Programming Effectively With Garbage Collection

A Handbook for Dynamic Object-Oriented Programming

by P. Tucker Withington and Andrew Shires

Sample.

harlequin

Europe:

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
U.K.

telephone +44 1223 873 800
fax +44 1223 873 873

North America:

Harlequin Incorporated
One Cambridge Center
Cambridge, MA 02142
U.S.A.

telephone +1 617 374 2400
fax +1 617 252 6505

Electronic Access:

http://www.harlequin.co.uk/
http://www.harlequin.com/

Contents

About This Sample

1 Introduction

This is a review sample for book about programming efficiently in a garbage-collected environment. By “garbage-collected environment”, we mean languages that provide garbage collection as standard, such as Java, Dylan, and Common Lisp, or languages to which you can add a garbage collector, such as C and C++. The book’s working title is *Programming Effectively with Garbage Collection*. For more information, see the draft preface on page 7 of this sample.

2 Contents of this sample

This sample contains:

- A list of the book’s proposed contents.
- A draft preface.
- Drafts of the first two book chapters.
- Drafts of two detailed programming discussions that will appear in the book. They are formatted in the style of cookbook recipes.

These drafts do not contain diagrams.

Draft Preface

1 Memory management and garbage collection

Do your programs grow bigger and bigger as they run, hogging your system and slowing it down? Do they sometimes halt because the system cannot give them any more memory? Does your code work fine during development and testing, only for memory problems to emerge when it is integrated into a larger or longer-running program?

Do your programs crash or contain bugs that appear to be related to memory problems? Do they access memory incorrectly, or somehow corrupt the data stored in it? Are you tired of having to write obscure code in order to keep memory problems in your programs under control? Is your code harder to maintain because of the attention it has to pay to memory bookkeeping?

Better memory management is the key to solving these problems. *Memory management* is the task of making programs use the memory available to them as efficiently as possible. In this book, we show how *garbage collection*, the automatic approach to memory management that is part of many modern programming languages and an option for many others, can eliminate memory errors and reduce performance problems, with smaller programming overheads and fewer maintenance requirements than manual memory management.

2 About this book

This book is a practical guide for programming in languages with garbage collection built in, such as Java and Dylan, and languages that can "plug in" a garbage collector as a library, such as C and C++. To help you adapt your programming style to get the best from garbage-collected languages, this book explains how garbage collectors work, and how programs and garbage collectors influence each other. This book offers a wealth of programming advice that will help you identify and fix code that is causing unnecessary memory consumption in your applications.

3 Audience

This book is relevant to a broad range of developers. If you are interested in languages that feature garbage collection, such as Java and Dylan, or in languages for which a plug-in garbage collector is an option, such as C++ and C, you should read this book because it shows how to improve the performance of your programs in those languages. Generally, if you are interested in object-oriented programming in a garbage collected environment, you will find this book useful.

4 What you need to know to understand this book

To understand this book, you should be a programmer familiar with a language like Java, C++, C, Dylan, Common Lisp, Smalltalk, or Standard ML. We do not assume that you can read all these languages, but we do assume you are familiar with programming in general.

Most of the program examples in this book are written in Java and C++. You do not need to know Java or C++ to find the examples relevant, because they always illustrate general principles.

You do not need to know anything about garbage collection or memory management in general to understand this book. The first part of this book explains them both, showing how garbage collection approaches the general problem of managing program memory, and how it compares with manual memory management techniques.

5 Conventions used in this book

Yet to be written.

6 Where to go for more information

This book does not explain garbage collection theory or algorithms. Nor does it explain how to implement a garbage collector. If you want to know more about those topics, the following sources are a good introduction:

> *Garbage Collection: Algorithms for Dynamic Memory Management*; Richard Jones, Rafael Lins; 1996, Wiley. ISBN 0-471-94148-4.
>
> *Uniprocessor Garbage Collection Techniques*; Paul R. Wilson; 1992, University of Texas at Austin.

Harlequin's Memory Management Reference on the World Wide Web has an FAQ, bibliography, glossary and some introductory essays on memory management:

> **`<URL:http://www.harlequin.com/mm/reference/>`**

If you want to know more about dynamic object-oriented programming, the following books are useful:

> *The Java Programming Language*; Ken Arnold and James Gosling; 1997, Addison-Wesley. ISBN 0-201-63451-1.
>
> *Dylan Programming: An Object-Oriented and Dynamic Language*; Neal Feinberg, Sonya Keene, Robert Mathews, P. Tucker Withington; 1996, Addison-Wesley. ISBN 0-201-47976-1.
>
> *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*; Sonya Keene; 1987, Addison-Wesley. ISBN 0-201-17589-4.

Proposed Contents

1 Proposed contents

Table of contents

1

Garbage Collection and Memory Management

1.1 Introduction

This chapter is an introduction to memory management and garbage collection, and to how garbage collection addresses the problems that memory management must solve. If you are familiar with the basics of memory management and garbage collection, you can skip this chapter.

1.2 What is memory management?

Memory management is the task of making programs use the memory available to them as efficiently as possible. Broadly, this means ensuring that programs use no more memory than necessary to represent the data they need, while keeping the time spent managing memory within limits acceptable to the program's user. Without memory management, most programs grow in size as they run and suffer performance problems; they may even run out of memory and halt.

Memory management is more important today than ever before. Today's applications, in their efforts to solve ever more demanding problems, are using increasing amounts of memory in increasingly complex ways. Although fast main memories like RAM are getting bigger and cheaper, memory remains a relatively scarce resource in computing.

Modern software methods and operating systems only increase the need for effective memory management. Object-oriented programming, for instance, encourages us to work by building and manipulating rich data models of our problem domain. A typical program uses large amounts of highly interconnected data, which makes effective memory management harder.

Multi-tasking operating systems make economical use of memory even more important, because they force a program to compete with other programs for shared memory resources. A program that wastes memory in such an environment makes difficulties not only for itself but also for the programs running alongside it.

1.3 Program data and computer memory

In this section we examine the relationship between program data and computer memory. High-level programming languages allow us to think about data in very abstract terms. Data types and classes, for instance, offer ways of talking about data at a much higher level of description than that used to store it upon physical memory devices.

