9 Memory management

Frankly, it would be nice to forget about memory.

Our programs would just create objects as they please. One after the other, unused objects would vanish into abysses, while those most needed would slowly move closer to the top, like meritorious employees of a large corporation who manage once in a while to catch the attention of a higher officer, and by making themselves indispensable to their immediate superiors will with a bit of luck, at the end of a busy career, be admitted into the inner circle.

But it is not so. Memory is not infinite; it does not harmoniously organize itself into a continuous spectrum of storage layers with decreasing access speeds, to which objects would naturally distribute. We do need to fire our useless employees, even if we must call it early retirement imposed with regret because of the overall economic situation. This chapter examines who should be thus downsized, how, and by whom.

9.1 WHAT HAPPENS TO OBJECTS

Object-oriented programs create objects. The previous chapter showed how useful it is to rely on dynamic creation to obtain flexible object structures, which automatically adapt to the needs of a system's execution in any particular case.

Object creation

We have seen the basic operation for allocating space to new objects. In its simplest form it appears as

!! *x*

and its effect was defined as threefold: create a new object; attach it to the reference *x*; and initialize its fields.

A variant of the instruction calls an initialization procedure; and you can also create new objects through routines *clone* and *deep_clone*. Since all these forms of allocation internally rely on basic creation instructions, we can restrict our attention to the form !! *x* without fear of losing generality.

We will now study the effect of such instructions on memory management.

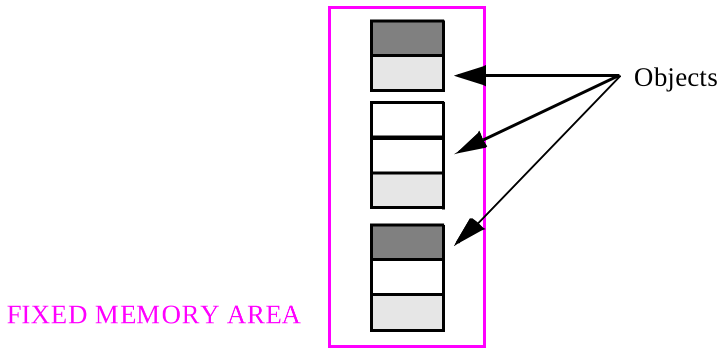
Three modes of object management

First it is useful to broaden the scope of the discussion. The form of object management used for object-oriented computation is only one of three commonly found modes: **static**, **stack-based** and **free**. The choice between these modes determines how an entity can become attached to an object.

Recall that an entity is a name in the software text representing a run-time value, or a succession of run-time values. Such values are either objects or (possibly void) references to objects. Entities include attributes, formal routine arguments, local entities of routines and *Result*. The term **attached** describes associations between entities and objects: at some stage during execution, an entity *x* is attached to an object O if the value of *x* is either O (for *x* of expanded type) or a reference to O (for *x* of reference type). If *x* is attached to O, it is sometimes convenient to say also that O is attached to *x*. But whereas a reference is attached to at most one object, an object may be attached to two or more references; this is the problem of dynamic aliasing, discussed in the previous chapter.

In the static mode, an entity may become attached to at most one run-time object during the entire execution of the software. This is the scheme promoted by languages such as Fortran, designed to allow an implementation technique which will allocate space for all objects (and attach them to the corresponding entities) once and for all, at program loading time or at the beginning of execution.

The static mode



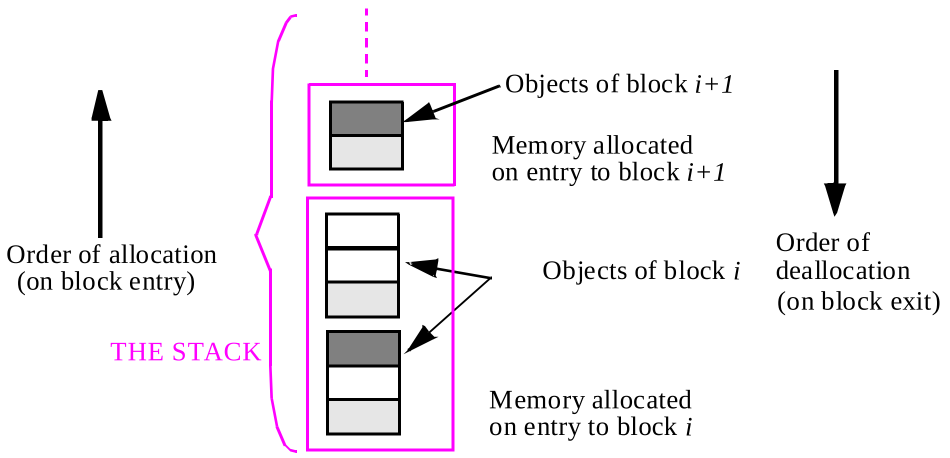
The static mode is simple and supports efficient implementation on the usual computer architectures. But it presents serious limitations:

- It precludes recursion, since a recursive routine must be permitted to have several incarnations active at once, each with its own incarnations of the routine's entities.

- It also precludes dynamically created data structures, since the compiler must be able to deduce the exact size of every data structure from the software text. Each array, for example, must be statically declared with its exact size. This seriously limits the modeling power of the language: it is impossible to handle structures that grow and shrink in response to run-time events, except by allocating the maximum possible space for each of them — a technique that wastes memory, and is rather dangerous since just one data structure may cause the whole system execution to fail if its size has been underestimated.

The second scheme of object allocation is the stack-based mode. Here an entity may at run time become attached to several objects in succession, and the run-time mechanisms allocate and deallocate these objects in last-in, first-out order. When an object is deallocated, the corresponding entity becomes attached again to the object to which it was previously attached, if any.

The stack-based mode

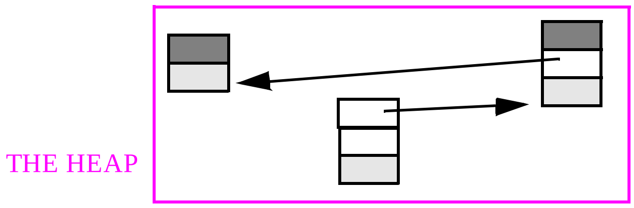

Objects of block *i+1*
Memory allocated on entry to block *i+1*
Order of allocation (on block entry)
Objects of block *i*
Order of deallocation (on block exit)
THE STACK
Memory allocated on entry to block *i*

Dynamic arrays can be created in C through the malloc function, a mechanism of the "free" kind, the mode studied next; some Pascal extensions support dynamic arrays.

Stack-based object management was made popular by Algol 60 and is supported (often in conjunction with one or both of the other two modes) in most posterior programming languages. Stack-based allocation supports recursion and, if the language permits it, arrays whose bounds only become known at run time. In Pascal and C, however, the mechanism only applies to variables of basic types and record types — not to arrays as it did in Algol. In practice the data structures that developers would most often want to allocate in this fashion are precisely arrays. Even when it applies to arrays, stack-based allocation still does not support complex data structures in their full generality.

To obtain such general data structures, we need the third and last scheme: the free mode, also called heap-based because of the way it is implemented. This is the fully dynamic mode in which objects are created dynamically through explicit requests. An entity may become successively attached to any number of objects; the pattern of object creations is usually not predictable at compile time. Objects may, furthermore, contain references to other objects.

The free (heap-based) mode



The free mode allows us to create the sophisticated dynamic data structures which we will need if, as discussed in the previous chapter, we are to take our software systems to their full modeling power.

Using the free mode

The free mode is clearly the most general, and is required for object-oriented computation. Many non-O-O languages use it too. In particular:

- Pascal uses the static mode for arrays, the stack-based mode for variables of type other than array or pointer, and the free mode for pointer variables. In the last case object creation is achieved by a call to a special creation procedure, *new*.
- C is similar to Pascal but in addition offers static non-array variables and free arrays. Dynamic allocation of pointer variables and arrays relies on a library function, *malloc*.
- PL/I supports all modes.
- Lisp systems have traditionally been highly dynamic, relying for the most part on the free mode. One of the most important Lisp operations, used repeatedly to construct lists, is *CONS*, which creates a two-field cell, ready to serve as a list element with the element's value in the first field and a pointer to the next element in the second field. Here *CONS*, rather than explicit creation instructions, will be the principal source of new objects

Space reclamation in the three modes

The ability to create objects dynamically, as in the stack-based and free modes, raises the question of what to do when an object becomes unused: is it possible to reclaim its memory space, so as to use it again for one or more new objects in later creation instructions?

In the static mode, the problem does not exist: for every object, there is exactly one attached entity; execution needs to retain the object's space as long as the entity is active. So there is no possibility for reclamation in the proper sense. A related technique is, however, sometimes used. If you are convinced that the objects attached to two entities will never be needed at the same time, if these entities need not retain their values between successive uses, and if space efficiency is a critical problem, you can assign the same memory location to two or more entities — if you are really sure of what you are doing. This technique, known as **overlay** is still, appallingly enough, practiced manually.

If used at all, overlay should clearly be handled by automatic software tools, as the potential for errors is too high when programmers control the process themselves. Once again a major problem is change: a decision to overlay two variables may be correct at a certain stage of the program's evolution, but an unexpected change may suddenly make it invalid. We will encounter similar problems below, in a more modern context, with garbage collection.

With the stack-based mode, the objects attached to an entity may be allocated on a stack. Block-structured language make things particularly simple: object allocation occurs at the same time for all entities declared in a given block, allowing the use of a single stack for a whole program. The scheme is elegant indeed, as it just involves two sets of concomitant events:

Allocation and deallocation in a block-structured language

Dynamic Property (event at execution time)	Static Property (location in the software text)	Implementation Technique
Object allocation	Block entry.	Push objects (one for each of the entities local to the block) onto stack.
Object deallocation	Block exit.	Pop stack.

The simplicity and efficiency of this implementation technique are part of the reason why block-structured languages have been so successful.

With the free mode, things cease to be so simple. The problem comes from the very power of the mechanism: since the pattern of object creation is unknown at compile time, it is not possible to predict when a given object may become useless.

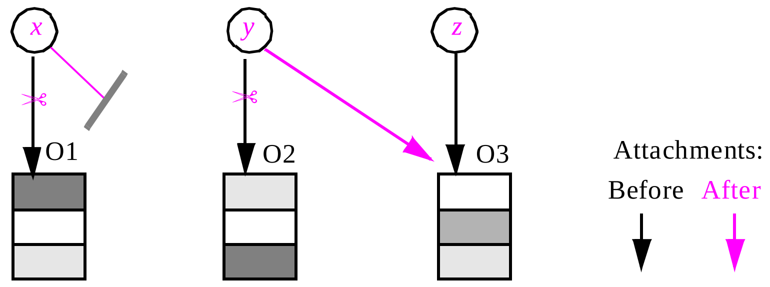
Detachment

Objects may indeed, in the free mode, become useless to the software at unpredictable times during execution, so that some mechanism (to be determined later in this discussion) may reclaim the memory they occupy.

