are equal. **`newmeta will be replaced by **1** if *ooc* works with a class description that defines a new metaclass. Therefore, the selectors are only generated for a new metaclass.

`{%–** is a loop over the dynamically linked methods in the class description. **`method is replaced by the current method name; **`result** is the current result type.

`{()** is a loop over the parameters of the current method. The meaning of **`const, **`type**, and **`name** should be fairly obvious: they are the pieces of the current parameter declarator. `_ is an underscore if the current parameter is an object of the current class. **`},** is a little trick: it emits a comma, if there is another parameter, and it terminates a loop like any other token starting with **`}**.

`%casts calls another report, **casts**, which is responsible for importing object parameters. For now, this report looks about as follows:

```
% casts              // the %casts statement
`{()                 // import
  `{if `_ _
`t  `const struct  `cast  *  `name  =  _ `name ; `n
  `}
`}n
`{if `linkage %      // for static linkage, need to check
  `%checks
`}
```

A line starting with **%** precedes reports in a report file and introduces the report name. The rest of the **casts** report should be clear: **`cast** refers to the name of the class of a parameter object and **`linkage** is the linkage of the current method, i.e.,

one of the section symbols in the class description. We construct a local variable to dereference a parameter if it is an object in the current class. **`}n** is another trick: it emits a newline if anything was generated for a group.

%casts is also responsible for checking any objects supplied as parameters to a method with static linkage. Since selectors have a similar problem, we use a separate report **checks** that can be called from a report to generate selectors:

```
% checks        // check all object parameters
`{()
  `{ifnot `cast `
`t  assert( `name ); `n
  `}fi
`}n
```

Until the next chapter, we can at least guard against null pointers by calling **assert()**. This test is required for objects: **`** is replaced by nothing, i.e., we generate **assert()** if we are looking at a parameter object from an arbitrary class.

Two words have not been explained yet: **`t** generates a tab and **`n** generates a newline character. We want to generate legible C programs; therefore, we have to closely monitor how much white space is generated. So that we may indent reports based on their own control structures in the groups, *ooc* will not generate leading white space and it will only generate a single empty line in a row. **`t** must be used to achieve indentation and **`n** must be specified to break the output into lines.*

7.5 *Object* Revisited

In section 7.1 we saw that in order to work on *Point* we need to collect information for its superclasses all the way back to the root class. So how do we specify *Object*? It would certainly not make sense to define *Object* as part of the *awk* program. The obvious approach is to write a class description file:

```
#include <stdarg.h>
#include <stddef.h>
#include <stdio.h>
%prot
#include <assert.h>

% Class  Object {
    const Class @ class;            // object's description
%
    void delete (_self);                // reclaim instance
    const void * classOf (const _self); // object's class
    size_t sizeOf (const _self);        // object's size
```

* Experiments with the beautifiers *cb* and *indent* have not produced acceptable results. The words **`t** and **`n** are only a minor nuisance and monitoring the generation of leading white space and successive newlines does not overly complicate the report generator.

```
%–
    void * ctor (_self, va_list * app);     // constructor
    void * dtor (_self);                    // destructor
    int differ (const _self, const Object @ b); // true if !=
    int puto (const _self, FILE * fp);      // display
%}
```

Unfortunately, this is a special case: as the root class, *Object* has no superclass, and as the first metaclass, *Class* has no meta superclass. Only one class description has this property; therefore, we let *ooc* recognize this special syntax for the class header as the description of the **`root** and **`metaroot** classes.

Class poses another problem: we saw in section 7.1 that new metaclasses can be declared right with a new class because they can only have method links as new components. *Class* is the first metaclass and does have some extra components:

```
% Class  Class: Object {
    const char * name;              // class' name
    const Class @ super;            // class' superclass
    size_t size;                    // object's memory size
%
    Object @ new (const _self, ...);    // create instance
    const void * super (const _self);   // class' superclass
%}
```

It turns out that our syntax for class descriptions is quite sufficient to describe *Class*. It is another special case for *ooc*: it is the only class that is allowed to have itself as a metaclass.

If we put both descriptions in the same class description file *Object***.d**, and if we let *Object* precede *Class*, the search for class descriptions in *ooc* will terminate all by itself. Our database is complete.

We could write the implementation of *Object* and *Class* by hand — there is little point in adding special code to *ooc* if it is only used to generate a single implementation. However, our report generation mechanism is good enough to be adapted for *Object* and we can get a substantial amount of assistance.

The interface files for *Point* and *Object* are quite analogous, with the exception that *Object.h* has no superclass interface to include and does not declare an initialization function. The corresponding report file *h.rep* is used many times, however, so we should avoid cluttering it up with conditionals that are not usually needed. Instead, we add a parameter to the command line of *ooc*:

```
$ ooc –R Object –h > Object.h
```

This parameter causes a special report file *h–R.rep* to be loaded which is tailored to the root class. Both report files mostly generate method headers and they can share yet another report file *header.rep* which contains the **header** report used in both cases.

Similarly, the representation files of *Point* and *Object* have much in common, and we use **–R** to load a report file *r–R.rep* in place of *r.rep* to account for the differences:

```
$ ooc –R Object –r > Object.r
```

Object.r has no superclass representation to include, and the metaclass structure for *Class* starts out with the extra components. The common code to declare superclass selectors and methods as metaclass components is located in another report file *va.rep*.

Finally, we can use –**R** and one more report file *c–R.rep* in place of *c.rep* to help in generating the implementation:

```
$ ooc –R Object Object.dc > Object.c
```

ooc will add **include** statements and preprocess method headers in *Object.dc* as in any other implementation file. The only difference lies in the implementation of **%init**: We can still let *ooc* generate the selectors and superclass selectors, but we have to code the static initialization of the class descriptions shown in section 6.7 by hand.

There is a question how the metaclass constructor *Class_ctor***()** should be written. If we do it by hand in *Object.dc* we essentially code the loop to process the selector/method pairs twice: once in *Object.dc* for *Class_ctor***()** and once in the report file *c.rep* for all other classes. It turns out that we have enough information to do it in *c–R.rep*. If we assume that the first few arguments to the constructor appear in the order of the components specified for *Class* we can generate the entire constructor and thus share the loop code as a report **meta-ctor-loop** in a common report file *etc.rep*.