High-level language abstractions are convenient and powerful, but because they obscure the relationship between program data and memory, it can sometimes be difficult to appreciate that the simple decisions we make about data can have a profound effect on memory usage and, ultimately, program performance.

In this section, we look more deeply into why memory management is necessary by briefly examining *memory allocation* and *memory recycling*.

1.3.1 Memory allocation

Memory allocation or *allocation* is a general term for the process of assigning memory to program data. The way allocation works depends on the *size* and *lifetime* of the data.

By *size*, we mean the amount of memory required to represent the data. This may or may not be something the compiler can work out. By *lifetime*, we mean the period of time for which the program will use the data, which again may or may not be determinable at compile time.

If the compiler can work out size and lifetime details of an item of data, it can ensure at compile time that it will be managed accurately and efficiently. If it cannot, memory for that data must be allocated at run time from the available operating system resources.

The kinds of data that the compiler can manage fall into one of two categories: *static data* and *stack-allocated data*.

Static data is data whose size can be determined by the compiler and whose lifetime is equal to the program's. Thus memory requirements for static data can be calculated at compile time, so that when the program starts up, memory for its static data will already be allocated and initialized. Global variables are an example of data that is typically static.

Stack-allocated data is data that exists only for the duration of a function call or the execution of some other block construct. Stack-allocated data is allocated at run-time on a stack; with their last-in, first-out (LIFO) semantics, stacks provide a natural automatic mechanism for allocating data upon entry to a function or block, and disposing of it upon exit. In addition to its lifetime, some programming language implementations may also need to know the data's size if it is to be stack-allocated.

If the compiler cannot work out the size or lifetime of an item of data, it must arrange for it to be allocated at run time, from the available operating system resources. We call the region of memory in which this allocation occurs the *heap*. The sort of data that ends up on the heap is that created with **`malloc`** and similar functions in C, the **`new`** operator in Java and C++, and the **`make`** function in Dylan and Common Lisp. (In C circles, this sort of allocation is called *dynamic* allocation.)

When this heap allocation is performed, part of the run-time system called the *allocator* serves the request for memory. Allocators keep a record of all heap memory known to be out of use or *free*. We call this record the *free list*. Each item in the free list is a reference to a block of memory. To serve an allocation request, the allocator searches the free list for a block of memory large enough to fit the data item being allocated.

For performing memory management, we are interested only in this latter category of data that the compiler cannot allocate — data allocated at run time on the heap. The fact that this data has an unknown lifetime implies we can *recycle* the memory it occupies if we can determine when it is no longer in use.

1.3.2 Recycling memory

Recycling memory involves reclaiming heap memory that stores data no longer required by the program, and then using it again to store new data. Thus it is roughly the opposite of the process of allocation. Rather like recycling things in the real world, recycling memory is worthwhile because it makes the best long-term use of limited resources, even if in the short term it requires effort.

1.3.3 Why heap growth is a concern

The reason to recycle heap memory is that it helps avoid unnecessary heap growth. If there is no more free memory on the heap, or no block of free memory large enough to contain the data that the program wants to create, the allocator is forced to ask the operating system for more memory, causing the heap to grow in size. As the heap grows, so does the size of the running program process. If the size of the heap is not carefully managed, it will soon cause problems. But why?

First, most modern operating systems implement a *virtual memory* system, which can make programs larger than a certain (system-dependent) size suffer serious performance problems. Virtual memory simulates a larger main memory than that physically available on the computer, by using the computer's backing storage devices, like hard disk, for temporary storage of the parts of running programs that are not currently being executed. This expanded memory address space can be used to run more programs simultaneously than would be possible if the system had to store every running program in main memory, and also makes it possible to run programs larger than the main memory.

But to provide this apparent memory expansion, the virtual memory system must swap or *page* programs or parts of programs between main memory and the slower backing store. We call this swapping process *paging* because virtual memory systems divide the virtual address space into fixed-size blocks of memory called *pages*; the system transfers memory in terms of these pages. Virtual memory systems treat the computer's main memory as a cache, and, like other cache systems, they use heuristics to help keep currently active program pages resident in main memory (or *paged in*), and, when necessary, to keep inactive program pages out in the backing store (*paged out*).

If a program tries to access a memory location that is currently paged out, the virtual memory system has to page in the page that contains the required memory location. Given the relative speeds of main memories like RAM and backing storage devices like hard disk, the cost of servicing accesses to paged-out memory is high. The larger a program's heap grows, the more pages it consumes, and the greater the chance that a memory access will require paging in order to be serviced. A massive reduction in overall system performance will occur when it spends more time paging than executing programs — a state we call *thrashing*.

The second problem we associate with heap growth is that if it continues without bound, the operating system will eventually run out of memory to give to the program, even in a virtual memory system. That is, the system can even run out of backing store. When the system runs out of memory in this way, the program will probably be unable to proceed further and is most likely to halt.

The longer a program runs, the more visible these problems will be. Even if they allocate massive amounts of memory on the heap, and do little or nothing to recycle it, programs that run for only a second or two are unlikely to degrade in performance to any noticeable degree, and just as unlikely to run out of memory. Having said that, even short-running programs can suffer when running alongside programs that hog the system memory, whether in a virtual memory system or not. The standard of the memory management in a program therefore affects not only its own performance but the performance of every simultaneously running program.

To maintain good performance in virtual memory systems, and to allay concerns about running out of memory, memory management must be performed constantly as the program runs. Only by doing so will the size of the heap remain within acceptable limits. Of course, heap growth is perfectly normal: sometimes all the memory on the heap really is in use by the program. But memory management must strive to avoid the heap growing because unused memory was left allocated.

1.3.4 How memory is recycled

To recycle heap memory we must first inform the allocator that the memory is available for re-use. We call this step *deallocating* or *freeing* the memory. When memory is freed, the allocator adds it to the free list and then re-allocates it at its own convenience. Thus, recycling is a two step process carried out by the allocator and a mechanism that frees memory.

Traditionally, programming languages have provided for freeing heap memory in one of two ways. C and C++ make freeing heap memory an explicit task for the programmer, and provide operations in the core language for the purpose. We give the name *manual memory management* to memory management schemes where the program frees heap memory explicitly.