The reason is the presence in our execution mode of operations performing what may be called **detachment** — the reverse of attachment. The previous chapter studied at length how entities can become attached to objects, but did not examine in any detail the consequences of detachments. Now is the time to correct this.

Detachment only affects entities *x* of reference types. If *x* is of expanded type, the value of *x* is an object O, and there is no way to detach *x* from O. Note, however, that if *x* is an expanded attribute of some class, O represents a subobject of some bigger object BO; then BO, and with it O, may become unreachable for any of the reasons studied below. So for the rest of this chapter we may confine our attention to entities of reference types.

Detachment



The principal causes of detachment are the following, assuming *x* and *y*, entities of reference type, were initially attached to objects O1 and O2. The figure illustrates cases D1 and D2.

D1 • An assignment of the form *x* := *Void*, or *x* := *v* where *v* is void, detaches *x* from O1.

D2 • An assignment of the form *y* := *z*, where *z* is attached to an object other than O2, detaches *y* from *O2*.

D3 • Termination of a routine detaches formal arguments from any attached objects.

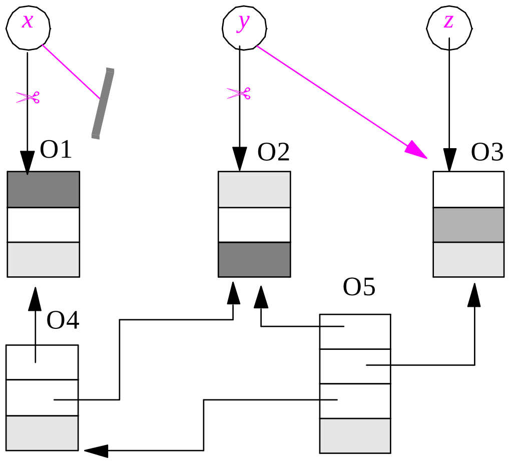
D4 • A creation instruction !! *x*, attaches *x* to a newly created object, and hence detaches *x* if it was previously attached to an object O1.

Case D3 corresponds to the rule given earlier that the semantics of an assignment *a* := *b* is exactly the same as that of initializing a formal argument *a* of a routine *r* at the time of a call *t*•*r* (…, *b*, …), where the position of *b* in the call corresponds to that of *a* in the declaration of *r*.

Unreachable objects

Does detachment mean that the detached object — O1 or O2 on the preceding figure — becomes useless and hence that the runtime mechanisms can reclaim the memory space it occupies, then recycle it for other objects? That would be too easy! The entity for which an object was initially created may have lost all interest in it, but because of dynamic aliasing other references may still be attached to it. For example the last figure may have shown only a partial view of attachments; looking at a broader context might reveal that O1 and O2 are still reachable from other objects:

Detachment is not always death



But this is still not the entire object structure. By getting even more context, we might now discover that O4 and O5 are themselves useless, so that in the absence of other references O1 and O2 are not needed after all.

So the answer to the question "what objects can we reclaim?" must follow from a global analysis of the entire set of objects created so far. We can identify three kinds of object:

C1 • Objects directly attached to entities of the software text, known (from the language rules) to be needed.

C2 • Dependents of objects of category C1. (Recall that the direct dependents of an object are those to which it has references; here we are considering both direct and indirect dependents.)

C3 • Objects which are in neither of the preceding two categories.

The objects of category C1 may be called the **origins**. Together with those of category C2, the origins make up the set of **reachable** objects. Those of category C3 are **unreachable**. They correspond to what was informally called "useless objects" above. A more lively if somewhat macabre terminology uses the term "dead objects" for C3, the origins and their dependents being then called "live objects". (Computing scientists, however, have not quite managed to reconcile their various metaphors, as the process of reclaiming dead objects, studied below, is called "garbage collection".)

The term "root" is also used for "origin". But here the latter is preferable because an O-O system also has a "root object" and a root class. The resulting ambiguity would not be too damaging since the root object, as seen below, is indeed one of the origins.

The first step towards addressing the problem of memory management under the free mode is to separate the reachable objects from the unreachable ones. To identify reachable objects, we must start from the origins and repeatedly follow all references. So the first question is to identify the origins; the answer depends on the run-time structure defined by the underlying language.

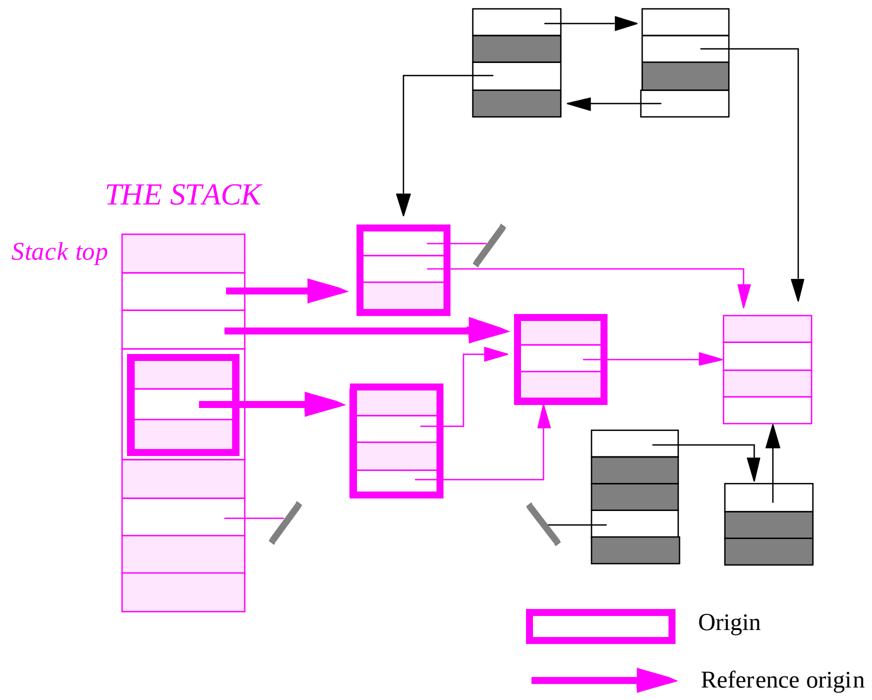
Reachable objects in classical approaches

Because the unreachability problem is already present in the run-time structure of such classical approaches as Pascal, C and Ada, it is interesting to start with this case. (More accurately, this is interesting for the reader who is familiar with one of these approaches. If you are not in this category, you may prefer to skip this section and go directly to the next one, which moves right on to the run-time structure of O-O software.)

The approaches quoted combine the stack-based and free modes of allocation. C and Ada also support the static mode, but to keep things simple we may ignore static allocation by viewing it as a special case of stack-based allocation: we treat static objects as if they were allocated once and for all, when execution starts, at the bottom of the stack. (This is indeed the way Pascal developers emulate static entities: they declare them in the outermost block.)

Another common property of these approaches is that entities may denote pointers. To provide a better preparation for the object-oriented approach of this book, where instead of pointers we use references (a more abstract notion, as discussed in the previous chapter), let us pretend that the pointers in question are actually references. This means in particular that we disregard the weakly typed nature of pointers in C.

With these assumptions and simplifications the origins, shown with thick borders on the following figure, are all the objects which are either allocated on the stack or attached to references allocated on the stack. The reachable objects (including the origins) appear in color, the unreachable objects in black.



Live objects (in color) and dead objects (in black) in a combined stack-based and free model

Because the unreachability problem only arises for objects allocated under the free mode, and such objects are always attached to entities of reference types, it is convenient to ignore the reclamation problem for objects allocated on the stack (which can be handled simply by popping the stack at the time of block exit) and to start from the references coming from the stack. We may call these references **reference origins**. They are shown with thick arrows in the figure. A reference origin is either:

O1 • The value of a local entity or routine argument of reference type (as with the top two reference origins in the figure).

O2 • A field of reference type, in an object allocated on the stack (as with the lowest reference origin in the figure).

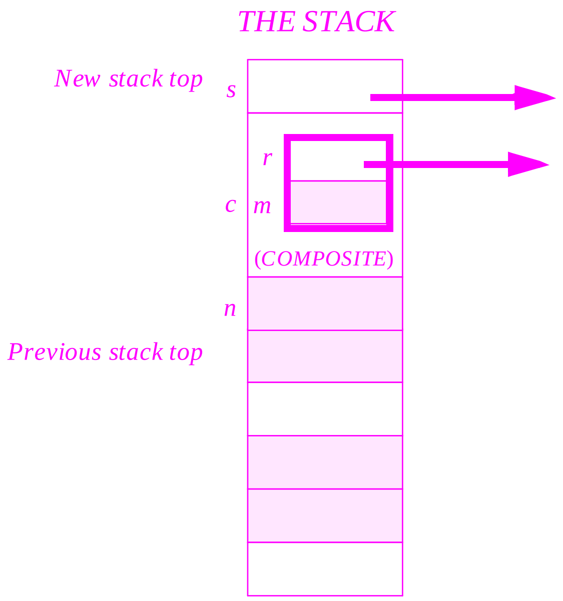
As an example, consider the following type and procedure declarations, written in a syntax half-way between Pascal and the notation of the rest of this book (an entity of type **reference** *G* is a reference that may become attached to objects of type *G*):

```
type
    COMPOSITE =
        record
            m: INTEGER
            r: reference COMPOSITE
        end
    …
```

```
procedure p is
    local
        n: INTEGER
        c: COMPOSITE
        s: reference COMPOSITE
    do
        …
    end
```

Every execution of *p* allocates three values on the stack:

Entity allocation for a procedure



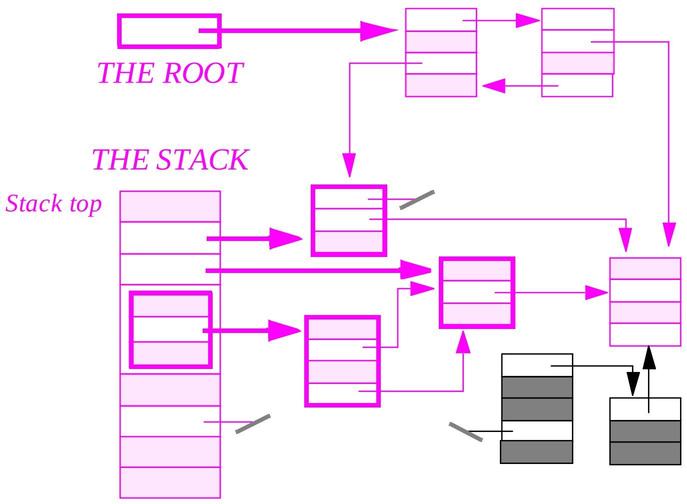
The three new values are an integer *n*, which does not affect the problem of object management (since it will disappear when the procedure terminates, and does not refer to any other object); a reference *s*, which is an example of category O1; and an object *c* of type *COMPOSITE*. This object is itself stack-based and its allocated memory may be reclaimed on procedure termination; but it contains a reference field for *r*, which is an example of category O2.