7.6 Discussion

Object demonstrates something that is both a strength and a weakness of our technique: we have a choice where to implement our decisions. Code can be placed in a class description or implementation, it can be turned into a report, or it can be buried somewhere in the *awk* program.

Obviously, we should be very careful about the last option: *ooc* is intended to be used for more than one project; therefore, the *awk* program should be kept free of any inherent knowledge about a project. It is allowed to collect information and offer it for replacement, and it is required to connect preprocessor statements to reports, but it should assume nothing about report contents or ordering.

The reports can be changed between projects and they are the place to enforce coding standards. Reports and all other files like class descriptions and implementations are searched in directories specified as an environment variable **OOCPATH**. This can be used to load different versions of the reports for different projects.

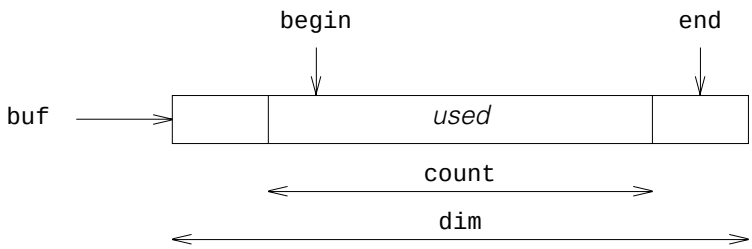
Our approach for *Object* demonstrates the flexibility of replacing reports: we can share common report code by calling reports in common files and we can avoid the overhead of checking for a special case in most processing. While one-of-a-kind code can be written into an implementation file, it is almost as easy to write it as a report so that it can benefit from report generation. There is hardly an excuse for duplicating code.

In general, report generation has advantages and drawbacks. On the positive side, it simplifies the development of a class hierarchy and changes during maintenance because the reports are a single, central place to enforce coding standards. If we want to trace selector calls, for example, we simply insert a trace line into the selector body in the reports file, and the trace will be generated all over.

Report generation takes more execution time than plain function calls, however. A preprocessor should generate **#line** stamps for the C compiler so that the error messages refer to the original source lines. There is a provision for generating **#line** stamps in *ooc*, but with report generation the **#line** stamps are not as good as they perhaps could be. Maybe once our reports are stable, we could write another preprocessor to generate an *awk* program from the reports?

7.7 An Example — *List, Queue,* and *Stack*

Let us look at a few new classes implemented from scratch using *ooc* so that we can appreciate the effort we are now spared. We start with a **List** implemented as a double-ended ring buffer that will dynamically expand as needed.



begin and **end** limit the used part of the list, **dim** is the maximal buffer size, and **count** is the number of elements currently stored in the buffer. **count** makes it easy to distinguish between a full and an empty buffer. Here is the class description *List.d*:

```
% ListClass: Class  List: Object {
    const void ** buf;  // const void * buf [dim]
    unsigned dim;       // current buffer dimension
    unsigned count;     // # elements in buffer
    unsigned begin;     // index of takeFirst slot, 0..dim
    unsigned end;       // index of addLast slot, 0..dim
%
    Object @ addFirst (_self, const Object @ element);
    Object @ addLast (_self, const Object @ element);
    unsigned count (const _self);
    Object @ lookAt (const _self, unsigned n);
    Object @ takeFirst (_self);
    Object @ takeLast (_self);
%–                      // abstract, for Queue/Stack
    Object @ add (_self, const Object @ element);
    Object @ take (_self);
%}
```

The implementation in *List.dc* is not very difficult. The constructor provides an initial buffer:

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

    if (! (self -> dim = va_arg(* app, unsigned)))
        self -> dim = MIN;
    self -> buf = malloc(self -> dim * sizeof * self -> buf);
    assert(self -> buf);
    return self;
}
```

Normally, the user will provide the minimal buffer size. As a default we define **MIN** with a suitable value. The destructor eliminates the buffer but not the elements still in it:

```
% List dtor {
%casts
    free(self -> buf), self -> buf = 0;
    return super_dtor(List, self);
}
```

addFirst() and **addLast()** add one element at **begin** or **end**:

```
% List addFirst {
%casts
    if (! self -> count)
        return add1(self, element);
    extend(self);
    if (self -> begin == 0)
        self -> begin = self -> dim;
    self -> buf[-- self -> begin] = element;
    return (void *) element;
}

% List addLast {
%casts
    if (! self -> count)
        return add1(self, element);
    extend(self);
    if (self -> end >= self -> dim)
        self -> end = 0;
    self -> buf[self -> end ++] = element;
    return (void *) element;
}
```

Both functions share the code to add a single element:

```
static void * add1 (struct List * self, const void * element)
{
    self -> end = self -> count = 1;
    return (void *) (self -> buf[self -> begin = 0] = element);
}
```

The invariants are different, however: if **count** is not zero, i.e., if there are any elements in the buffer, **begin** points to an element while **end** points to a free slot to

be filled. Either value may be just beyond the current range of the buffer. The buffer is used as a ring; therefore, we first map the variables around the ring before we access the buffer. **extend()** is the hard part: if there is no more room, we use **realloc()** to double the size of the buffer:

```
static void extend (struct List * self) // one more element
{
    if (self -> count >= self -> dim)
    {   self -> buf =
                realloc(self -> buf, 2 * self -> dim
                        * sizeof * self -> buf);
        assert(self -> buf);
        if (self -> begin && self -> begin != self -> dim)
        {   memcpy(self -> buf + self -> dim + self -> begin,
                self -> buf + self -> begin,
                (self -> dim - self -> begin)
                        * sizeof * self -> buf);
            self -> begin += self -> dim;
        }
        else
            self -> begin = 0;
    }
    ++ self -> count;
}
```

realloc() copies the pointers stored in **buf[]**, but if our ring does not start at the beginning of the buffer, we have to use **memcpy()** to shift the beginning of the ring to the end of the new buffer.

The remaining functions are much simpler. **count()** is simply an access function for the **count** component. **lookAt()** uses an arithmetic trick to return the proper element from the ring:

```
% List lookAt {
%casts
    return (void *) (n >= self -> count ? 0 :
        self -> buf[(self -> begin + n) % self -> dim]);
}
```

takeFirst() and **takeLast()** simply reverse the invariants of the corresponding **add** functions. Here is one example:

```
% List takeFirst {
%casts
    if (! self -> count)
        return 0;
    -- self -> count;
    if (self -> begin >= self -> dim)
        self -> begin = 0;
    return (void *) self -> buf[self -> begin ++];
}
```

takeLast() is left as an exercise — as are all selectors and initializations.

List demonstrates that *ooc* gets us back to dealing with the implementation issues of a class as a data type rather than with the idiosyncrasies of an object-oriented coding style. Given a reasonable base class, we can easily derive more problem-specific classes. **List** introduced dynamically linked methods **add()** and **take()** so that a subclass can impose an access discipline. **Stack** operates on one end:

Stack.d

```
% ListClass  Stack: List {
%}
```

Stack.dc

```
% Stack add {
    return addLast(_self, element);
}

% Stack take {
    return takeLast(_self);
}

%init
```

Queue can be derived from **Stack** and overwrite **take()** or it can be a subclass of **List** and define both methods. **List** itself does not define the dynamically linked methods and would; therefore, be called an *abstract base class*. Our selectors are robust enough so that we will certainly find out if somebody tries to use **add()** or **take()** for a **List** rather than a subclass. Here is a test program demonstrating that we can add plain C strings rather than objects to a **Stack** or a **Queue**:

```
#include "Queue.h"

int main (int argc, char ** argv)
{   void * q;
    unsigned n;

    initQueue();
    q = new(Queue, 1);

    while (* ++ argv)
        switch (** argv) {
        case '+':
            add(q, *argv + 1);
            break;
        case '–':
            puts((char *) take(q));
            break;
        default:
            n = count(q);
            while (n –– > 0)
            {   const void * p = take(q);

                puts(p), add(q, p);
            }
        }
    return 0;
}
```

If a command line argument starts with **+** it is added to the queue; for a – one element is removed. Any other argument displays the contents of the queue:

```
$ queue +axel – +is +here . – . – .
axel
is
here
is
here
here
```

Replacing the **Queue** by a **Stack** we can see the difference in the order of the removals:

```
$ stack +axel – +is +here . – . – .
axel
is
here
here
is
is
```

Because a **Stack** is a subclass of **List** there are various ways to nondestructively display the contents of the stack, for example:

```
n = count(q);
while (n —— > 0)
{   const void * p = takeFirst(q);
    puts(p), addLast(q, p);
}
```

7.8 Exercises

It is an interesting exercise to combine **Queue** with **Point** and **Circle** for a skeleton graphics program with redrawing capabilities.

The reports –**r** and **include** can be modified to implement the opaque structure definitions suggested in section 4.6.

The **init** reports can be modified to generate a method to display a **Class** structure.

Selectors and superclass selectors are generated by reports in *etc.rep*. They can be modified to provide an execution trace or to experiment with various levels of parameter checking.

The *ooc* command script and the modules *main.awk* and *report.awk* can be changed so that an argument –*x* results in a report *x.rep* to be loaded, interpreted, and removed. Given that change, a new report *flatten.rep* can show all methods available to a class.

8
Dynamic Type Checking Defensive Programming

8.1 Technique

Every object is accessed as **void ***. While this simplifies writing code, it does invite disaster: manipulating a non-object or the wrong object in a method, or worse, selecting a method from a class description that does not contain it, will cause significant amounts of grief. Let us trace a call to a dynamically linked method. **new()** produces a circle and the selector **draw()** is applied to it:

```
void * p = new(Circle, 1, 2, 3);

    draw(p);
```

The selector believes and dereferences the result of **classOf()**:

```
void draw (const void * _self) {
    const struct PointClass * class = classOf(_self);

    assert(class -> draw);
    class -> draw(_self);
}
```

The selected method believes and dereferences **_self**, which is the original pointer value produced by **new()**:

```
static void Circle_draw (const void * _self) {
    const struct Circle * self = _self;

    printf("circle at %d,%d rad %d\n",
        x(self), y(self), self -> rad);
}
```

classOf() also believes and dereferences a pointer. As a small consolation, it makes sure that its result is not a null pointer:

```
const void * classOf (const void * _self) {
    const struct Object * self = _self;

    assert(self);

    assert(self -> class);
    return self -> class;
}
```

In general, every assignment of a generic **void *** value to a pointer to some structure is suspect to begin with, and its legality ought to be verified. We have designed our methods to be polymorphic, i.e., the ANSI-C compiler cannot perform this check for us. We have to invent a dynamic type checking facility which closely limits how much damage a stray object or non-object can do.

Fortunately, our **void *** values know what they are pointing to: they point to objects, which all inherit from **Object** and, therefore, contain the component **.class**

pointing to their class description. Each class description is unique; therefore, the pointer value in **.class** can be used to determine if an object belongs to a particular class:

```
int isA (const _self, const Class @ class);
int isOf (const _self, const Class @ class);
```

These are two new, statically linked methods for **Object** and thus for any object: **isA()** is true if an object directly belongs to a specific class; **isOf()** is true if an object is derived from a specific class. The following axioms hold:

`isA(0, anyClass)`	always false
`isOf(0, anyClass)`	always false
`isOf(x, Object)`	true for all objects

It turns out that yet another static method for **Object** is even more useful:

```
void * cast (const Class @ class, const _self);
```

If **isOf(_self, class)** is true, **cast()** returns its argument **_self**, otherwise **cast()** terminates the calling process.

cast() is now going to replace **assert()** for most of our damage control. Wherever we are not so sure, we can wrap **cast()** around a dubious pointer to limit the damage which an unexpected value could do:

```
cast(someClass, someObject);
```

The function is also used for safely dereferencing pointers upon import to a method or selector:

```
struct Circle * self = cast(Circle, _self);
```

Notice that the parameters of **cast()** have the natural order for a casting operation: the class is written to the left of the object to be casted. **isOf()**, however, takes the same parameters in opposite order because in an **if** statement we would ask if an object ''is of'' a particular class.

Although **cast()** accepts **_self** with a **const** qualifier, it returns the value without **const** to avoid error messages on assignment. The same pun happens to be in the ANSI-C standard: **bsearch()** delivers a **void *** result for a table passed as **const void ***.

8.2 An Example — *list*

As an example of what we can do with **isOf()** and how safe **cast()** can be made, consider the following modification of the test program in section 7.7:

```
#include "Circle.h"
#include "List.h"

int main (int argc, char ** argv)
{   void * q;
    unsigned n;

    initList();
    initCircle();
```

```
    q = new(List, 1);

    while (* ++ argv)
        switch (** argv) {
        case '+':
            switch ((* argv)[1]) {
            case 'c':
                addFirst(q, new(Circle, 1, 2, 3));
                break;
            case 'p':
                addFirst(q, new(Point, 4, 5));
                break;
            default:
                addFirst(q, new(Object));
            }
            break;
        case '–':
            puto(takeLast(q), stdout);
            break;
        case '.':
            n = count(q);
            while (n –– > 0)
            {   const void * p = takeFirst(q);

                if (isOf(p, Point))
                    draw(p);
                else
                     puto(p, stdout);
                addLast(q, p);
            }
            break;
        default:
            if (isdigit(** argv))
                addFirst(q, (void *) atoi(* argv));
            else
                addFirst(q, * argv + 1);
        }
    return 0;
}
```

For arguments starting with **+** this program will add circles, points, or plain objects to a list. The argument – will remove the last object and display it with **puto()**. The argument **.** will display the current contents of the list; **draw()** is used if an entry is derived from **Point**. Finally, there is a deliberate attempt to place numbers or other strings as arguments into the list although they would cause problems once they were removed. Here is a sample output:

```
$ list +c +p + – . 1234
Circle at 0x122f4
Object at 0x12004
"." at 4,5
Object.c:66: failed assertion `sig == 0'
```

As we shall see in section 8.4, **addFirst()** uses **cast()** to make sure it only adds objects to the list. **cast()** can even be made robust enough to discover that a number or a string tries to pose as an object.

8.3 Implementation

With the axioms above, the methods **isA()** and **isOf()** are quite simple to implement:

```
% isA {
    return _self && classOf(_self) == class;
}
% isOf {
    if (_self)
    {   const struct Class * myClass = classOf(_self);

        if (class != Object)
            while (myClass != class)
                if (myClass != Object)
                    myClass = super(myClass);
                else
                    return 0;
        return 1;
    }
    return 0;
}
```

A first, very naive implementation of **cast()** would be based on **isOf()**:

```
% cast {
    assert(isOf(_self, class));
    return (void *) _self;
}
```

isOf(), and therefore **cast()**, fails for null pointers. **isOf()** believes without further inquiry that any pointer points at least to an instance of **Object**; therefore, we can be sure that **cast(Object, x)** will only fail for null pointers. However, as we shall see in section 8.5, this solution can easily backfire.

8.4 Coding Standard

The basic idea is to call **cast()** as often as necessary. When a statically linked method dereferences objects in its own class, it should do so with **cast()**:

```
void move (void * _self, int dx, int dy) {
    struct Point * self = cast(Point, _self);

    self -> x += dx, self -> y += dy;
}
```

If such a method receives objects from another class, it can still call **cast()** to make sure that the parameters are what they claim to be. We have introduced the **%casts** request of *ooc* to handle the import of a parameter list:

```
% move {
%casts
    self -> x += dx, self -> y += dy;
}
```

%casts is implemented with the report **casts** in *etc.rep*; therefore, we can control all object imports by changing this report. The original version was shown in section 7.4; here is how we introduce **cast()**:

```
% casts         // implement %casts request

`{()                            // import
  `{if `_ _
`t  `const struct  `cast  *  `name  =  ` \
                        cast( `cast ,  _ `name ); `n
  `}fi
`}n

`{if `linkage %                 // for static linkage only
  `%checks
`}fi
```

The replacement `_ is defined as an underscore if the current parameter was specified with a leading underscore, i.e., if it is in the current class. Instead of a plain assignment, we call **cast()** to check before we dereference the pointer.

The first loop at **import** takes care of all the object in a method's own class. The other objects are checked in the report **checks**:

```
% checks        // check all other object parameters

`{()
  `{ifnot `cast ` `{ifnot `_ _
`t  cast( `cast ,  `name ); `n
  `}fi `}fi
`}n
```

Originally, this loop generated **assert()** for *all* objects. Now we can restrict our attention to those objects which are not in the current class. For them we generate a call to **cast()** to make sure they are in their proper class.

The report **casts** differentiates between methods with static and dynamic linkage. Statically linked methods need to do their own checking. **casts** and **checks** generate local variables for dereferencing and statements to check the other objects, i.e., **%casts** must be used at the end of the list of local variables declared at the top of a method body with static linkage.

Dynamically linked methods are only called through selectors; therefore, the job of checking can mostly be delegated to them. **%casts** is still used to dereference the parameter objects in the current class, but it will only initialize local variables:

Circle.dc

```
% Circle draw {
%casts
    printf("circle at %d,%d rad %d\n",
            x(self), y(self), self -> rad);
}
```

Point.c

```
void draw (const void * _self) {
    const struct Point * self = _self;

    ...
```

Circle.c

```
static void Circle_draw (const void * _self) {
    const struct Circle * self = cast(Circle, _self);

    ...
```

We have to be careful: while the selector could check if an object belongs to the current class **Point**, once it calls a subclass method like **Circle_draw()** we have to check there whether or not the object really is a **Circle**. Therefore, we let the selector check the objects which are not in the current class, and we let the dynamically linked method check the objects in its own class. **casts** simply omits the call to **checks** for methods which are called through a selector.

Now we have to modify the selectors. Fortunately, they are all generated by the report **init**, but there are several cases: selectors with a **void** result do not **return** the result of the actual method; selectors with a variable argument list must pass a pointer to the actual method. **init** calls the report **selectors** in *etc.rep* which in turn delegates the actual work to the report **selector** and various subreports. Here is a typical selector:

```
int differ (const void * _self, const void * b) {
    int result;
    const struct Class * class = classOf(_self);

    assert(class –> differ);
    cast(Object, b);
    result = class –> differ(_self, b);
    return result;
}
```

This is generated by the report **selector** in *etc.rep*:*

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

`%ifmethod
`%checks
`%call
`%return
} `n `n
```