By contrast, languages like Java and Dylan free unused heap memory automatically, and do not offer explicit deallocation operations. They achieve this automatic deallocation by relying on the run-time system to perform heap management automatically. We call this kind of memory management scheme *automatic memory management* or *garbage collection*, and the part of the run-time system devoted to this task a *garbage collector*.

1.3.5 Manual memory management

Manual memory management, then, is the name given to whatever method you, as the programmer, use to keep track of memory allocated through calls to operators like `malloc` and `new`, and then to release it for recycling when you know you are finished with it. This bookkeeping can be implicit or explicit in the program.

Manual memory management is only possible in languages that provide operations for allocating and deallocating blocks of memory. In C, when you know that some data is no longer used by the program, you can arrange for it to be recycled by with the operator `free`. C++ offers the `delete` operator for freeing objects allocated using `new`. C's `malloc` and `free` deal with raw bytes, which means that upon allocation you must form those bytes into the data type you want and also initialize them; by contrast, C++'s `new` and `delete` create and destroy well-formed objects.

These allocation and deallocation functions provide the interface to the allocator. When you call `malloc` and `new`, the allocator either returns a pointer to some free memory on the heap or, if the heap has no free block large enough, requests a new block of memory from the operating system and then returns a reference to that. When you call **`free`**, you pass the reference to the block you want to be recycled; the allocator returns that block to the free list.

1.3.6 Garbage collection

Garbage collection eliminates the bookkeeping burden that manual management places on programmers, by determining which heap memory is not in use and recycling it automatically. This allows you to allocate memory without having to keep track of it or worry about deallocating it.

Garbage collection is performed by a part of the program called a *garbage collector*. Garbage collectors can track and recycle all heap-allocated memory. They use a variety of techniques to find data that is no longer in use, and also to recycle the memory occupied by that data. In the next section, we look very briefly at how typical garbage collectors work.

Garbage collection is a standard feature of many languages. These languages include Java, Dylan, Common Lisp, Smalltalk, Standard ML, Perl, BASIC, the PostScript language, and AppleScript. We call these languages *garbage-collected* languages. Typical garbage-collected languages do not provide an explicit deallocation operator.

Garbage collection can also be added to languages like C++ and C, usually through integration with the allocator.

1.4 A rough guide to garbage collection

In this section, we look very briefly at how garbage collection works. As we have noted already, there are many garbage collection techniques, so we take a simplified, abstract look at common garbage collection techniques now, and will return to the topic in more detail later.

1.4.1 Overview and terminology

It is important to understand the terminology we use when discussing garbage collection. We say that allocated data is *live* if the program will access it in the future, and *dead* or *garbage* if it will not. The principal goal of garbage collection is to find garbage, and then return the heap memory it occupies to the allocator's free list.

Most garbage collectors identify data that is no longer in use by the program by exhaustively tracing the references that the program has to memory on the heap. The garbage collector starts the trace from the *root* data items in the program. These roots are data that is always considered to be live, such as global variables, and therefore never collected. The program or compiler must declare the *root set* to the garbage collector so that it knows both where to start tracing and what it is not permitted to collect.

If a path can be traced to a data item via references from the roots, that item is said to be *reachable*; otherwise, the item is *unreachable*. Because the program cannot access it in future, unreachable data is definitely garbage and can be recycled. The garbage collector automatically deallocates the memory occupied by unreachable data, thereby returning it to the allocator's free list and making it available for re-use.

In garbage collection terminology, the program that the garbage collector serves is known as the *mutator*. It is so called because, from the garbage collector's point of view, the program is nothing but a process that mutates the graph of references that the garbage collector traces. In general, a garbage collector knows nothing about the high-level purpose of the mutator, but it is possible to tune some garbage collectors to make them work better with a particular program's memory usage characteristics.

Garbage collectors usually work in cooperation with the allocator. In fact, in terms of its implementation, a garbage-collected language's garbage collector is often a part of the allocator. In languages that do not provide garbage collection by default, plug-in garbage collectors are often implemented as replacements for the language's heap-allocation function, for example C's **`malloc`**.

One benefit of the close relationship between the allocator and the garbage collector is that it allows the garbage collector to know the location and length of every allocated block on the heap. This means that a garbage collector cannot "lose" allocated blocks, leaving them unaccounted for.

1.4.2 The conservative approximation

Typical garbage collectors only recycle memory that is unreachable. If memory is reachable, the program *could* access it in the future, and it would probably be disastrous for the program if that memory was recycled. Equally, the program could retain a reference to memory without ever accessing it again. Thus liveness and reachability are *not* the same.

In general, garbage collectors cannot know whether reachable data is live or garbage, so for safety they assume that it is live. We call this assumption the *conservative approximation*. The conservative approximation can be the cause of unnecessary heap growth in garbage-collected systems, because data that is reachable but not live is not recycled. Later we study in some depth how to identify and rewrite code that causes data to remain reachable even when it is not live, therefore causing the garbage collector to retain it.

1.4.3 Styles of garbage collection

There are many styles of garbage collection. Here we introduce some of them. We study garbage collection algorithms in much greater detail in the Appendix.

Mark-sweep garbage collection works by marking each data item reached during the tracing phase, and then "sweeping up" the memory occupied by the unmarked data items afterwards: the unmarked data items are known to be unreachable. The sweeping phase does not need to trace the program references again, so simply examines each item on the heap in turn to see whether it is marked.

Copying garbage collection works by copying reachable data from one region of the heap into another, and then recycling the old, now unused, region. As with unmarked data in mark-sweep garbage collection, the data items that were not copied to the new region by copying garbage collection are by definition unreachable and therefore garbage. The data items that were copied can also be recycled because they are represented in the new region. Thus the entire old region can be recycled at once, which mitigates the cost of copying somewhat.

Simple versions of mark-sweep and copying garbage collection are not invoked until the allocator runs out of free memory. They then put mutator execution on hold until they have traced everything on the heap and collected any unreachable data. This pause is necessary for simple garbage collectors because if the mutator modified the graph of references during the tracing or collection phases, there could be errors in the collection.

Because pauses in execution can be unacceptable in interactive programs and real-time systems, *incremental* garbage collectors are now common. Incremental garbage collectors can interleave garbage collection work with mutator execution, therefore spreading the cost of garbage collection more evenly. Some incremental collectors can provide guarantees about how long they will spend working before returning control to the mutator.

1.5 Summary

Yet to be written.