In summary, to determine the reachable objects in a classical approach combining the stack-based and free modes, you can start from the references on the stack (variables of reference types, and reference fields of composite objects), and repeatedly follow all reference fields of the attached objects if any.

Reachable objects in the object-oriented model

The object-oriented run-time structure presented in the preceding chapter has a few differences from the one just discussed.

Reachability in the object-oriented model



The execution of any system starts with the creation of one object, called the root object of the system, or just its root (when there is no confusion with the root class, a static notion). Clearly, the root is one of the origins in this case.

Another set of origins arises because of the possible presence of local entities in a routine. Assume a routine of the form

```
some_routine is
        local
                rb1, rb2: BOOK3
                eb: expanded BOOK3
        do
                …
                !! rb1
                … Operations possibly involving rb1, rb2 and eb …
        end
```

Whenever a call to *some_routine* is executed, and for the duration of that execution, the instructions in the routine's body may refer to *rb1*, *rb2* and *eb*, and hence to the attached objects if any. (For *eb* there is always an attached object, but at various points *rb1* and *rb2* may be void.) This means that such objects must be part of the reachable set, even though they are not necessarily dependents of the root.

Local entities of reference types, such as *rb1* and *rb2*, are similar to the local routine variables which, in the previous model, were allocated on the stack. Local entities of expanded types, such as *eb*, are similar to the stack-based objects.

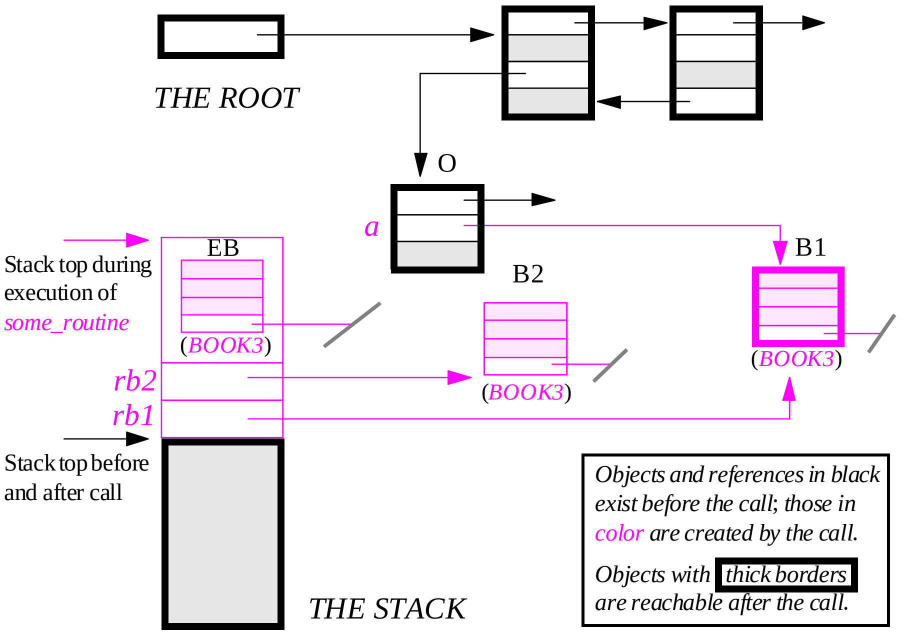
When a call to *some_routine* terminates, the current incarnations of entities *rb1*, *rb2* and *eb* disappear. As a result, any attached objects cease to be part of the origin set. This does not necessarily mean that they become unreachable, as they may in the meantime have become dependents of the root object or other origins.

Assume for example that *a* is an attribute of the enclosing class and that the whole text of the routine is:

```
some_routine is
    local
        rb1, rb2: BOOK3
        eb: expanded BOOK3
    do
        !! rb1; !! rb2
        a := rb1
    end
```

The following figure shows in color the objects that a call to *some_routine* will create and the references that it will reattach.

Objects attached to local entities


THE ROOT
O
a
EB
B2
B1
Stack top during execution of *some_routine*
(*BOOK3*)
(*BOOK3*)
(*BOOK3*)
rb2
rb1
Stack top before and after call
THE STACK
Objects and references in black exist before the call; those in color are created by the call.
Objects with thick borders are reachable after the call.

When a call to *some_routine* terminates, the object O that served as target of the call is still reachable (otherwise there would have been no call!). The *a* field of O is now attached to the *BOOK3* object B1 created by the first creation instruction (the one of target *rb1*), which, then, remains reachable. In contrast, the objects B2 and EB that were attached to *rb2* and *eb* during the call now become unreachable: with the routine text as given there is no possibility that any of the other objects of the system, reachable or not, could "remember" B2 or EB.

The memory management problem in the object-oriented model

We may summarize the preceding analysis by defining the origins, and hence of the reachable objects, in the object-oriented framework:

Definition: origins, reachable and unreachable objects

At any point during the execution of a system, the set of **origins** is made of the following objects:

- The system's root object.
- Any object attached to a local entity or formal argument of a routine currently being executed (including the local entity *Result* for a function).

Any dependent, direct or indirect, of these origins is **reachable**. Any other object is unreachable; it is possible to reclaim the memory it occupies (for example to recycle it for other objects) without affecting the correct semantics of the system's execution.

The problem of memory management arises from the unpredictability of the operations which affect the set of reachable objects: creation and detachment. Because these operations are instructions, appearing as part of a system's control structures, there is usually no way to determine with certainty, from a mere examination of the software text, the pattern of object creation and detachment at run time.

More precisely, such a prediction is possible in some cases, for data structures managed in a strictly controlled way. An example is the *LINKED_LIST* library class studied in a later chapter, with the associated class *LINKABLE* which describes linked list elements. Instances of *LINKABLE* are only created through specific procedures of *LINKED_LIST*, and can only become unreachable as a result of executing the *remove* procedure of that class. For such classes one might envision specific reclamation procedures. (This approach will be explored later in this chapter.)

About linked lists see "Linked list representation", page 774, and subsequent sections.

But such examples, although important, are only special cases. In the most general case we must face a difficult question: what do we do about unreachable objects?

The three answers

Three general attitudes are possible as to objects that become unreachable:

- Ignore the problem and hope that there will be enough memory to accommodate all objects, reachable or not. This may be called the **casual approach**.
- Ask developers to include in every application an algorithm that looks for unreachable objects, and give them mechanisms to free the corresponding memory. This approach is called **manual reclamation**.

- Include in the development environment (as part of the so-called runtime system) automatic mechanisms that will detect and reclaim unreachable objects. This is called **automatic garbage collection**.

The rest of this chapter discusses these approaches.

9.2 THE CASUAL APPROACH

The first approach consists in forgetting about the problem: abandon dead objects to their fate. Execute creation instructions as needed, and do not worry about what may later happen to those objects that have thus been allocated.

Can the casual approach be justified?

One case in which the casual approach presents no particular problem is that of systems that do not create many objects, such as small-scale tests or experiments.

More interesting is the case of systems that may in fact create many objects, but in such a way that it is possible to guarantee that none or very few of them become unreachable. As with the static allocation scheme, no objects are ever retired; the difference is that creation occurs at execution time.

This case provides a good justification for the casual approach, as there is no need for reclamation. The number of objects created may still be too big for the available memory, but no reclamation policy would alleviate the problem if there is nothing to reclaim.

Some real-time programs follow this scheme: for efficiency reasons, they create all needed objects statically or at initialization time, avoiding any non-predictable patterns of dynamic object creation.

This method has its advocates, who usually are involved in the construction of "hard-real-time" systems demanding guaranteed sub-millisecond response times to external events (such as a missile detection), and who as a consequence insist that the time to execute every operation must be fully predictable. But then memory management is only a small part of what we must give up: predictability requires the absence of any kind of object allocation (creation instruction, *malloc*, recursion, possibly any call of a routine with local entities) after initialization; and it assumes a dedicated, single-user, single-processing machine, with no preemptive operating system call and in fact no operating system in the usual sense of the term. In such environments people sometimes choose to program in assembly language, as they fear the additional unpredictability of compiler-generated code. All this, of course, restricts the discussion to a tiny (although strategic) part of the software development world.

Do we care about memory any more?

Another argument sometimes heard to justify the casual approach is the increasing availability of large memory spaces, and the decreasing cost of memory.

The memory involved may be virtual as well as real. On a virtual memory system, both primary and secondary memory are divided into blocks called pages; when primary memory is needed, blocks of primary memory that have not been frequently used are moved to secondary memory ("paged out"). If such a system is used to run object-oriented systems, pages that contain reachable objects will tend to be paged out and leave main memory space to frequently used ones.

If we indeed had almost infinite amounts of almost free memory, we could satisfy ourselves (as suggested at the very beginning of this chapter) with the casual approach. Unfortunately this is not the case.

One reason is that in practice virtual memory is not really equivalent to real memory. If you store large numbers of objects in virtual memory, where a minority of reachable objects are interspersed with a majority of unreachable ones, the system's execution will constantly cause pages to be moved in and out, a phenomenon known as **thrashing** which leads to dramatic degradation of time performance. Indeed, virtual memory systems make it harder to separate the space and time aspects of efficiency.

See "Efficiency", page 9.

But there is a more serious limitation to the casual approach. Even systems with a large memory have limits; it is always surprising to see how quickly programmers will reach them. And as was pointed out in the more general discussion of efficiency, hardware advances — in time or in space — should be put to good use. Larger memories are bought to be used, not wasted.

As soon as you move beyond the case discussed above in which it is possible to prove that only a small number of objects will become unreachable, you will have to face the reclamation problem.

A byte here, a byte there, and soon we will be talking real corpses

It is time to lend our ears to the sad and edifying story of the London Ambulance Service.

For the source of this information and of the quotes which follow, see the bibliographic notes, page 315.