The reports **result** and **return** define and return the **result** variable, unless the return type is **void**:

```
% result        // if necessary, define result variable

`{ifnot `result void
`t  `result  result;
`}n
```

* The actual report is slightly more complicated to account for methods with a variable parameter list.

```
% return        // if necessary, return result variable

`{ifnot `result void
`t  return  result;
`}n
```

The report **ifmethod** checks if the desired method exists:

```
% ifmethod      // check if method exists

`t  assert(class  –>  `method ); `n
```

We have to be a bit careful with the report **classOf**: if a selector retrieves a method from **Class** we can rely on **classOf()** to produce a suitable class description, but for subclasses we have to check:

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

The superclass selector is similar. Here is a typical example:

```
int super_differ (const void * _class, const void * _self,
                                        const void * b) {
    const struct Class * superclass = super(_class);

    cast(Object, b);

    assert(superclass –> differ);
    return superclass –> differ(_self, b);
}
```

Once again, if we don't work with **Class** we need to check the result of **super()**. Here is the report from *etc.rep*:

```
% super–selector    // superclass selector

`%super–header  { `n
`{if `meta `metaroot    // can use super()
`t  const  struct  `meta  *  superclass  =  super(_class); `n
`} `{else               // must cast
`t  const  struct  `meta  *  superclass  =  ` \
                            cast( `meta ,  super(_class)); `n
`} `n
`%checks

`t  assert(superclass  –>  `method ); `n
`t  `{ifnot `result void return  `} \
        superclass  –>  `method \
    ( `{() `_ `name `}, `{ifnot `,... ` ,  app `} ); `n
} `n `n
```

Other objects are checked with **checks** as if the superclass selector were a method with static linkage.

Thanks to *ooc* and the reports we have established a defensive coding standard for all methods that we might implement. With the change to all selectors and with the convention of using **%casts** in all methods, we account for all objects passed as parameters: their pointers are checked upon import to the callee. As a consequence, the result of a method can go unchecked because the user of the result is expected to apply **cast()** to it.

This is reflected by the convention of using classes in the return types of our methods. For example in *List.d*:

```
Object @ addFirst (_self, const Object @ element);
```

addFirst() was shown in section 7.7 and it returns a **void ***. *ooc*, however, generates:

```
struct Object * addFirst (void * _self, const void * element) {
    struct List * self = cast(List, _self);

    cast(Object, element);

    ...
    return (void *) element;
}
```

struct Object is an incomplete type in an application program. This way the ANSI-C compiler checks that the result of a call to **addFirst()** is assigned to **void *** (to be checked later, hopefully) or that it is passed to a method expecting a **void *** which by our conventions will check it with **cast()**. In general, by a careful use of classes in the return types of methods, we can use the ANSI-C compiler to check for unlikely assignments. A class is a lot more restrictive than a **void ***.

8.5 Avoiding Recursion

In section 8.3 we tried to implement **cast()** as follows:

```
% cast {
    assert(isOf(_self, class));
    return (void *) _self;
}
```

Unfortunately, this causes an infinite loop. To understand this, let us trace the calls:

```
void * list = new(List, 1);
void * object = new(Object);

    addFirst(list, object) {
      cast(List, list) {
        isOf(list, List) {
          classOf(list) {
            cast(Object, list) {
              ifOf(list, Object) {
                classOf(list) {
```

cast() is based on **isOf()** which calls **classOf()** and possibly **super()**. Both of these methods observe our coding standard and import their parameters with **%casts**, which in turn calls **cast()** to check if the arguments are an **Object** or a **Class**,

respectively. Our implementation of **isOf()** in section 8.3 calls **classOf()** before observing the third axiom that any object at least belongs to **Object**.

How strong do we want type checking to be? If we trust our code, **cast()** is a no-op and could be replaced by a trivial macro. If we don't trust our code, parameters and all other dereferencing operations need to be checked by being wrapped in **cast()** in all functions. Everybody has to use and then believe **cast()** and, clearly, **cast()** cannot employ other functions to do its checking.

So what does **cast(class, object)** guarantee? At least the same as **isOf()**, namely that its **object** is not a null pointer and that its class description can be traced to the **class** argument. If we take the code of **isOf()**, and think defensively, we obtain the following algorithm:

```
(_self = self)  is an object
(myClass = self -> class)  is an object

if (class != Object)
    class  is an object
    while (myClass != class)
        assert(myClass != Object);
        myClass  is a class description
        myClass = myClass -> super;

return self;
```

The critical parts are in italics: which nonzero pointer represents an object, how do we recognize a class description? One way to distinguish arbitrary pointers from pointers to objects is to let each object start with a magic number, i.e., to add a component **.magic** to the class description in *Object.d*:

```
% Class  Object {
    unsigned long magic;            // magic number
    const Class @ class;            // object's description
%
        ...
```

Once the magic number is set by **new()** and in the initialization of **Class** and **Object**, we check for it with the following macros:

```
#define MAGIC   0x0effaced  // magic number for objects
// efface: to make (oneself) modestly or shyly inconspicuous

#define isObject(p) \
        ( assert(p), \
        assert(((struct Object *) p) -> magic == MAGIC), p )
```

Strictly speaking, we need not check that **myClass** is an object, but the two extra assertions are cheap. If we do not check that **class** is an object, it could be a null pointer and then we could slip an object with a null pointer for a class description past **cast()**.

The expensive part is the question whether **myClass** is a class description. We should not have very many class descriptions and we should know them all, so we could consult a table of valid pointers. However, **cast()** is one of the innermost functions in our code, so we should make it as efficient as possible. To begin with,

myClass is the second element in a chain from an object to its class description and both have already been verified to contain a magic number. If we disregard the problem of stray pointers destroying class descriptions, it is reasonable to assume that the **.super** chain among the class descriptions remains unharmed after **Class_ctor()** sets it up. Therefore, we remove the test from the loop altogether and arrive at the following implementation for **cast()**:

```
static void catch (int sig)     // signal handler: bad pointer
{
    assert(sig == 0);       // bad pointer, should not happen
}

% cast {
    void (* sigsegv)(int) = signal(SIGSEGV, catch);
#ifdef  SIGBUS
    void (* sigbus)(int) = signal(SIGBUS, catch);
#endif
    const struct Object * self = isObject(_self);
    const struct Class * myClass = isObject(self -> class);

    if (class != Object)
    {   isObject(class);
        while (myClass != class)
        {   assert(myClass != Object);  // illegal cast
            myClass = myClass -> super;
        }
    }

#ifdef  SIGBUS
    signal(SIGBUS, sigbus);
#endif
    signal(SIGSEGV, sigsegv);
    return (void *) self;
}
```

Signal processing protects us from mistaking a numerical value for a pointer. **SIGSEGV** is defined in ANSI-C and indicates an illegal memory access; **SIGBUS** (or **_SIGBUS**) is a second such signal defined by many systems.