2

The Problems of Memory Management

2.1 Introduction

Poor memory management can lead to programs performing poorly and even halting because the system cannot give them any more memory. It can also cause a program compute incorrect results or to crash.

In this chapter we look at the problems of memory management: the tasks that memory managers must carry out, and the pitfalls that they must avoid, whether they are manual management or garbage collection schemes. We compare manual schemes with garbage collection and show how garbage collection can provide a good solution to nearly all of these problems.

2.2 Efficiency requirements

All memory managers, whether they are manual management or garbage collection schemes, must strive to meet the following basic efficiency requirements: they must be thorough, they must recycle garbage in a timely fashion, and they must do their work unobtrusively. In the following subsections we examine each of these three requirements.

2.2.1 Thoroughness

If a memory manager keeps track of only a proportion of the memory that the program allocates on the heap, it reduces its chances of minimizing unnecessary heap growth. As we saw in Chapter 1, heap growth can degrade program performance severely in virtual memory systems and, if it continues without bound, can cause a program to run out of memory and crash. Accordingly, memory managers need to keep thorough account of heap allocation, so as to maximize the amount of memory they could discover to be garbage and therefore recycle.

In manual memory management, accounting is the programmer's responsibility, and accordingly its thoroughness depends on the effort put into it. By contrast, because most garbage collectors are integrated with the allocator, they are able to see the location and size of all data allocated on the heap, which means they can always find allocated data that is no longer in use. Hence they can recycle data that the program has, as it were, forgotten.

If you are writing a hybrid program where some components use garbage collection and others use manual memory management — such as a Java program linked via *native methods* to cooperate with someone else's code written in C — the different modules may use different allocators, in which case the garbage collector may not be able to manage the entire heap. Later we will examine ways to improve memory management in such hybrid programs.

2.2.2 Timeliness

Concerns about heap growth also lead to timeliness requirements. Heap growth can only be minimized if garbage is recycled promptly. In practice, that means discovering memory is garbage as soon as possible after it becomes unreachable, and then returning that memory to allocator's free list as soon as possible after that. Again, if the heap appears to be full of live data, the allocator will seek more memory from the operating system, which will cause the heap to grow in size. Timely recycling can help minimize heap growth.

Manual memory management can be *very* timely, if done correctly, since in theory the programmer knows exactly when the program is finished with an item of heap-allocated data. However, there is always the chance that manual management will be "too timely" if the programmer makes the slightest mistake — leading to a *premature free* (see Section 2.3.3 on page 29). Some manual managers seem to be written with the importance of timeliness overestimated and the possibility of making a mistake underestimated.

Garbage collection has a reputation for not being timely, which derives from the performance of ancient Lisp systems. Modern garbage collectors can be quite timely through *idle collection* (utilizing program idle time, when waiting for user input or other I/O completion) and *incremental collection* (interleaving unnoticeable pieces of garbage collection work with the mutator computation).

2.2.3 Unobtrusiveness

Because memory management is an administrative function rather than central to the purpose of the program, it should consume as small a proportion of overall program execution time as possible while still doing its job. Particularly, it should strive not to cause pauses in interactive programs, and in real-time programs must meet very strict bounds on pause times.

Because it is typically implemented as a coroutine or subroutine, manual memory management can be obtrusive. If allocation or free operations require the free list to be searched or coalesced, or if they require memory to be requested from or returned to the operating system, they may cause delays precisely when the program is doing important work. Inserting these overheads at the precise moment when the program is "working" — allocating or freeing data — is just about the worst time to do so if unobtrusiveness is a goal.

Garbage collection has a reputation for being obtrusive. However, modern garbage collectors performing incremental and idle collection typically run as a separate thread, and can utilize thread scheduling to defer many of these overhead tasks, or pre-compute them while the mutator is not actively working.

2.3 Accuracy problems

Perhaps still more important than efficiency is that memory management should be accurate. In this section, we look at some mistakes memory managers can make: *leaks*, *uncollected garbage*, *dangling references*, and *double frees*.

2.3.1 Leaks

When a memory manager fails to recycle memory that is no longer in use by the program, we say there has been a *memory leak*, or *leak* for short.

Leaks typically occur when the last reference to a data item is destroyed, but the memory management scheme fails to notice that it was the last reference and therefore that the object ought to be recycled. For example, this might occur because the last reference to some data was destroyed or lost without being sent to a deallocation operation such as **`delete`**.

Manual memory management schemes cannot recover from leaks. This is because the only way to deallocate heap-allocated memory in manual management is through explicit reference to it — functions like **`free`** and **`delete`** need a pointer to the memory they are asked to deallocate. But once some memory has leaked, the program has, by definition, no reference to it, and so it cannot be deallocated. In programs with complex heap memory usage, it is easy for a manual management scheme to lose track of references and leak memory.

There is no real notion of a leak in a garbage-collected system. A garbage collector does not need the program to have a reference to an item of data in order to recycle it; indeed, it is precisely because the program does not have such a reference that the garbage collector *will* recycle it. So while unreferenced data may waste memory on the heap temporarily, the garbage collector will soon remove it.

A small memory leak might not have noticeable effects on program performance, but larger leaks, or repeated small leaks, will cause a manually managed program's heap to grow considerably, with the usual problems that implies. Leaks in program modules are often unnoticeable during testing, only becoming visible when integrated into larger or longer running programs. One of the main reasons to add a garbage collector to a language like

C++ is to cure leaks when the manual management scheme becomes unmaintainable.

2.3.2 Uncollected garbage

When a memory manager fails to recycle data that is still referenced but no longer in use, we say that there is *uncollected garbage*. Garbage can go uncollected in this way in both manual management and garbage collected systems.

Typically, garbage goes uncollected in a manually managed program because of inaccurate accounting: the program fails to free something *and* retains a reference to it.

Uncollected garbage is the main inaccuracy of garbage collection, and is a result of the conservative approximation. Recall that the conservative approximation is the approximation of liveness with reachability: the garbage collector assumes that if something is reachable, it is also live, and thus will never recycle it. Some reachable data will never be used by the program again, but the garbage collector cannot know that.