The London Ambulance Service, said to be the largest in the world, serves an area of about 1500 square kilometers, a resident population of almost seven million people and an even larger daytime population. Every day it handles over five thousand patients and receives between two and three thousand calls.

As you may have guessed from the somber tone of this introduction, computers (and more to the point computer software) got involved at some stage. At more than one stage, in fact: several attempted systems were discarded as inadequate without being ever put into actual use, the latest in 1991, having burned seven and half million pounds. Then in 1992 a new system, developed at a cost of a million pounds, was put into operation. It soon made headlines again; on October 28 and 29, television and press reports were announcing that twenty lives had been lost because of the system's inadequacy; in one particular case an ambulance crew is said to have radioed base on reaching the location of their call, to ask why the undertaker had got there first. The Service's chief executive resigned and an inquiry commission was appointed.

The Service did not immediately scrap the computerized system but switched to a hybrid mode — partly manual, partly relying on the system. According to the official report:

"XX" in this quotation and the next stands for the software company (named in the report) which produced the system.

This [*hybrid*] *system operated with reasonable success from the afternoon of 27 October 1992 up to the early hours of 4 November. However, shortly after 2AM on 4 November the system slowed significantly and, shortly after this, locked up altogether. Attempts were made to re-boot* (*switch off and restart workstations*) *in the manner that staff had previously been instructed by XX to do in these circumstances. This re-booting failed to overcome the problem with the result that calls in the system could not be printed out and mobilizations via* [*the system*] *from incident summaries could not take place. Management and staff* […] *reverted fully to a manual, paper-based system with voice or telephone mobilization.*

What caused the system to fail in such a dismal way that it could not be kept even as an adjunct to a manual operation? The inquiry report identifies several reasons, but here is the clincher:

The Inquiry Team has concluded that the system crash was caused by a minor programming error.

In carrying out some work on the system some three weeks previously the XX programmer had inadvertently left in the system a piece of program code that caused a small amount of memory within the file server to be used up and not released every time a vehicle mobilization was generated by the system.

Over a three week period these activities had gradually used up all available memory thus causing the system to crash. This programming error should not have occurred and was caused by carelessness and lack of quality assurance of program code changes. Given the nature of the fault it is unlikely that it would have been detected through conventional programmer or user testing.

The reader will be the judge of how accurate it is to call the programming error "minor", especially in view of the crucial last comments (that the error would have been hard to find through testing), which will be discussed again below.

For anyone wondering whether the casual approach may be good enough, and more generally for anyone who may be tempted to dismiss memory management as "just an implementation issue", the twenty victims of the London Ambulance Service will serve as a sobering reminder of the seriousness of the problems covered by this chapter.

9.3 RECLAIMING MEMORY: THE ISSUES

If we go beyond the casual approach and its simplistic assumptions, we must find how and when to reclaim memory. This in fact involves two issues:

- How we will find out about dead elements (**detection**).
- How the associated memory is actually reclaimed (**reclamation**).

For each of these tasks, we may look for a solution at any one of two possible levels:

- The language implementation level — compiler and runtime system, providing the support common to all software written in a certain language in a certain computing environment.
- The application level — application programs, intended to solve specific problems.

In the first case the selected memory management functions will be handled automatically by the hardware-software machine. In the second case, each application developer has to take care of these functions on his own.

There is in fact a third possible level, in-between these two: working at the **component manufacturing** level, that is to say handling memory management functions in the general-purpose reusable library classes in an object-oriented environment. As at the application level, you can only use the programming language's official mechanisms (rather than enjoying direct access to hardware and operating system facilities); but as at the language implementation level, you can address the memory management problem, or part of it, once and for all for all applications.

Given two tasks and three possibilities for each, we are in principle faced with nine possibilities. Actually, only four or so make sense. We will review those which are actually available in existing systems.

9.4 PROGRAMMER-CONTROLLED DEALLOCATION

One popular solution is to provide a reclamation facility at the implementation level, while passing on the detection problem to software developers.

This is certainly the easiest solution for language implementers: all they have to do is to provide a primitive, say *reclaim*, such that *a*. *reclaim* tells the runtime system that the object attached to *a* is no longer needed and the corresponding memory cells may be recycled for new objects.

This is the solution adopted by such non object-oriented languages as Pascal (*dispose* procedure), C (*free*), PL/I (*FREE*), Modula-2 and Ada; you will also find it in most of the "hybrid object-oriented languages", in particular C++ and Objective-C.

This solution is favored by many programmers, especially in the C world, who like to feel in full control of what happens. As a typical reaction here is a Usenet message, posted on the *comp.lang.objective-c* discussion group in response to a suggestion that Objective-C could benefit from automatic reclamation:

I say a big NO! Leaving an unreferenced object around is BAD PROGRAMMING. Object pointers ARE like ordinary pointers — if you [allocate an object] you should be responsible for it, and free it when its finished with (didn't your mother always tell you to put your toys away when you'd finished with them?).

Posting by Ian Stephenson, 11 May 1993.

For serious software development this attitude is not defensible. Grown-up developers must be prepared let someone else play with their "toys" for two reasons: reliability and ease of development.

The reliability issue

Assume developers are in control of deallocating objects with a *reclaim* mechanism. The possibility of an erroneous *reclaim* is always lurking, especially in the presence of complex data structures. In particular, as the software evolves, a *reclaim* that used to be justified may become incorrect.

Such a mistake causes what is known as the *dangling reference* problem: the case in which an object keeps, in one of its fields, a reference to another object which has been reclaimed. If the system then tries to use the reference after that object's memory area has been recycled to hold wholly unrelated information, the result will usually be a run-time crash or (worse yet) erroneous and erratic behavior.

This type of error is known to be the source of some of the most common and nasty bugs in the practice of C and derived languages. Programmers in these languages particularly fear such bugs because of the difficulty of tracing down their source, a difficulty that is easy to understand: if the programmer forgot to note that a certain reference was still attached to an object, and as a result wrongly issued a *reclaim* on the object, it is often because the missed reference came from a completely different part of the software. If so there will be a great conceptual and physical distance between the error (the wrong *reclaim*) and its manifestation (a crash or other abnormal behavior due to an attempt to follow an incorrect reference); the latter may occur long after the former, and in a seemingly unrelated part of the system. In addition the bug may be hard to reproduce if the operating system does not always allocate memory in the same way.

Dismissing the issue, as in the Usenet message reproduced above, by claiming that only "BAD PROGRAMMING" leads to such situations, does nothing to help. To err is human; to err when programming a computer is inevitable. Even in a moderately complex application, no developer can be trusted, or trust himself, to keep track of all run-time objects. This is a task for computers, not people.

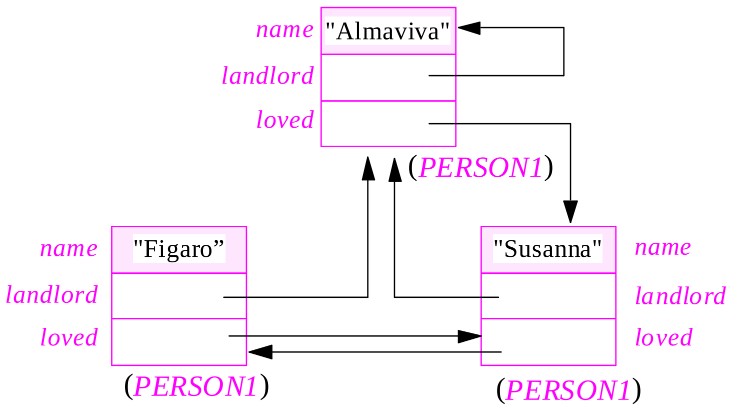
Many a C or C++ programmer has spent many a night trying to figure out what in the world could have happened to one of his "toys". It is not rare to see a project repeatedly delayed by such mysterious memory bugs.

The ease of development issue

Even if we were able to avoid erroneous *reclaim* calls, the question remains of how realistic it would be to ask developers to handle object reclamation. The snag is that, assuming you have positively identified an object that is up for reclamation, just releasing that object is usually not sufficient, as it may itself contain references to other objects.

This figure originally appeared on page 226. dispose, as noted, is the name of the Pascal procedure for what is called reclaim in this discussion.

Take the structure shown by the figure at the top of the next page, the same one used in the previous chapter to describe the dynamic nature of object structures. Assume you have correctly deduced that you may reclaim the top object. Then in the absence of any other references you may also reclaim the other two objects, which it references directly in one case and indirectly in the other. Not only may you reclaim them, you *should* do so: how good would it be to reclaim only part of a structure? In Pascal terminology this is sometimes called the **recursive dispose** problem: if the reclaim operations are to make any sense, they must recursively apply to a whole data structure, not just to an individual object. But of course you need to make sure that no references remain to the other objects from the outside. This is an arduous and error-prone task.



Direct and indirect self-reference

In this figure all the objects are of the same type. Consider now an entity *x* attached to an object O of type *MY_TYPE*, with the class declaration

```
class MY_TYPE feature
        attr1: TYPE_1
        attr2: TYPE_2
end
```

Every object of type *MY_TYPE*, such as O, contains references which (unless void) are attached to objects of types *TYPE_1* and *TYPE_2*. Reclaiming O may imply that these two objects should also be reclaimed, as well as any of their own direct or indirect dependents. Implementing the recursive dispose in this case means writing a set of reclamation procedures, one for each type of objects that may contain references to other objects. The result will be a set of mutually recursive procedures of great complication.

All this leads to disaster. It is indeed not uncommon, in languages that do not support automatic garbage collection, to see a large part of the text of an “application” system, and a large part of the development effort, being devoted to memory management. Such a situation is unacceptable. As an application developer, you should be able to concentrate on your job — solving application problems —, not become a bookkeeper or garbage collector (whichever metaphor is more appropriate).

Needless to say, the increased software complexity resulting from manual memory management results in decreased quality. In particular, it hampers readability and such other properties as ease of error detection and ease of modification. The resulting complexity further compounds the problem highlighted in the previous section — reliability. The more complex a system, the more likely it is to contain errors. The sword of Damocles of a possible erroneous *reclaim* is always hanging over your head, likely to fall at the worst possible time: when the system goes from testing to production and, as a result, starts creating bigger and more intricate object structures.

The conclusion is clear. Except in tightly controlled situations (as discussed in the next section), manual memory management is not appropriate for serious software development — at least if there is any concern for quality.

9.5 THE COMPONENT-LEVEL APPROACH

The next section is "AUTOMATIC MEMORY MANAGEMENT", 9.6, page 301.

(This section describes a solution useful in a specific case only; you may skip it on first reading.)

Before we move on to more ambitious schemes such as automatic garbage collection, it is interesting to look at a solution which may be described as a responsible alternative to the previous one, avoiding some of its drawbacks.