8.6 Summary

void * is a very permissive type which we had to resort to in order to construct polymorphic methods and, in particular, our mechanism for the selection of dynamically linked methods. Because of polymorphisms, object types need to be checked at runtime, i.e., when an object appears as a parameter of a method.

Objects point to unique class descriptions; therefore, their types can be checked by comparing their class description pointers to the class descriptions known in a project. We have provided three new methods for this: **isA()** checks that an object belongs to a specific class, **isOf()** is true if an object belongs to a class or one of its subclasses, and **cast()** terminates the calling program if an object cannot be treated as a member of a certain class.

As a coding standard we require that **cast()** is used whenever an object pointer needs to be dereferenced. In particular, methods with static linkage must use **cast()** on all their object parameters, selectors use it on all object parameters not in their own class, and methods with dynamic linkage use it on all object parameters which claim to be in their own class. Result values need not be checked by their producers, because the consumer can only dereference them by using **cast()** again.

ooc provides significant assistance in enforcing this coding standard because it generates the selectors and provides the **%casts** statement for the import of parameters. **%casts** generates the necessary calls to **cast()** and should be used last in the local variable declarations of a method.

cast() cannot prove the correctness of data. However, we try to make it fairly difficult or improbable for **cast()** to be defeated. The whole point of defensive programming is to recognize that programmers are likely to make mistakes and to limit how long a mistake can go unrecognized. **cast()** is designed to strike a balance between efficiency for correct programs and (early) detection of flaws.

8.7 Exercises

Technically, superclass selectors can only be used from within methods. We could decide not to check the parameters of superclass selectors. Is that really wise?

We believe that a pointer identifies an object if the object starts with a magic number. This is expensive because it increases the size of every object. Could we only require that a class description must start with a magic number?

The fixed part of a class description (name, superclass and size) can be protected with a checksum. It has to be chosen carefully to permit static initialization for **Class** and **Object**.

cast() duplicates the algorithm of **isOf()**. Can **isOf()** be changed so that we can use the naive implementation of **cast()** and not get into an infinite recursion?

cast() is our most important function for producing error messages. Rather than a simple **assert()** the messages could contain the point of call, the expected class, and the class actually supplied.

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**-funtions 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 {               // (self, filter, fnm, buf)
%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 looses 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 –**lMalloc-Debug**, the standard versions of **malloc()** and related functions are replaced by a module that can communicate with the *MallocDebug* application. We start *Malloc-Debug* 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.