You will get uncollected garbage in a garbage-collected system if you do not null out references you no longer need. Tools can help find this kind of stale data. Most stale references are in data that is itself garbage — they can be collected without difficulty. Some references, such as those in stack-allocated (function or block) variables automatically clean themselves up when they go out of scope. The only problematic references are those that can be reached from some global (static) structure; you must take care with these to prevent uncollected garbage.

2.3.3 Dangling references

When a memory manager recycles some memory that is still in use by the program, one or more *dangling references* to the memory are left behind. This problem is also known as a *premature free*.

Dangling references are particularly dangerous because they can change a program's behavior. The program assumes the dangling reference is still valid, but as far as the allocator is concerned, the memory that the reference points to is free and available for allocation to new data.

Dangling references do not necessarily cause problems right away: the next time the program follows the reference, the memory it refers to could be unchanged. But because the memory is available for reallocation at any time, it is almost inevitable that it will be allocated to some new data, making the values it contains subject to change.

Once the data's memory has been reallocated and reinitialized, subsequent accesses via the dangling reference will return unexpected values that may crash the program or, perhaps worse, produce incorrect results but be otherwise safe — forming a bug that can be extremely difficult to track down. If the program should modify the old data, it corrupts the new data, and vice versa.

Like leaks, dangling references are easy to create by mistake in manually managed programs. Garbage-collected programs can suffer from dangling references too, if the garbage collector contains a bug, but as we shall argue in "Software engineering issues" on page 33, garbage collectors are comparatively unlikely to do so.

2.3.4 Double frees

When a memory manager frees a block of memory that it has already freed, a *double free* has occurred. There are two ways this can happen.

The first is when the program frees an object *a* prematurely, and then frees the dangling references left behind before the allocator reallocates the memory that *a* occupied. The effect of this sort of double free depends on the allocator. If the allocator is capable of detecting and ignoring the fact that the freed block is already on its free list, the double free is innocuous. Other allocators, however, crash in these circumstances. Still other allocators add the free block to their free list again, which leads to undefined behavior in future allocations.

The second kind of double free is when the program frees object *a* prematurely, then reallocates the memory it occupied to object *b*, and then frees one of the dangling references to *a*. Thus *b* even though *b*'s memory management thus far has been correct, it is freed incorrectly. This double free again tests the robustness of the allocator, and also risks corruption of *b* because the program still has references to it.

This second kind of double free is extremely hard to debug. If *b* is in a separate module to *a*, no amount of debugging in *b*'s module will reveal the problem.

The bug will only occur when *b*'s module is running in concert with the broken module containing *a*.

In manual memory management, double frees are possible whenever there are dangling pointers. Double frees do not occur in a garbage-collected system for the simple reason that there is no explicit deallocation: garbage-collected languages offer no explicit deallocation operations, and plug-in garbage collectors for C and C++ replace the standard `free` and `delete` operators with operators that do nothing. (These "placebo" operators are supplied for compatibility with existing code; the garbage collector will carry out the real recycling of unused data. Premature `free` and `delete` operations are likely to be one reason the plug-in collector is being used, so it is essential that those operations be replaced with placebos.)

2.4 Heap fragmentation and compaction

Over time, some allocation policies cause the free and used memory in the heap to become heavily interleaved. Rather than having one contiguous free region and one contiguous used region, the heap contains a lot of variably sized free blocks interspersed with blocks that are in use. When the heap is in this state, we say it suffers from *fragmentation*. Memory leaks contribute to fragmentation, but it can simply be the result of allocating objects of varying sizes during program execution.

Fragmentation can cause difficulties for allocators, because for any given request, the chances of there being a block large enough to satisfy it are smaller than they would be if all the free memory was in one contiguous block. Increases in fragmentation could lead to program performance degrading rapidly, with longer and longer periods spent in allocation searching a long and growing free list. Some allocators avoid this by maintaining more than one free list, with each list containing free blocks of a single size. These dedicated free lists make it easier for the allocator to check whether it has a suitable free block.

If no free block is large enough to satisfy an allocation request, the allocator will turn to the operating system for more heap memory. If the memory manager does not fix the fragmentation, the heap will continue growing. Many memory managers attempt to fix fragmentation by *compacting* the heap so that the free blocks make up larger regions or a single region.

To be able to move blocks of memory, any memory manager needs to know the program intimately, so that it can safely update every reference to every block it moves.

In manual management schemes, the allocator is the obvious candidate for performing compaction, because only it has direct knowledge of the heap. But an allocator in a manually managed language typically knows nothing about the program's view of the memory on the heap. It simply allocates and frees blocks of memory, without understanding their contents or the relationships that exist between them. Typically, compacting manual allocators avoid the need to understand the graph of program references to data by hiding the real address of the memory from the program in a *handle* object, through which the program must refer to the memory indirectly. The allocator can invisibly move data for compaction by *atomically* updating the address in the handle.

But most languages that permit manual management also have *transparent representations* — that is, the details of data representation, in particular the address of the representation, are accessible to programs. That means using handles in such a language is inherently unsafe, because a program can always look inside a handle to see the real address it is pointing to. What is more, the indirection of handles is an extra overhead on every reference to heap data.

Manual memory managers that do support compacting typically wait until the heap is so fragmented that compaction is absolutely necessary, and then suspend the program while compacting. Because of this, errors where the program has incorrectly cached the address in a handle for efficiency may appear to work correctly but fail under stress.

By contrast, because garbage-collected languages already need to understand how to trace the graph of heap data references, they have exactly the knowledge necessary to compact the heap without resorting to using handles. Also, typical incremental collectors are already equipped with the mechanisms necessary to perform compaction incrementally, avoiding pauses.

Unfortunately, a plug-in garbage collector for a language with transparent representations cannot by itself make flexible compaction possible.

2.5 Locality of reference

Locality of reference is a term used to express how a program accesses memory. If a program repeatedly accesses one memory location, or successively accesses nearby memory locations, we say it has good locality of reference, because over time it accesses memory locations that are close together. A program that prints out each element of an array in turn would have good locality of reference, as long as the elements of the array were stored in a contiguous memory locations.

Good locality of reference is a desirable property for programs running under virtual memory, as it is for any cache architecture. If a program has good locality of reference, the memory accesses it makes will not increase paging, because they will successively refer to addresses on the same page. If a program has poor locality of reference, with a large proportion of successive memory accesses occurring on different pages, the chances that the virtual memory system will have to fetch a page from backing storage are increased. Fragmentation and memory leaks contribute to poor locality of reference.