This solution is only applicable within an object-oriented, bottom-up approach to software design, where data structures are not developed "on the spot" as programs need them, but built as reusable classes: general-purpose implementations of abstract data types, with all the associated operations — features.

What sets the object-oriented approach apart with respect to memory management? Part of the novelty, rather than technical, is organizational: with the method's emphasis on reuse of libraries, there now stands between the application developers and the implementers of the base technology (compiler and development tools), a third group of people responsible for writing reusable components that implement the main data structures. Its members — who may of course participate at times in the other two activities — may be called the **component manufacturers**.

The component manufacturers have total control over all uses of a given class, and so are in a better position to find an acceptable solution to the memory management problem for all instances of that class.

If the pattern of allocation and deallocation for the class is simple enough, the component manufacturers may be able to find an efficient solution which does not even require the underlying runtime system to provide a specific *reclaim* routine; they can express everything in terms of higher-level concepts. This may be called the component-level approach.

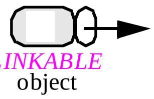
Managing space for a linked list

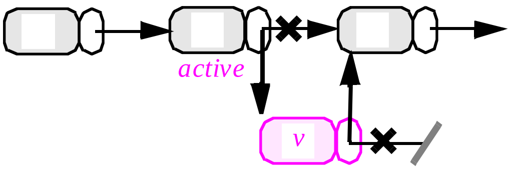
See illustrations of linked lists and LINKABLE objects on the next page.

Here is an example of the component-level approach. Consider a class *LINKED_LIST*, describing lists that consist of a header and any number of linked cells, themselves instances of a class *LINKABLE*. The allocation and deallocation pattern for linked lists is simple. The objects of concern are the "linkable" cells. In this example, the component manufacturers (the people responsible for classes *LINKED_LIST* and *LINKABLE*) know exactly how linkables are created — by the insertion procedures — and how linkables may become dead — as a result of the deletion procedures. So they can manage the corresponding space in a specific way.

Let us assume that *LINKED_LIST* has only two insertion procedures, *put_right* and *put_left*, which insert a new element at the left and right of the current cursor position. Each will need to create exactly one new *LINKABLE* object; they are the basic source of allocation due to *LINKED_LIST*. A typical implementation is:

```
put_right (v: ELEMENT_TYPE) is
        -- Insert an element of value v to the right of cursor position.
    require
        …
    local
        new: LINKABLE
    do
        !! new.make (v)
         active. put_linkable_right (new)
        … Instructions to update other links …
    end
```

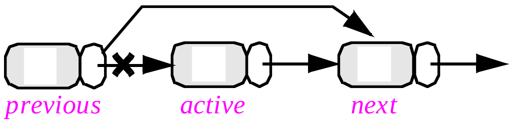




The creation instruction !! *new.make* (*v*) directs the language implementation level to allocate memory for a new object.

In the same way that we control where objects are created, we know exactly where they can become unreachable: through one of the deletion procedures. Let us assume three such procedures *remove*, *remove_right*, *remove_left*; there may also be others such as *remove_all_occurrences* (which removes all occurrences of a certain value) and *wipe_out* (which remove all elements), but we may assume that they internally rely on the first three, each of which makes exactly one *LINKABLE* unreachable. Procedure *remove*, for example, may have the following form:

```
remove is
        -- Delete element at cursor position.
    do
        …
         previous. put_linkable_right (next)
        … Instructions to update other links …
        active := next
    end
```



These deletion procedures provide the exact context for detecting unreachable objects and, if desired, putting them aside for later reuse. In the absence of any automatic scheme for releasing memory, the component manufacturer may safely conserve memory, by avoiding the allocation requested by an insertion when previous deletions have created unreachable *LINKABLE* objects and stored them somewhere for later reuse.

Assume we keep these instances of *LINKABLE* in a data structure called *available*; we will see below how to represent it. Then we may replace the creation instructions such as !! *new.make* (*v*) in *put_right* and *put_left* by

```
new := fresh (v)
```

where *fresh* is a new secret function of *LINKED_LIST*, which will return a ready-for-use linkable. Function *fresh* will attempt to obtain its result from the *available* list, and will only perform a creation if the list is empty.

Elements will be fed into *available* by the deletion procedures. For example, the body of *remove* should now be of the form

```
do
    recycle (active)
            -- The rest as before:
    … Instructions to update links: previous, next, first_element, active …
```

where *recycle*, a new procedure of *LINKED_LIST*, plays the opposite role of *fresh*: adding its argument to the list of available objects. This procedure will be secret (not exported to any client) since it is for internal use only.

Dealing with recycled objects

To implement *fresh* and *recycle*, we may, among other possible choices, represent *available* as a stack: *fresh* will pop from and *recycle* will push onto the stack. Let us introduce a class *STACK_OF_LINKABLES* for the occasion and add the following secret features to *LINKED_LIST*:

Exercise E23.1, page 807 (based on later methodological discussions), asks you to discuss whether it is proper for function *fresh* to produce a side effect.

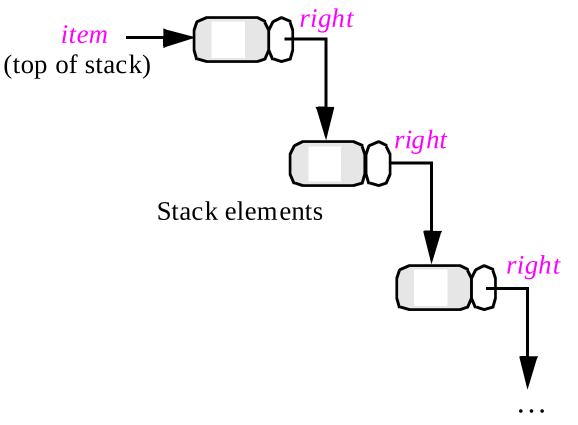
```
available: STACK_OF_LINKABLES

fresh (v: ELEMENT_TYPE): LINKABLE is
            -- A new element with value v, for reuse in an insertion
    do
        if available.empty then
                    -- No choice but to perform an actual allocation
            !! Result.make (v)
        else
                    -- Reuse previously discarded linkable
            Result := available.item; Result.put (v); available.remove
        end
    end

recycle (dead: LINKABLE) is
            -- Return dead to the available list.
    require
        dead /= Void
    do
        available.put (dead)
    end
```

We may declare class *STACK_OF_LINKABLES* as follows:

```
class
    STACK_OF_LINKABLES
feature {LINKED_LIST}
    item: LINKABLE
        -- Element at top
    empty: BOOLEAN is
            -- Is there no item?
        do
            Result := (item = Void)
        end
    put (element: LINKABLE) is
            -- Add element on top.
        require
            element /= Void
        do
            element.put_right (item); item := element
        end
    remove is
            -- Remove last item added.
        require
            not empty
        do
            item := item.right
        end
end
```



The stack representation, as pictured, takes advantage of the *right* field already present in every *LINKABLE* to link all recycled elements without using any extra space. *LINKABLE* must export *right* and *put_right* to *STACK_OF_LINKABLES* as well as *LINKED_LIST*.

Feature *available*, as declared, is an attribute of the class. This means that each linked list will have its own stack of linkables. It is of course a better use of space, if a given system contains several lists, to share the pool of recycled linkables over the whole system. The technique to do this, *once functions*, will be introduced later; making *available* a once function means that only one instance of the class will exist throughout a given system execution, achieving the desired goal.

Exercise E9.3, page 316, and E9.4. On once functions see "CONSTANTS OF CLASS TYPES", 18.3, page 646.

Discussion

This example shows what the component-level approach can do to alleviate the problem of space reclamation by treating it at the component manufacturing level. It assumes that the underlying language implementation does not offer the automatic mechanisms described in the next sections; rather than burdening application programs with memory management problems, with all the risks discussed earlier, the solution presented assigns both detection and reclamation to the basic reusable classes.

The drawbacks and benefits are clear. Problems of manual memory management (reliability risks, tediousness) do not magically vanish; coming up with a foolproof memory management scheme for a particular data structure, as done above for linked lists, is hard. But instead of letting each application developer cope with the issue, we assign this job to component manufacturers; it should fit well in their general task of chiseling out high-quality reusable components. The extra effort is justified by the benefits of having good components available for frequent use by many different applications.

The component-level approach assumes a data structure whose patterns of creation and obsolescence are simple and perfectly understood. This covers only certain cases; for many structures the pattern is unpredictable or too complicated. When the approach is applicable, it provides a better solution, when the underlying language system does not offer automatic memory management, than letting each application developer try to handle the problem manually, or not handle it at all.

9.6 AUTOMATIC MEMORY MANAGEMENT

None of the approaches seen so far is fully satisfactory. A general solution to the problem of memory management for objects involves doing a serious job at the language implementation level.

The need for automatic techniques

A good O-O environment should offer an automatic memory management mechanism which will detect and reclaim unreachable objects, allowing application developers to concentrate on their job — application development.

The preceding discussion should suffice to show how important it is to have such a facility available. In the words of Michael Schweitzer and Lambert Strether:

From [Schweitzer 1991], *page 57.*

An object-oriented program without automatic memory management is roughly the same as a pressure cooker without a safety valve: *sooner or later the thing is sure to blow up*!

Many development environments advertized as O-O still do not support such mechanisms. They may have other features which make them attractive at first; and indeed they may work nicely on small systems. But for serious development you run the risk that they will let you down as soon as the application reaches real size. To summarize in the form of concrete advice:

> In choosing an O-O environment — or just an O-O language compiler — for production development, restrict your attention to solutions that offer automatic memory management.

Two major approaches are applicable to automatic memory management: reference counting and garbage collection. They are both worth examining, although the second one is by far the more powerful and generally applicable.

What exactly is reclamation?

One technical point before we look at reference counting and garbage collection. With any form of automatic storage management, the question arises of what it concretely means for the mechanism to "reclaim" an object which it has detected as being unreachable. Two interpretations are possible:

- The mechanism may add the object's memory to a "free cell list" which it constantly maintains, in line with the techniques used by the earlier component-level scheme. A subsequent creation instruction (!! *x*…) will then look first in this list to find space for the desired new object; only if the list is empty, or contains no appropriate cell, will the instruction require memory from the underlying operating system. This may be called the **internal free list** approach.
- Alternatively, reclaiming an object may mean returning the associated memory to the operating system. In practice, this solution will usually include some aspects of the first: to avoid the overhead of repeated system calls, reclaimed objects will temporarily be kept in a list, whose contents are returned to the operating system whenever their combined size reaches a certain threshold. This may be called the **actual reclamation** approach.