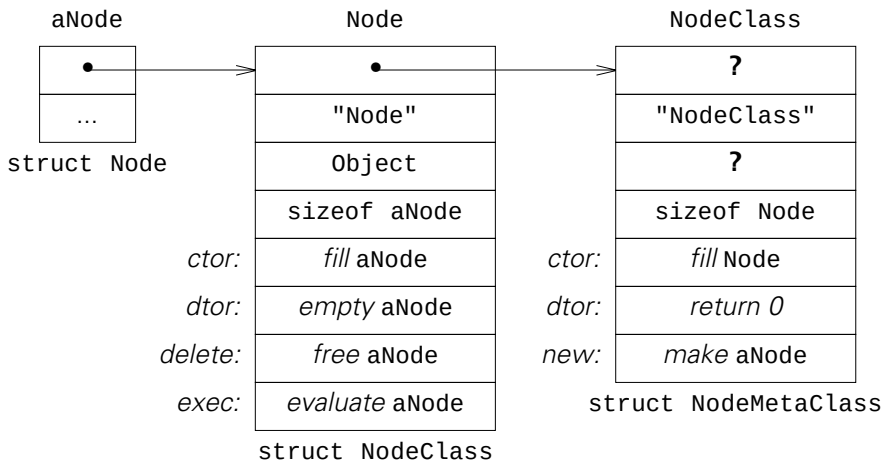
11.2 Class Methods

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

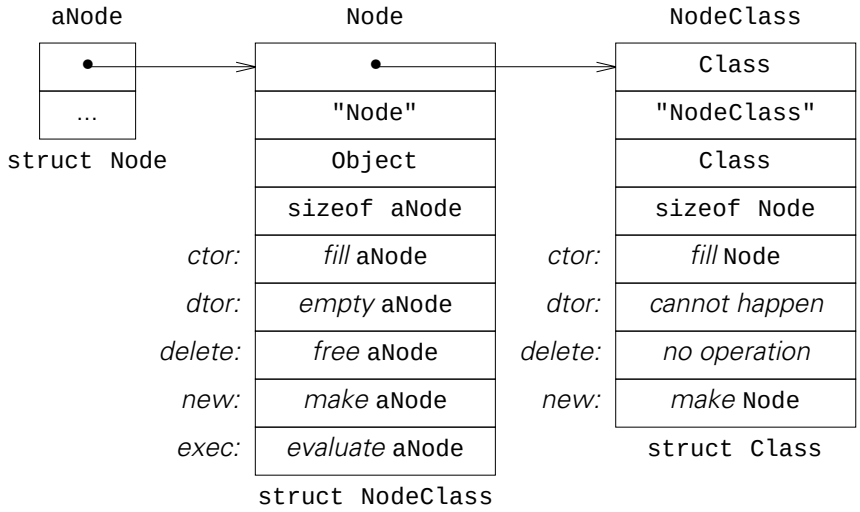
The internal difference between a class method and another dynamically linked method is encapsulated in the selector. Consider **exec()**, a dynamically linked method to evaluate a node. The selector applies **classOf()** to get the class description and looks for **.exec** in there:

```
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 superclass 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		*see below*
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	*see below*
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.

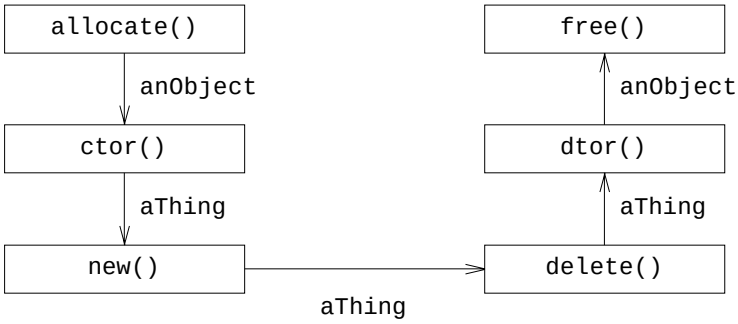
Just like other methods, class methods can have static or dynamic linkage, but the syntax of *ooc* only permits static linkage for class methods that apply to the root metaclass. Therefore, the term *class method* has been introduced here to only describe a method with dynamic linkage that is applied 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 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\n\tlex %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. Metaclasses 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		%- dtor, puto, geto
Minus		%- exec
Dyad	left, right	%- ctor
Ref		%- dtor, puto
Assign		%- geto, exec
Builtin		%- geto, exec
User		%- geto, exec
Binary		%- dtor, 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.

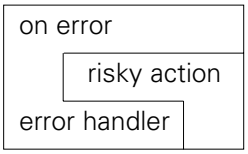
value can be extended with control structures so that functions are more powerful. In this case **stmt()** needs to be split into true statements such as **let** and commands like **save**, **load**, or **def**.

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:



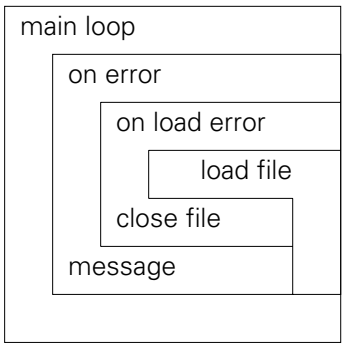
First, an error handler is set up. Either the risky action completes correctly or the error handler gets a chance to clean up before the compound action completes. In ANSI-C, **setjmp()** and **longjmp()** are used to implement this error recovery scheme:

```
#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);
    vfprintf(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

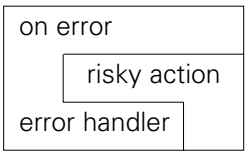
```
$ except
caused –1
caused 1
caused 2
caused 3
caused 4
```

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.

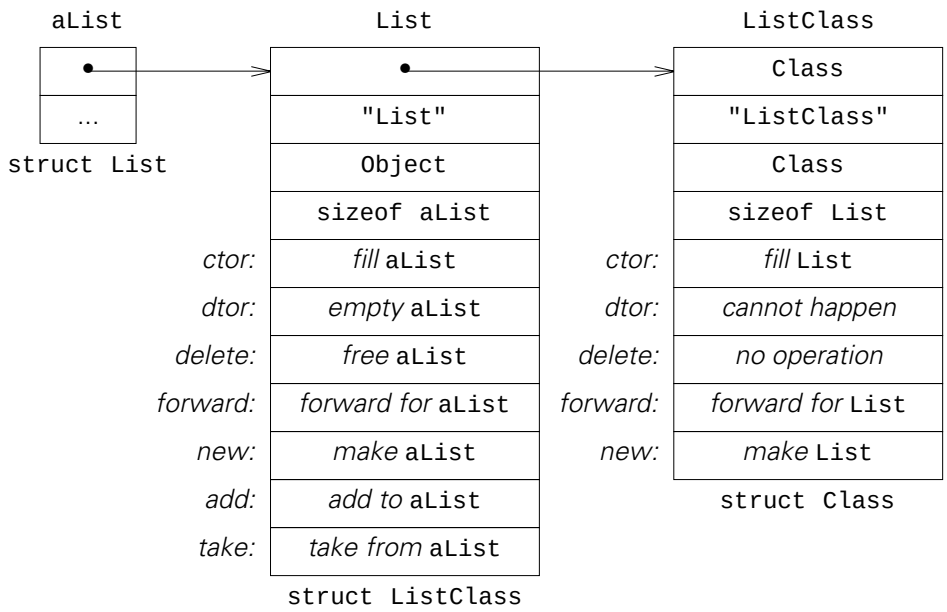
classOf is the other report that gets modified significantly. A call to **cast()** is removed, but the interesting question is what happens if a call to a class method needs to be forwarded. Let us look at the selector which *ooc* generates for **new()**:

```
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
`}
```

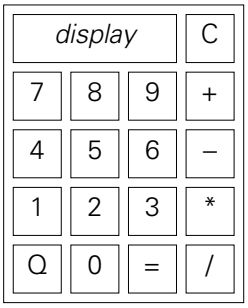
```
`} `{else
`t const struct `meta * class = ` \
                    cast( `metaroot (), _self); `n
`} `n
```

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:



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`			information to pass
	`kind`		type of data
	`data`		text, position, etc.
`Ic`			base class for application
	`out`		object I am connected to
		`wire`	connect me to another object
		`gate`	send information to `out`
`LineOut`			model the display
		`wire`	not used
		`gate`	display incoming information
`Button`			model an input device
	`text`		label, defines interesting information
		`gate`	look at incoming information: if it matches `text`, send it on
`Calc`			computing chip
	state		current value, etc.
		`gate`	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.

Ic.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 **Ic** 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 — *Ic*

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

```
% Ic 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 **Ic** to another object by storing the object address in **out**:

```
% Ic 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 **Ic** 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()
digit	*any*	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	*any*	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.

```
Object                Object
  Ic                    Ic
    Button                Button
      CButton
    LineOut               LineOut
      CLineOut
    Crt                   Crt
                            CButton
                            CLineOut
```

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 **Ic** 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 **Ic** class. Our class hierarchy forces us to compromise and (more or less) use two **Ic** objects in a **CButton**. If the next project does not use the **Ic** class, we cannot reuse the code developed to hide the details of *curses*.

14.6 A Graphical Interface — *Xt*

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 *widgets*.