It is possible to optimize locality of reference by organizing the heap to fit the program's memory access patterns. But to do so, the memory manager needs both a model of these patterns and the ability to reorganize the heap unobtrusively.

As we saw in "Heap fragmentation and compaction" on page 31, incremental movement of heap data is difficult in manual memory management, which makes locality of reference optimizations harder. Locality of reference optimizations are again simpler for garbage-collected languages, because incremental movement of heap data is safe. Garbage collected languages can also use the graph of reachable data as an approximation of data usage relationships. They can also track dynamic access patterns using the same mechanisms that support incremental collection and compaction.

2.6 Software engineering issues

This section examines software engineering issues in writing and maintaining memory managers.

2.6.1 Safety and correctness

The earlier points about leaks, uncollected garbage, dangling references, double frees, and fragmentation have shown that bad memory management leads to poor and unsafe programs. A program that suffers from dangling references and double frees will produce incorrect results or crash. A program with a persistent memory leak may cause the heap to grow without bound, slowing program execution and eventually halting the program; fragmentation also contributes to heap growth and poor overall utilization of resources.

By comparison with garbage collection, manual memory management is more likely to compromise program reliability. For instance, there is a greater risk of programmer error in manual management. This risk is due simply to the fact that in any programming project, memory management is not the problem we are supposed to be solving. Any garbage collector in wide use, however, is likely to have been tested, debugged, and proven reliable for managing memory in a variety of programs.

Garbage collection is simply more likely to be accurate. A garbage collector can never cause a dangling reference because of the conservative approximation — the use of reachability as an approximation of liveness. Garbage collectors never recycle memory unless they have proven that it is dead. Thus while garbage collectors may cause some garbage to go uncollected because it is dead but still referenced, they will never cause an error by collecting data prematurely. The central topic of this book is how to rid your programs of uncollected garbage, thereby minimizing the impact of this weakness.

2.6.2 Large-scale software development

Large-scale, modular software benefits from a uniform memory management policy across all modules. In manual memory management, when heap-allocated data passes across module interfaces, there must be some agreement about which module is responsible for disposing of it if it is no longer required. This issue can only make manual memory management schemes more complicated; uncertainties about which module is responsible for which data can be a major cause of errors.

In programs written in garbage-collected languages, all modules implicitly share the same memory management policy, making it much simpler to pass objects across module interfaces. However, similar uncertainties about deallocation can trouble hybrid programs where garbage-collected and manually managed modules are linked together. This is because the garbage collector may not be able to see the part of the heap that the manually managed or *foreign* module allocates in.

Some garbage-collected languages, including Java and Dylan, offer help through a *finalization* interface. Finalization allows a final action to be performed on an object that has been discovered to be garbage, before the collector recycles it. With finalization, if an object representing some foreign heap data becomes garbage, and the garbage-collected language is responsible for deallocating the memory associated with the foreign data, the finalization action can call the foreign language's explicit deallocation function. After that, the garbage-collected language's representation object can be collected without causing a leak in the foreign module's part of the heap.

2.6.3 Reuse and maintenance issues

Manual management is always a custom solution, tied to the particular program being worked on. Though there are common principles for writing manual management code, such as explicit *reference counting*, the principles themselves prescribe implementations that are so tightly coupled with the rest of the program code that they cannot be modularized and re-used. What is more, as a program is developed over time, its memory behavior may change sufficiently to render its custom-written manual management scheme obsolete.

Because it requires explicit coding within the program itself, manual memory management leads to more complex code and interdependent module interfaces that are cluttered by memory management requirements. By contrast, garbage collection simplifies code and module interfaces because it subsumes all memory management with a single, reusable mechanism.

Manual management suffers from this problem almost by definition. It requires that memory management and program code are integrated tightly; conversely, garbage collection requires a less intimate memory management policy that can be organized into a self-contained module. Garbage collection can therefore be reusable, with all of the implied reliability benefits noted in "Safety and correctness" on page 34.

2.7 Summary

Yet to be written.

3

Cooking with Garbage Collection

3.1 Introduction

In this chapter we present a series of "recipes" that will help you to get the most out of a garbage collected programming environment. The recipes are laid out in a "cookbook" format intended for easy browsing. They address a wide range of issues that may arise when programming in a garbage collected language, from mundane guidelines to "tricks of the GC masters" and from techniques for languages with built-in garbage collection to how to improve a legacy program when adding on a garbage collector.

3.2 Format of the recipes

Each recipe has a Symptoms section that lists typical signs that you may observe with your program that would indicate the recipe would be of help. There is a Diagnosis section that will help you determine if the recipe is applicable. The main body of each recipe goes into detail with program examples. In the sidebar accompanying each recipe, there is a thumbnail (indicated by ⇦) that can be skimmed to get the flavor of the recipe, references to related information in the book (indicated by ⇨) that can be followed to find other relevant recipes and chapters, and additional information of interest (indicated by &) that may provide background information on the examples, anecdotes, or other ancillary notes.

On the following page is a roadmap to the recipes with each section explaining its purpose. The recipes themselves follow, grouped by related symptoms.

Recipe Roadmap

Symptoms

- The symptoms that indicate this recipe may be applicable to your program are given in this section.
- Recipes are grouped by symptom, so looking at nearby recipes is a quick way to find the recipe that is most applicable.

In this section, a thumbnail, or overview, of the recipe is given. Typically it is only one or two paragraphs. Looking here is a good way to skim through the recipes.

Diagnosis

This section will help you to decide if the recipe is applicable by describing how the recipe interprets the symptoms and the proposed cure.

This section will direct you to related recipes if they are not in the same symptom category. It will also refer you to other sections of the book that may be of interest.

The main section of the recipe will go into detail giving an example program that demonstrates the symptoms and an example program that has been cured. Finally a general guideline is given for how to avoid the problem in the future.

Anecdotal information or other background information is given in this section. Sometimes this section is used to explain intricacies of the example for programmers who may not be familiar with the example language.

String Theory

Symptoms

- High number of recycled strings
- A single function is responsible for the majority of recycled strings.
- The recycled strings are intermediate or temporary values.

Diagnosis

The garbage strings are all coming from one *allocation site* and that site is inside a loop. The challenge is to see if the allocation can be moved out of the loop.