Although both solutions are possible, long-running systems (in particular systems that must run forever) require actual reclamation. The reason is easy to understand: assume an application which never stops creating objects, of which a large proportion will eventually become unreachable, so that there is an upper bound on the total number of objects reachable at any one time, even though the total number of created objects since the beginning of a session is unbounded. Then with the internal free list approach it is possible to have a situation where the application will forever keep asking for more memory even though its actual memory needs are not growing. An exercise at the end of this chapter asks you to construct a pattern that will exhibit this behavior.

Exercise E9.1, page 316.

It would be frustrating to have automatic memory management and still find ourselves in the London Ambulance Service situation — encroaching byte by byte on the available memory for no good reason, until execution runs out of space and ends in disaster.

9.7 REFERENCE COUNTING

The idea behind the first automatic memory management technique, reference counting, is simple. In every object, we keep a count of the number of references to the object; when this count becomes null, the object may be recycled.

This solution is not hard to implement (at the language implementation level). We must update the reference count of any object in response to all operations that can create the object, attach a new reference to it and detach a reference from it.

Any operation that creates an object must initialize its reference count to one. This is the case in particular with the creation instruction !! *a*, which creates an object and attaches it to *a*. (The case of *clone* will be studied shortly.)

Any operation that attaches a new reference to an object O must increase O’s reference count by one. Such attachment operations are of two kinds (where the value of *a* is a reference attached to O):

A1 • *b* := *a* (assignment).

A2 • *x*.*r* (…, *a*, …), where *r* is some routine (argument passing).

Any operation which detaches a reference from O must decrease its reference count by one. Such detachment operations are of two kinds:

D1 • Any assignment *a* := *b*. Note that this is also an attachment operation (A1) for the object attached to *b*. (So if *b* was also attached to O we will both increment and decrement O’s count, leaving it unchanged — the desired outcome.)

D2 • Termination of a routine call of the form *x*.*r* (…, *a*, …). (If *a* occurs more than once in the list of actual arguments we must count one detachment per occurrence.)

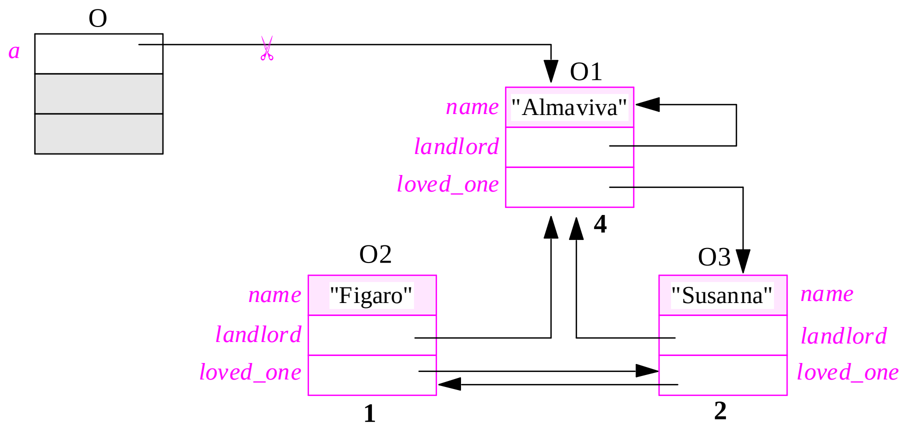
After such an operation, the implementation must also check whether O’s reference count has reached value zero; if so, it may reclaim the object.

Finally the case of *clone* must be handled carefully. The operation *a* := *clone* (*b*), which duplicates the object OB attached to *b*, if any, and attaches the resulting new object OA to *a*, must not duplicate the reference count. Instead, it must initialize the reference count of OA to one; in addition, if OB had any non-void reference fields, it must increase by one, for every such field, the reference count of the attached object. (If two or more fields are attached to a single object, its reference count will be increased as many times.)

One obvious drawback of reference counting is the performance overhead in both time and space. For every operation on references the implementation will now execute an arithmetic operation — and, in the detachment case, a conditional instruction. In addition, every object must be extended with an extra field to hold the count.

But there is an even more serious problem which makes reference counting, unfortunately, of little practical use. (“Unfortunately” because this technique is not too hard to implement.) The problem is cyclic structures. Consider once again our staple example of a structure with mutually referring objects:

Uncollectible cyclic structure



The objects in the right part of the figure, O1, O2 and O3, contain cyclic references to each other; assume that no outside object other than O contains a reference to any of these objects. The corresponding reference counts have been displayed below each object.

Now assume that (as suggested by the ✂) the reference from O to O1 is detached, for example because a routine call with target O executes the instruction

a := *Void*

Then the three objects on the right have become unreachable. But the reference counting mechanism will never detect this situation: the above instruction decreases O1's count to three; after that the reference counts of the three colored objects will stay positive forever, preventing them from being reclaimed.

Because of this problem, reference counting is only applicable to structures which are guaranteed never to include any cycle. This makes it unsuitable as a general-purpose mechanism at the language implementation level, since it is impossible to guarantee that arbitrary systems will not create cyclic structures. So the only application that would seem to remain is as a technique to be used by library developers at the component manufacturing level. Unfortunately if the component-level techniques of the previous section are not applicable it is usually because the structures at hand are too complex, and in particular because they contain cycles.

9.8 GARBAGE COLLECTION

The most general technique, and in fact the only fully satisfactory one, is automatic garbage collection, or just garbage collection for short.

The garbage collection mechanism

A garbage collector is a facility included in the runtime system for a programming language. (The runtime system, or just runtime for short, is a component of the programming language's implementation; it complements the compiler by providing the mechanisms needed at execution time to support the execution of software systems written in the language.) The garbage collector will take care of both detecting and reclaiming unreachable objects, without the need for explicit handling by application software — although application software may have various facilities at its disposal to control the collector's operation.

See the bibliographical notes on page 315 for references on garbage collection.

A detailed exploration of garbage collection techniques would justify a book of its own (which remains to be written). Let us take a look at the general principles of garbage collectors and the problems that they raise, focusing on the properties that are directly relevant to application developers.

Garbage collector requirements

A garbage collector should, of course, be correct. This is actually two requirements:

Garbage collector properties

Soundness: every collected object is unreachable.

Completeness: every unreachable object will be collected.

It is just as easy to write a sound collector (never collect any object) as a complete one (collect all objects); the difficulty is of course to achieve both properties in a single product.

Soundness is an absolute requirement: better no garbage collector than one which, once in a while, steals an active object from your application! You must be able to trust memory management blindly; in fact, you should be able to forget about it most of the time, being content enough to know that someone, somehow, collects the mess in your software the same way someone, somehow, collects the garbage in your office while you are gone — but does not take away your books, your computer, or the family pictures on your desk.

Completeness is desirable too, since without it you may still face the problem that the garbage collector was supposed to solve: memory wasted on useless objects. But here we may be able to accept less than perfection: a quasi-complete collector could still be useful if it collects the bulk of the garbage while occasionally missing an object or two.

Let us refine and improve this observation. In reality you *will* want any industrial-grade collector to be complete, lest you get back to the uncertainties of environments with no memory management. Completeness is in practice just as necessary as soundness, but less pressing if we rephrase the definition as: “every unreachable object will *ultimately* be collected”. Suppose that we can make the collection process more efficient overall through an algorithm that eventually collects every unreachable object but may lag in getting to some of them: such a scheme would be acceptable for most applications. This is the idea of “generation scavenging” algorithms discussed below, which for efficiency’s sake spend most of their efforts scanning the memory areas most likely to contain unreachable objects, and take care of the remaining areas at less frequent intervals.

If we start considering such tradeoffs it will be necessary to characterize a garbage collector, beyond the yes-no criteria of soundness and completeness, by a more quantitative property which we may call **timeliness**: the time it takes — both the average value and the upper bound will be interesting — between the moment an object becomes unreachable and the moment the collector, assumed to be both sound and complete, reclaims it.

The definition of soundness illuminates the difficulties associated with garbage collection for some languages, and the respective roles of a language and its implementation. Why, for example, is garbage collection usually not available for C++? The reasons most commonly cited are cultural: in the C world each developer is supposed to take care of his toys (in Stephenson’s words); he simply does not trust any automatic mechanism to manage his own business. But if this were the true reason, rather than a posteriori justification, C++ environments could at least offer garbage collection as an option, and most do not.

The real issue is language design, not compiling technology or even cultural attitudes. C++, in the C tradition, is rather weakly typed, providing *casts* — type conversions — through which you can refer to an object of a certain type through an entity of another type, with few restrictions on possible type combinations. The syntax

(OTHER_TYPE) x

denotes *x* viewed as an entity of type *OTHER_TYPE*, related or not to the true type of *x*. Good C++ books censure the wilder applications of this common practice, but methodological aspersions are of no use to the compiler writer, who must deal with the language as defined. Now imagine the following scenario: a reference to an object of some useful type, say *NUCLEAR_SUBMARINE*, is temporarily cast into an integer; the garbage collector jumps in and examines the value, seeing nothing but the most innocent-looking of integers; finding no other reference to the object, it reclaims it; but now the program casts the integer back to its true vocation of nuclear submarine reference; and it tries to access the now inexistent object, with consequences of great sorrow to all affected.

Various techniques have been proposed around this problem. Because they usually involve some restrictions on the use of the language, they have not found their ways into common commercial offerings. The Java language may be viewed as a form of C++ which has dramatically restricted the type system — going so far as to remove genericity and multiple inheritance — to make garbage collection possible at last in a C-based world.

With a carefully designed type system, it is of course possible to provide the whole power of multiple inheritance and genericity while ensuring type safety and language support for efficient garbage collection.

Garbage collection basis

Let us come now to how a garbage collector works.

The basic algorithm usually includes two phases, at least conceptually: mark and sweep. The mark phase, starting from the origins, follows references recursively to traverse the active part of the structure, marking as reachable all the objects it encounters. The sweep phase traverses the whole memory structure, reclaiming unmarked elements and unmarking everything.

The origins are the objects directly attached to entities of the software text. See "Reachable objects in the object-oriented model", page 288.

As with reference counting, objects must include an extra field, used here for the marking; but the space overhead is negligible, since one bit suffices per object. As will be seen when we study dynamic binding, implementation of O-O facilities requires that every object carry some extra internal information (such as its type) in addition to its official fields corresponding to the attributes of the generating class. This information typically occupies one or two words per object; the marking bit can usually be squeezed into one of these extra words, so that in practice there is no observable overhead.

All-or-nothing collection

When should the garbage collector be triggered?