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 **Ic**, 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**:

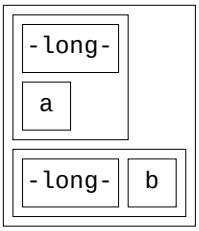
```
% XButton forward {
%casts
    if (selector == wire)
        wire(va_arg(* app, void *), self -> button);
    else
        super_forward(XButton(), _self, result,
                                    selector, name, app);
}
```

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 **Ic** class developed for our calculator. Message forwarding lets classes like **XButton** function as if they were descendants of **Ic**. 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:

```
double sqrt (double arg)        ANSI-C
{ ... }

double sqrt (arg)               classic
  double arg;
{ ... }
```

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:

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

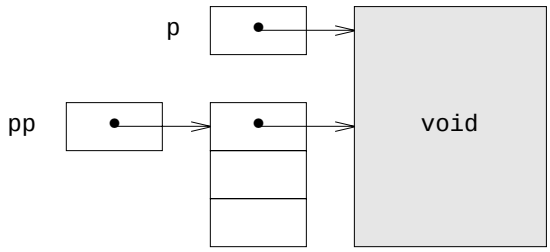
%p is used as a format specification for **printf()** and (theoretically) for **scanf()** to write and read pointer values. The corresponding argument type is **void *** and thus any pointer type:

```
void * vp;
  printf("%p\n", vp);           display value
  scanf("%p", & vp);            read value
```

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

```
void * p, ** pp;
  p + 1                         wrong
  pp + 1                        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:

```
const void * vp;                vp [•] -> const void

int * ip;
int * const p = ip;             ok for local variable

  vp = ip;                      ok, blocks assignment
  ip = vp;                      wrong, allows assignment
  ip = (void *) vp;             ok, brute force
  * (const int **) & ip = vp;   ok, overkill
  p = ip;                       wrong, pointer is blocked
  * p = 10;                     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:

`ap = cp;`	wrong
`c.x, cp -> x`	wrong
`cp -> a.x`	ok, fully specified
`((struct a *) cp) -> x`	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:

`void *  f  (void *);`	function
`void * (* fp) (void *) = f;`	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:

```
int x;
  f     (& x);
  fp    (& x);
  (* fp)(& x);
```

using a function name	
using a pointer, ANSI-C	
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 is 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:

`size_t`	result of `sizeof`
`ptrdiff_t`	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 **OOCPATH** 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.

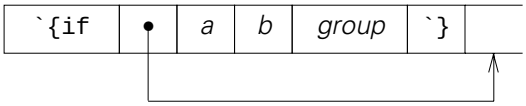
Reports are generated by calling **gen()** with the name of a report. A certain effort is made to avoid emitting leading spaces or more than one empty line in a row: a global variable **newLine** is 0 at the left margin or 1 once we have printed something; an internal function **lf()** prints a newline 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.

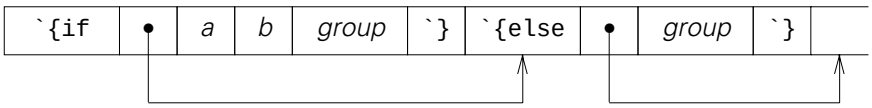
If a single blank or tab character is used as the split character, and if we emit a single blank for an empty word, a report closely resembles the generated output: two blanks represent one blank in the output. Generation is reasonably efficient if we can quickly identify words to be left unchanged (they 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 *option*s 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.

–**l**
: 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 introduced 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 *declarator*s 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 *class*	dynamic and class methods defined for *class*
`{struct *class*	components for *class*
`{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

`	no text (empty string)
``	` (back quote)
`t	tab
`n	newline (limited to one blank line)

set up once class descriptions are loaded

`desc	last *description* from command line
`root	root class' name
`metaroot	root's metaclass name

set up for a class **% %– %+ `{dcl `{prot `{pub `{super**

`class	class' name
`super	class' superclass name
`meta	class' metaclass name
`supermeta	metaclass' superclass name

set up for a method **`{% `{%– `{%+ `{links** *class*

`method	method's name
`result	method's result type
`linkage	method's linkage: **%**, **%–**, or **%+**
`tag	method's tag
`,...	**, ...** if variable arguments are declared, empty if not
`_last	last parameter's name if variable arguments, undefined if not

set up for a declarator **`{()** **`{struct** *class*

`name	name in declarator
`const	**const** followed by a blank, if so declared
`type	**void *** for objects, declared type otherwise
`_	_ if used in declaration, empty otherwise
`cast	object's class name, empty otherwise

set up for lines from the description file **`{dcl `{prot `{pub**

`class	set up for a class description, empty otherwise
`line	line's text if not class description, undefined otherwise
`newmeta	**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

OOCPATH 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	*class*.**d**	description for *class*
	class.**dc**	implementation for *class*
	report.**rep**	report file
	*AWKPATH/ **.**awk*	modules
	*AWKPATH/ **.**dbg*	debugger modules
	OOCPATH/**c.rep**	implementation file reports
	OOCPATH/**dc.rep**	implementation thunks report
	OOCPATH/**etc.rep**	common reports
	OOCPATH/**h.rep**	interface file report
	OOCPATH/**header.rep**	common reports
	OOCPATH/**m.rep**	*makefile* dependency report
	OOCPATH/**r.rep**	representation file reports
	OOCPATH/**va.rep**	common reports
	OOCPATH/**[chr]-R.rep**	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
  Ic                             IcClass
    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.

Ic(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)

IcClass Crt: Ic — input/output objects for curses

Objct
  Ic
    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: Objct — 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 Ic(4)

IcClass: Class Ic: Objct — basic input/output/transput objects

Objct
Ic
Button
Calc
LineOut
Mux

new(Ic());
new(Button(), "*text***");**
new(Calc());
new(LineOut());
new(Mux());

%–
void wire (Objct @ *to***,** *self***)**
enum { reject, accept } gate (*self***, const void *** *item***)**

An **Ic** object has an output pin and an input action. **wire()** connects the output to some other object. If an **Ic** object is sent a data *item* with **gate()**, it will perform some action and send some result to its output pin; some **Ic** objects only create output and others only consume input. **gate()** returns **accept** if the receiver accepts the data.

Ic is a base class. Subclasses overwrite **gate()** to implement their own processing. **Ic_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

Objct
- **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.