Strings are a common building block in many programs and can be responsible for a significant proportion of the program memory resources. In a manual management system, strings must be carefully accounted for to prevent them from *leaking* and wasting memory resources. In a garbage collected system, string manipulation is much easier because the garbage collector will automatically recover strings when they are no longer used. Nevertheless, profligate use of strings can cause the garbage collector to have to work much harder than necessary.

Here is an example in Java of how it is easy to create a lot of garbage with an apparently simple program. The program simply calculates square roots and prints them out. It uses the Java + operator to compose each output line in a string which is then passed to the system print method:

Strings are one of the most common objects allocated by programs, especially programs that interact with the user. Often strings are created and used only once, although it may be difficult for the compiler to infer this.

When strings are a major source of garbage, it is worthwhile to look for strings being created in a loop and to try to move the creation out of the loop.

If you cannot reduce allocation as suggested here, *Working with the Garbage Collector* on page 121 discusses ways to control when collections occur.

The string buffer technique illustrated in this example is a special case of using a *resource*. See *Very Resourceful* on page 123 for a discussion of resources in general.

&

Most object-oriented languages that support strings also support string buffers. A *string buffer* consists of a string and a *fill pointer*, an indicator of how much of the string is valid. Inserting characters into the string at the fill pointer and adjusting the fill pointer lets you append to a string buffer without creating a new string. (Of course, this is a *destructive* operation — it changes the string buffer directly so that anyone referring to the string buffer sees the new value.) If you try to make the string buffer bigger than the underlying string, it grows automatically by allocating a new string and copying the contents of the old one into it. Most string buffer implementations let you specify an initial size to minimize the number of automatic grow operations.

In some languages, such as Lisp and Dylan, fixed-length strings and string buffers can be used interchangeably. In Java, a string buffer is automatically converted to a string if it is used where a string is expected. The former approach avoids making copies; the latter approach prevents unexpected sharing errors.

&

There are many forms of *loop optimization*. Most compilers will do some loop optimization. Simple optimizations involve moving *loop invariants*, computations that do not change each time around the

```
public class PrintSquareRoots {

  public static void main(String args[]) {
    int limit = args.length > 0 ?
      Integer.parseInt(args[1], 10) : 10000;

    for (int i = 0; i < limit; i++) {
      System.out.println( "Sqrt(" + i + ") = "
        + Math.sqrt(i));
    }

}
```

A good Java compiler will be savvy enough to not create separate strings for each use of the + operator. Instead it will create a `StringBuffer` to assemble the string in and convert the buffer to a `String` to pass to `System.out.println`. Unfortunately, most Java compilers will create a new string buffer each time around the loop. Potentially, the string buffer will be grown by each + operation too. This allocating and growing, each time around the loop, causes excess work for the garbage collector.

In this case, there is a fairly simple solution: we can allocate our own `StringBuffer`, outside the loop and reuse it each time around the loop. Here is a rewritten loop that creates only a single `StringBuffer`:

```
public class PrintSquareRoots {

  public static void main(String args[]) {
    int limit = args.length > 0 ?
      Integer.parseInt(args[1], 10) : 10000;
    StringBuffer buf = new StringBuffer();

    for (int i = 0; i < limit; i++) {
      buf.setLength(0);
      buf.append("Sqrt(").append(i);
      buf.append(") = ").append(Math.sqrt(i));
      System.out.println(buf);
    }

}
```

Here we explicitly create a single `StringBuffer`, outside the loop. Each time we enter the loop, we empty the string buffer by setting its length to 0, then we append each of the strings to the buffer and pass the buffer on to `System.out.println`. Appending to a buffer automatically adjusts the length of the buffer, and passing it as an argument to a function that takes a `String` argument automatically calls the buffer's `toString` method, which converts the contents up to the current length into a string.

Essentially we are manually managing the storage of the string buffer here, so the garbage collector does not have to recycle all the temporary buffers created by the original program. This is a reasonably safe optimization, but could not be used if we were passing the buffer on to a function that we did not completely understand. It is only because we know that the call to `System.out.println` will implicitly call the buffer's `toString` method, taking a snapshot of the current buffer contents, that we can reuse the buffer next time around the loop without disturbing our display of square roots.

There still are some intermediate results allocated in this example. When the buffer's `append` method is called on `i` and `Math.sqrt(i)`, these numbers must be converted into strings. Calling the buffer's `toString` method creates a string that might not be necessary. *All Strung Out* on page 113 discusses methods to eliminate even these intermediate results.

loop, out of the loop. A more difficult optimization is *strength reduction*, substituting a less complex or expensive operation that achieves the same effect, often relying on realizing that the computation is an *arithmetic progression* that can be calculated incrementally each time around the loop.

Few compilers are sophisticated enough to realize that allocation can be a loop invariant or that replacing a string with a string buffer is a form of strength reduction. Ideally, you want an optimization like the one illustrated here to be done automatically by the compiler. Like any optimization that is done by hand, it results in less readable code, which may in turn lead to errors and poorer maintainability.

Less Filling

Symptoms

- High number of recycled objects
- Recycled objects show “clustering” — there are one or more groups of recycled objects with the same type, size or type and size. There may be many objects with the same value or contents.
- The clusters of objects have a small number of *allocation sites*, often class constructors.

Diagnosis

The clusters of objects can be identified as pieces of the representations of program classes that dynamically allocate memory for those representations.

These symptoms typically occur in C++ programs that were originally implemented with manual memory management when an add-on garbage collector is later added to reduce bugs and maintenance costs. (It may be that memory management problems with the classes or representations identified in the symptoms were the reason the garbage collector was added.)

In a manually managed system, the programmer must carefully design classes that dynamically allocate memory for their representation if they are to behave correctly when declared, assigned, and passed as arguments. The default copy constructor and assignment operator provided by C++ will cause leaks, dangling pointers and double frees, hence must be overloaded for each class.

When garbage collection is added to C++, many of the “boilerplate” member functions you are used to supplying become unnecessary. More importantly, these unnecessary member functions can actually make the garbage collector do *more* work.

If you add a garbage collector to a C++ program that was originally written for manual memory management, you can improve the performance by considering the change in balance of **new** and **delete** calls.