Classical garbage collectors are activated on demand and run to completion. In other words the garbage collector is inactive as long as there is some memory left to the application; when the application runs out of memory, it triggers an entire garbage collection cycle — mark phase followed by sweep phase.

This technique may be called the all-or-nothing approach. Its advantage is that it causes no overhead as long as there is enough memory; the program is only penalized by memory management when it has exceeded available resources.

But all-or-nothing garbage collection has a serious potential drawback: a complete mark-sweep cycle may take a long time — especially in a virtual memory environment in which filling the memory means filling a very large virtual address space, which the garbage collector will then have to traverse entirely, all that time preventing the application from proceeding.

This scheme may be acceptable for batch applications, although with a high ratio of virtual to real memory thrashing may cause serious performance degradation if a system creates many objects and a large proportion of them become unreachable. All-or-nothing garbage collection will not work, however, for interactive or real-time systems. Imagine a missile interception system which has a 50-millisecond window to react when an enemy missile is fired. Assume everything works fine until the software runs out of memory, at which stage it defers to the garbage collector; but — bad luck — this is precisely when the missile comes in! Even in less life-threatening applications, such as a interactive systems, it is not pleasant to use a tool (for example a text editor) which, once in a while, gets unpredictably hung for ten minutes or so because the underlying implementation has entered a garbage collection cycle.

In such cases the problem is not necessarily the global effect of garbage collection on efficiency: a certain overall performance penalty may be perfectly tolerable to users and developers as the price to pay for the gain in reliability and convenience afforded by automatic garbage collection. But such a penalty should be evenly spread; what will usually not be acceptable is the unpredictable bursts of memory management activity caused by the all-or-nothing approach. Better a tortoise than a hare which, once in a while and without warning, takes a half-hour nap. Reference counting, were it not for its fatal flaw, would satisfy this observation that uniformly slow is often preferable to usually fast but occasionally unpredictable.

Of course the penalty, besides being uniform, must also be small. If the application without a garbage collector is a indeed a hare, no one will really settle for a tortoise; what we can accept is a somewhat less agile hare. A good garbage collector will have an overhead of 5% to 15%. Although some managers, developers and users will scream that this is unacceptable, I know very few applications that cannot tolerate this kind of cost, especially in light of the obvious observation that in the absence of garbage collection the software will have to perform manual reclamation, which does not come for free either (even if we concentrate on execution overhead only and disregard the overhead on development time and the reliability problems). Unfortunately most of the few benchmarks that exist in this area end up, in their effort to measure the measurable, comparing the incomparable: a system executed with no garbage collection and no manual reclamation, versus one running with garbage collection. Even under this unfavorable light, however, a performance cost in the quoted range makes garbage collection shine.

This discussion has identified the two complementary efficiency issues for garbage collectors: *overall performance* and *incrementality*.

Advanced approaches to garbage collection

A good collector should provide good performance both overall and incrementally, making itself suitable for interactive or even real-time applications.

A first requirement is to give developers some control over the activation and deactivation of collector cycles. In particular, the environment's libraries should offer procedures

collection_off

collection_on

collect_now

such that a call to the first specifies that no collector cycle should start until further notice; a call to the second resumes normal operation; and a call to the third immediately triggers a complete cycle. Then if a system contains a time-critical section, which must not be subject to any unpredictable delay, the developer will put a call to *collection_off* at the beginning of the section and a call to *collection_on* at the end; and at any point where the application is known to be idle (for example during certain input or output operations) the developer may, if useful, include a call to *collect_now*.

A more advanced technique, used in some form by most modern garbage collectors, is known as **generation scavenging**. It follows from the experimental observation that "old objects will stay around": the more garbage collection cycles an object has survived, the better chance it has of surviving many more cycles or even remaining forever reachable. This property is precious since the sweep part of garbage collection tends to consume a considerable amount of time, so that the collector will greatly benefit from any information allowing it to examine certain categories less frequently than others.

Generation scavenging will detect objects that have existed for more than a certain number of cycles. This is called **tenuring** (by analogy with the mechanisms that protect instances of the real-life class *PROFESSOR* once they have survived a few cycles of university politics). Tenured objects will be set aside and handled by a separate collection process, which will run less frequently than the collector of "young" objects.

Generation scavenging helps incrementality, but does not fully achieve it, since there remains a need to perform full collections occasionally.

Practical implementations of generation scavenging use many variations on this basic idea. In particular, it is common to divide objects not just into young and old, but into several generations, with different policies for collecting the various generations. These ideas have a considerable effect on the overall performance of garbage collection.

Parallel garbage collection algorithms

To obtain a full solution to the incrementality problem, an attractive idea (if the underlying operating system supports multiprocessing) is to assign garbage collection to a separate thread of control. This is known as **on-the-fly**, or **parallel**, garbage collection.

With on-the-fly garbage collection, execution of an O-O system involves two separate threads (often corresponding to two separate processes of the operating system): the application and the collector. Only the application can allocate memory, through creation instructions; only the collector can free memory, through *reclaim* operations.

The collector will run continuously, repeatedly executing a mark phase followed by a sweep phase to detect and pick up the application's unreachable objects. Think of an endless New York ticker-tape parade, forever marching through the streets of the city. The application is the parade, generously scattering, wherever it passes, objects of all kinds; the garbage collector is the cleaning squad which follows at a short distance, gathering all that has been left.

The notion of coroutine will be introduced in the concurrency chapter. See "Coroutines", page 1012.

The separate threads of control need not be physically distinct processes. With modern operating systems they can be threads; or, to avoid the overhead of switching between processes or even threads, they may be plain coroutines. Even so, however, on-the-fly garbage collection tends in practice to have unsatisfactory overall performance. This is regrettable since the method's incrementality is indeed (with Dijkstra's algorithm, see the reference in the bibliographic notes) quite good.

In my opinion (the proper word here, since this comment reflects hope, not a scientifically established result) parallel garbage collection remains the solution of the future, but will require cooperation from the hardware. Rather than stealing time from the processor which handles the application, garbage collection should be handled by a separate processor, entirely devoted to that task and designed so as to interfere as little as possible with the processor or processors devoted to the application.

This idea requires changes to the dominant hardware architectures and so is not likely to be widely implemented soon. But in an answer to the sometimes asked question

"*What kind of hardware support would be most useful for object technology*?"

the presence of a separate garbage collection processor should, I believe, be the first item on the wish list.

9.9 PRACTICAL ISSUES OF GARBAGE COLLECTION

An environment providing automatic memory management through garbage collection must not only use excellent garbage collection algorithms but also provide a few facilities which, although not central to a theory of memory management, are essential for the practical use of the environment.

Class *MEMORY*

See "ADVANCED EXCEPTION HANDLING", 12.6, page 431 about EXCEPTIONS and "REQUESTING SPECIAL SERVICE", 30.8, page 998 about CONCURRENCY.

Several of the required facilities can be provided in the form of features callable by application software. As always in such cases (facilities to be used by developers who need to tune or adapt a basic mechanism of the method and language) the most convenient approach is to group these features in a class, which we will call *MEMORY*. Then any class that needs these facilities will inherit from *MEMORY*.

A similar approach will be used for adapting the exception handling mechanism (class *EXCEPTIONS*) and the concurrency mechanism (class *CONCURRENCY*).

Among the features of class *MEMORY* will be the procedures discussed earlier for stopping the collection mechanism, resuming it, and triggering a full collection: *collection_off*, *collection_on*, *collect_now*.

A disposal mechanism

Another important procedure of *MEMORY* is *dispose* (not to be confused with its Pascal namesake, which frees memory). It addresses an important practical problem sometimes called *finalization*. If the garbage collector reclaims an object that is associated with some external resources, you may wish to specify a certain action — such as freeing the resources — to be executed at reclamation time. A typical example is a class *FILE*, whose instances will represent files of the operating system. It is desirable to have a way of specifying that whenever the garbage collector reclaims an instance of *FILE* that has become unreachable it will call a certain procedure to close the associated physical file.

More generally let us assume a procedure *dispose* which executes the operations needed at the time an instance of the class is reclaimed. With a manual approach to memory management no particular problem would arise: it would suffice to include a call to *dispose* just before every call to *reclaim*. The "destructors" of C++ take care of both operations: *dispose* and *reclaim*. With a garbage collector, however, the software does not directly control (for all the good reasons that we have explored) the moment at which an object is reclaimed; so it is impossible to include explicit calls to *dispose* at the right places.

The answer relies on the power of object technology and in particular on inheritance and redefinition. (These techniques are studied in later chapters but their application here is simple enough to be understandable without a detailed grasp of their principles.) Class *MEMORY* has a procedure *dispose*, whose body performs no action at all:

```
dispose is
        -- Action to be taken in case of reclamation by garbage collector;
        -- nothing by default.
        -- Called automatically by garbage collector.
    do
    end
```

Then any class which requires special dispose actions whenever the collector reclaims one of its instances will redefine procedure *dispose* to perform these actions. For example, assuming that class *FILE* has a boolean attribute *opened* and a procedure *close*, both with the obvious semantics, it will redefine *dispose* appropriately:

```
dispose is
        -- Action to be taken in case of reclamation by garbage collector:
        -- close the associated file if open.
        -- Called automatically by garbage collector.
    do
        if opened then
            close
        end
    end
```

As the comments indicate, the rule is that any object reclamation will cause a call to *dispose* — either the original empty procedure for the (by far commonest) case in which no redefinition has occurred in the generating class, or the redefined version.

Garbage collection and external calls

See "INTERFACING WITH NON-O-O SOFTWARE", 13.1, page 439.

A well-engineered object-oriented environment with garbage collection must address another practical problem. O-O software will in many cases have to interact with software written in other languages. In a later chapter we will see how best to ensure this interaction with the non-O-O world.

If your software indeed uses calls to routines written in other languages (called *external routines* in the rest of this discussion), you may have to pass to these routines references to objects. This causes a potential danger with respect to memory management. Assume that an external routine is of the following form (transposed to the appropriate foreign language):

```
r (x: SOME_TYPE) is
    do
        …
        a := x
        …
    end
```

where *a* is an entity which may retain its value between successive activations of *r*; for example *a* could be a global or "static" variable in traditional languages, or a class attribute in our O-O notation. Consider a call *r* (*y*), where *y* is attached to some object O1. Then it is possible that some time after the call O1 becomes unreachable from the object-oriented side while there is still a reference to it (from *a*) in the external software. The garbage collector could — and eventually should — reclaim O1, but this is wrong.