An *add-on* garbage collector can eliminate bugs and reduce maintenance costs because it automatically eliminates leaks, dangling pointers, and double frees. But because the program that you add it on to was not designed with garbage collection in mind, there will certainly need to be some tuning to get the best performance. See *Add-on Garbage Collectors* on page 99 for an introduction to adding a garbage collector to a manually managed language.

&

In C++, if you want to be able to declare objects of your class without initializing them, you have to provide a *default constructor* — one will be provided automatically only if there are no other constructors for your class. Typically, you will want to provide one, so that even uninitialized instances of your class are well-behaved. In the example, the default constructor creates an empty string. We could have initialized the representation to **0**, but then every member function would have to test for that special case.

In C++ a *copy constructor* will be automatically provided if needed, but it simply does member-by-member copying. This can work if all members are classes and *their* copy constructors are designed properly. In general, this does not work for non-class members, however, because they will end up being shared between the copy and the original. If the shared member is deleted in the class destructor, there will be a *dangling pointer* when either the original or copy is deleted, and there will be a *double free* when the other is deleted. In the example, a separate copy is created to avoid this problem.

Similarly, in C++ a default *assignment operator* is provided if needed, but it also simply does member-by-member assigning.

The simplest design for a class that has dynamically allocated representation members involves copying the representation any time an object of that class is initialized, assigned, or passed by value. Here is an example of a simple `String` class one might write to improve on `char*`:

```
class String {
public:
  int length() const;
  String();
  String(const String&);
  String& operator=(const String&);
  ~String();
  String(const char*);
private:
  // Never modified, so declare const
  const char *data;
};

int String::length() const {
  return strlen(data);
}

// Default constructor
String::String() {
  // Ensure always a valid object
  data = strcpy(new char[1], "");
}

// Copy constructor
String::String(const String& s) {
  data = strcpy(new char[s.length() + 1], s.data);
}

// Assignment operator
String& String::operator=(const String& s) {
  // Use care if assigning to self
  if (data == s.data)
    return *this;
  delete[] data;
  data = strcpy(new char[s.length() + 1], s.data);
  return *this;
}
```

```
// Destructor
String::~String() {
  delete[] data;
}

// Custom constructor, from char*
String::String(const char *str) {
  data = strcpy(new char[strlen(str) + 1], str);
}
```

Perhaps some commentary is needed. Really all that is desired is a class that can hold character strings (other member functions would need to be added to make the class useful). But with manual management, there is a significant amount of work to do simply to get the class to work properly even before adding functionality. There must be a default constructor that allocates a default representation. To prevent leaks, double frees, and dangling pointers, there must also be a copy constructor, and `operator=` must be overloaded, so that the dynamically allocated representations in the class instances are properly copied and deleted, or if they are to be shared, reference count bookkeeping must be updated (we have shown the former here).

When garbage collection is added to C++, the `delete` operator is replaced with a *placebo*; it is there for compatibility with existing code, but it does not actually do any deleting. All recycling is now done by the garbage collector. But if we simply use our existing C++ programming style, if we do not take advantage of the simplifications that garbage collection permits, we end up making many extra copies of representations that will have to be swept up by the garbage collector. We should examine each `new` in our old code — only those that are there to prevent unwanted sharing are needed under a garbage collector. Any `new` that is there simply to balance a (now inert) `delete`, is superfluous. Any `new` that is there as a part of a memory management scheme, is also suspect.

This can work if all members are classes and *their* assignment operators are designed properly. In general, this does not work for non-class members, because the default assignment will simply overwrite each member. If the non-class member was expected to be deleted in the class destructor, the overwriting will result in a *leak*. In the example, we delete any previous value before assigning to prevent such a leak (being careful *not* to delete in the obscure case where an object is assigned to itself).

&

A quirk of the C++ language is that the **`const`** type modifier, when used on a pointer or reference *formal parameter*, can mean either that the function will not modify that argument, or, if there is another function with the same name and parameters, differing only in the **`const`** modifier, that the function is overloaded and one or the other will apply, depending on whether the *actual parameter* is a **`const`** object or not.

In our examples, the use of **`const`** in the constructor from **`char*`** is a promise that the constructor will not modify its argument — it simply makes a copy of the character array. If we really trusted all clients of our code not to use any loopholes to modify **const** initialization arguments, we could have separate constructors for **char*** and **const char*** and not copy the initialization argument in the latter. We feel such an optimization would be risky because of the many ways in which **const** objects can be surreptitiously modified. In the case of the example, such an optimization would be particularly risky because of another quirk of C++: compilers are permitted to share string literals, so they should be treated by the programmer as **const char[]**, but for compatibility with Classic C they are of type **char[]**, which means compliers are not required to prevent a program from modifying them!

When garbage collection is added to C++, we can greatly simplify our program, down to just what we wanted, a class that holds character strings:

```
class String {
public:
  int length() const;
  String();
  String(const char*);
private:
  // Never modified, so declare const
  const char *data;
};

int String::length() const {
  return strlen(data);
}

// Default constructor
String::String() {
  // Ensure always a valid object
  data = (const char *)"";
}

// Custom constructor, from char*
String::String(const char *str) {
  // Copied to prevent unwanted sharing
  data = strcpy(new char[strlen(str) + 1], str);
}
```

With a garbage collector, we do not need a destructor — the garbage collector takes care of finding and recycling unused representations. Since we are not manually freeing the representation in a class destructor, we do not need the default constructor to allocate a null representation — it is sufficient to initialize the representation to a constant empty string (the garbage collector will not attempt to recycle static constants). With a garbage collector, we do not need a copy constructor or to overload `operator=` — the default member functions that copy or assign member-by-member will work correctly under garbage collection: an object that is referred to by more than one member (due to copying or assignment) will not be recycled and an object that is referred to by no members (due to its last reference being overwritten by an assignment) will automatically be recycled.

We do still make a copy of the initialization argument in the constructor from `char*`. We do not want to share that value, otherwise a modification to the initialization argument would cause the value of our newly constructed string to change too.

If you add garbage collector to a legacy C++ program, you will get better performance if you go through your classes and examine the default constructor, copy constructor, assignment operator, and destructors you have written to implement manual management bookkeeping and consider whether allocations that they make are still necessary. In simple cases, you may not need to define these standard member functions at all.