For such cases we must provide procedures, callable from the external software, which will protect a retained object from the collector, and terminate such protection. These procedures may be called under the form

```
adopt (a)
wean (a)
```

and should be part of any interface library supporting the communication between object-oriented and external software. The C interface library of the mechanism described in the next section supports such a facility. "Adopting" an object takes it off the reach of the reclamation mechanism; "weaning" it makes it reclaimable again.

Passing objects to non-object-oriented languages and retaining references to them from the foreign side of the border is of course risky business. But it is not always possible to avoid it. For example an object-oriented project may need a special interface between the O-O language and an existing database management system; in such cases you may need to let the other side retain information about your objects. Such low-level manipulations should never appear in normal application software, but should be encapsulated in utility classes, written with particular care so as to hide the details from the rest of the software and protect it against possible trouble.

9.10 AN ENVIRONMENT WITH MEMORY MANAGEMENT

As a conclusion let us take a peek at how one particular environment — the one presented more broadly in the last chapter of this book — handles memory management. This will give an example of practical, state-of-the-art approaches to the problem.

Basics

Memory management is automatic. The environment includes a garbage collector, which is always on by default. It is sufficiently unobtrusive to have caused users to call and ask "*what should I do to turn on the garbage collector*?", only to be told that it is already on! In normal usage, including interactive applications, you will not notice it. You can turn it off through *collection_off* as discussed earlier.

Unlike the collectors found in many other environments, the garbage collector does not just free memory for reuse by further object allocations in the same system execution, but actually returns it to the operating system for use by other applications (at least on operating systems that do provide a mechanism to free memory for good). We have seen how essential that property was, especially for systems that must run permanently or for a long time.

Additional engineering goals presided over the garbage collector design: efficient memory collection; small memory overhead; incremental behavior (avoiding blocking the application for any significant period of time).

Challenges

The garbage collector must face the following issues, following from the practical constraints on object allocation in a modern, O-O environment:

- O-O routines can call external functions, in particular C functions, which have their own needs for memory allocation. We must therefore consider that there are two distinct kinds of memory: object memory and external memory.

- All objects are not created equal. Arrays and strings have a variable size; instances of other classes have a fixed size.

- Finally, as noted, it is not enough to free memory for reuse by the O-O application: we must also be able to give it back for good to the operating system.

For these reasons, memory allocation cannot rely on the standard *malloc* system call which, among other limitations, does not return memory to the operating system. Instead, the environment asks the operating system's kernel for memory chunks and allocates objects in these chunks using its own mechanisms.

Object movement

The need to return memory to the operating system is the source of one of the most delicate parts of the mechanism: garbage collection can move objects around.

This property has by far caused the most headaches in the implementation of the collector. But it has also made the mechanism robust and practical; without it there would be no way to use garbage collection for long-running, mission-critical systems.

If you stay within the O-O world you need not think about object movement, except as a guarantee that your system will not expand forever, even if it keeps creating new objects (provided the total size of reachable objects is bounded). But you will need to consider this property if you also use external routines, written for example in C, and pass objects to them. If the C side stores somewhere, in the form of a plain address (a C pointer), a reference to an object from the O-O world, you may be in trouble if it tries to use it without protection ten minutes later: by then the object may have moved elsewhere, and the address may contain something completely different, or nothing at all. A simple library mechanism solves the issue: the C function should "access" the object and access it through the appropriate macro, which will find the object wherever it is.

Garbage collection mechanism

Here is an outline of the algorithm used by the garbage collector.

Rather than a single algorithm the solution actually relies on a combination of basic algorithms, used together (for some of them) or independently. Each activation of the collector selects an algorithm or algorithm combination based on such criteria as the urgency of the memory need. The basic algorithms include generation scavenging, mark-and-sweep and memory compaction, plus a few others less relevant to this discussion.

The idea behind **generation scavenging** was described earlier in this chapter: concentrate on young objects, since they have the greatest likelihood of yielding collectable garbage. A main advantage of this algorithm is that it need not explore all the objects, but only those which can be reached from local entities, and from old objects containing references to young objects. Each time the algorithm processes a generation, all the surviving objects become older; when they reach a given age, they are tenured to the next generation. The algorithm looks for the right tradeoff between low tenure age (too many old objects) and high tenure age (too frequent scavengings).

The algorithm still needs, once in a while, to perform a full **mark-and-sweep** to find any unreachable objects that generation scavenging may have missed. There are two steps: *mark* recursively explores and marks the reachable objects; *sweep* traverses applicable memory and collects the marked objects.

Memory compaction compacts memory, returning unused parts to the operating system, at the lowest possible cost. The algorithm divides the memory into n blocks and takes $n–1$ cycles to compact them all.

Bulimia and anorexia

Since operating system calls (allocate memory, return memory) are expensive, the memory compaction algorithm is conservative: rather than returning all the blocks that have been freed, it will keep a few of them around to build a small reserve of available memory. This way if the application starts shortly afterwards to allocate objects again the memory will be readily available, without any need to call the operating system.

Without this technique, the fairly frequent case of a bulimic-anorexic application — an application that regularly goes into a mad allocation binge, followed by a purge period during which it gets rid of many objects — would cause the memory management mechanism constantly to get memory from the operating system, return it, then ask again.

Garbage collector operation

The garbage collector gets into action when one of the two operations that request memory, a creation instruction (!! *x*…) or a clone, triggers it. The trigger criterion is not just that the application has run out of memory: preferring prevention to cure, the mechanism may activate itself when it detects various conditions in advance of actual memory exhaustion.

If the primary allocation area is full, the collector will execute a scavenging cycle. In most cases this will free enough memory for the current needs. If not, the next step is to go through a full mark-and-sweep collection cycle, generally followed by memory compaction. Only if all this fails to provide the required space will the application, as a last resort, ask the operating system for more memory, if it is still not possible to allocate a new object.

The main algorithms are incremental, and their time consumption is a few percent of the application's execution time. Internal statistics keep track of the memory allocated and help determine the proper algorithm to call.

You can tune the collector's behavior by setting various parameters; in particular, selecting the *speed* option will cause the algorithms not to try to collect all available memory (through the compaction mechanism described above) but instead to call the operating system's allocation facilities earlier. This optimizes speed over compactness. The various parameter-setting mechanisms are obtained, like *collection_off*, *collect_now* and *dispose*, from class *MEMORY*.

The memory management mechanism resulting from the combination of all these techniques has made it possible to develop and run successfully the kind of large, ambitious applications which need to create many objects, create them fast, and (while remaining careful about overall usage of space) let someone else worry about the mundane consequences.

9.11 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- There are three basic modes of object creation: static, stack-based and free. The last is characteristic of object-oriented languages but also occurs elsewhere, for example in Lisp, Pascal (pointers and *new*), C (*malloc*), Ada (access types).
- In programs that create many objects, objects may become unreachable; their memory space is lost, leading to memory waste and, in extreme cases, failure from lack of space even though some space is not used.
- The issue may be safely ignored in the case of programs that create few unreachable objects, or few objects altogether as compared to the available memory size.
- In all other cases (highly dynamic data structures, limited memory resources), any solution will involve two components: *detection* of dead objects, and *reclamation* of the associated space.
- Either task may be handled by the language implementation, the component manufacturing level or application programs.
- Leaving application programs in charge of detection is cumbersome and dangerous. So is a memory reclamation operation in a high-level language.
- In some contexts, it is possible to provide simple memory management at the component level. Detection is handled by the components; reclamation, by either the components or the language implementation.
- Reference counting is inefficient, and does not work for cyclic structures.
- Garbage collection is the most general technique. It is possible to keep its potential overhead on normal system execution acceptably low and, through sufficiently incremental algorithms, not visible in normal interactive applications.
- *Generation scavenging* improves the efficiency of garbage collection algorithms by using the observation that many objects die (become unreachable) young.
- A good memory management mechanism should return unused space not just to the current application but to the operating system.
- A practical memory management scheme was described, offering a combination of algorithms and ways for application developers to tune the mechanism, including turning garbage collection off and on in sensitive sections.

9.12 BIBLIOGRAPHICAL NOTES

A broader perspective on the different models of object creation, discussed at the beginning of this chapter, is provided by the "**contour model**" of programming language execution, which may be found in [Johnston 1971].

The information about the London Ambulance Service fiasco comes from an extensive set of messages posted on the Risks forum (*comp.risks* Usenet newsgroup) moderated by Peter G. Neumann, in April and October of 1992. I relied particularly on several messages by Brian Randell — quoting journal articles (*The Independent*, 29 and 30 October 1992) and BBC bulletins — as well as Trevor Jenkins, Jean Ramaekers, John

Jones, Tony Lezard, and Paul Johnson (to whom I am grateful for bringing this example to my attention). The primary *comp.risks* issue on this topic is 14.48; see also 13.38, 13.42, 13.43, 14.02. The newsgroup archives are accessible through the World-Wide Web at *http://catless.ncl.ac.uk/Risks*.

A parallel garbage collection algorithm was introduced in [Dijkstra 1978]. [Cohen 1984] discusses the performance issues of such algorithms. Generation scavenging was introduced in [Ungar 1984].

The garbage collection mechanism of ISE's environment sketched at the end of this chapter was built by Raphaël Manfredi and refined by Fabrice Franceschi (whose technical report served as the basis for the presentation here) and Xavier Le Vourch.

EXERCISES

E9.1 Patterns of object creation

"What exactly is reclamation?", page 302.

In the discussion of automatic memory management it was pointed out that the "internal free list" approach (in which the space of reclaimed objects is not physically returned to the operating system, but kept in a list for use by future creation instructions) may cause the memory allocated to an application to grow forever even though the actual memory requirement is bounded, whereas the "actual reclamation" approach (in which a reclaim operation actually returns memory) would result in bounded memory usage. Devise a pattern of object creation and reclamation which exhibits this problem.

You may describe such a pattern as a sequence $o_1\ o_2\ o_3$ … where each o_i is either 1, indicating the allocation of one memory unit, or $-n$ (for some integer n), indicating the reclamation of n memory units.

E9.2 What level of reclamation?

"THE COMPONENT-LEVEL APPROACH", 9.5, page 297.

The component level policy, if implemented in a language like Pascal or C where an operating system *dispose* or *free* facility is available, could use this facility directly rather than managing its own free list for every type of data structure. Discuss the pros and cons of both approaches.

E9.3 Sharing the stack of available elements

(This exercise assumes familiarity with the results of chapter 18.) Rewrite the feature *available*, giving the stack of available elements in the component-level approach, so that the stack will be shared by all linked lists of a certain type. (**Hint**: use a once function.)

E9.4 Sharing more

(This exercise assumes that you have solved the previous one, and that you have read up to chapter 18.) Is it possible to make the *available* stack shared by linked lists of all types?