Preface

Version 25/04/00 10:10 - 2

Once upon a time computer memory was one of the most expensive commodities on earth, and large amounts of human ingenuity were spent trying to simulate supernova explosions with nothing more than a future Nobel prize winner and a vast array of valves. Nowadays many people have enough computer memory to support simulating the destruction of most of the galaxy in any one of their hand-held phones, digital diaries, or microwave ovens.

But at least two things have remained constant throughout the history of computing. Software design remains hard [Gamma et al 1995], and its functionality still expands to fill the memory available [Potter 1948]. This book addresses both these issues. Patterns have proved a successful format to capture knowledge about software design; these patterns in particular tackle memory requirements.

As authors we had other several additional aims in writing this book. As patterns researchers and writers we wanted to learn more about patterns and pattern writing, and as software designers and architects we wanted to study existing systems to learn from them. In particular:

- We wanted to gain and share an in-depth knowledge of portable small memory techniques; techniques that work in many different environments.
- We wanted to write a complete set of patterns dealing with one single force — in this case, memory requirements.
- We wanted to study the relationships between patterns, and to group and order the patterns based on these mutual relationships, and lastly:
- We wanted an approachable book, one to skim for fun rather than to suffer as a penance.

This book is the result. It's written for software developers and architects, like ourselves, whether or not you may happen to be facing memory constraints in your immediate work.

To make the book more approachable (and more fun to write) we've taken a light-hearted slant in most of our examples for the patterns, and Duane Bibby's cartoons are delightfully frivolous. If frivolity doesn't appeal to you, please ignore the cartoons and the paragraphs describing the examples: the remaining text is as rigorous as we can make it.

This book is still a work in progress. We have incorporated the comments of many people, and we welcome more. You can contact us at our web site, `http://www.smallmemory.com/`

Dedication

To Julia and to Katherine.

Who have suffered long and are kind.

Acknowledgements

No book can be the work of just its authors. First, we need to thank John Vlissides, our indefatigable series editor: we still have copies of the email where he suggested this mad endeavour, and we're grateful for his many comments on our patterns, from the original EuroPLoP paper to the final drafts. Second, this book would not be the same without Duane Bibby's illustrations, and we hope you like them as much as we do.

We take the blame for this book's many weaknesses, but credit for most of its strengths in this book goes to the those members of the Memory Preservation Society (and fellow travellers) who took the time to read and comment on drafts of the patterns and the manuscript. These people include but are not limited to John Vlissides (again), Paul Dyson, Linda Rising, Klaus Marquardt, and Liping Zhao (EuroPLoP and KoalaPLoP shepherds for these patterns), Tim Bell, Jim Coplien, Frank Buschmann, Alan Dearle, Martine Devos, Martin Fowler, Nick Grattan, Neil Harrison, Benedict Heal, David Holmes, Ian Horrocks, Nick Healy, Dave Mery, Matt Millar, Alistair Moffat, Eliot Moss, Alan O'Callaghan, Will Ramsey, Michael Richmond, Hans Rohnert, Andreas Rüping, Peter Sommerlad, Laurence Vanhelsuwe, Malcolm Weir, and the Software Architecture Group at the University of Illinois and Urbana-Champagne, including: Federico Balaguer, John Brant, Alan Carrol, Ian Chai, Diego Fernandez, Brian Foote, Alejandra Garrido, John Han, Peter Hatch, Ralph Johnson, Apu Kapadia, Aaron Klish, An Le, Dragos-Anton Manolescu, Brian Marick, Reza Razavi, Don Roberts, Paul Rubel, Les Tyrrell, Roger Whitney, Weerasak Witthawaskul, Joseph W. Yoder, and Bosko Zivaljevic.

The team at Addison-Wesley UK (or Pearson Education, we forget which) have been great in dealing with two authors on opposite sides of the globe. We'd like to thank Sally Mortimore (for starting it all off), Allison Birtwell (for finishing it all up), and Katherin Elkstrom (for staying the distance). Credit goes also to two artists, George Platts for suggesting illustrations and Trevor Coard for creating many of them.

Finally, we must thank all the members of the patterns community, especially those who have attended the EuroPLOP conferences in Kloster Irsee. We are both admirers of the patterns 'literature' (in the same way one might be might be a fan of science fiction literature) and hope this collection of patterns will be a worthy contribution to the cannon.

Contents

[Plus the other chapters …]

Introduction

"Small is Beautiful"
E. F. Schumacher

"You can never be too rich or too thin"
Barbara Hutton

Designing small software that can run efficiently in a limited memory space was, until recently a dying art. PCs, workstations and mainframes appeared to have exponentially increasing amounts of memory and processor speed, and it was becoming rare for programmers even to need to think about memory constraints.

At the turn of a new century, we're discovering an imminent market of hundreds of millions of mobile devices, demanding enormous amounts of high-specification software; physical size and power limitations means these devices will have relatively limited memory. At the same time, the programmers of Web and database servers are finding that their applications must be memory-efficient to support the hundreds of thousands of simultaneous users they need for profitability. Even PC and workstation programmers are finding that the demands of video and multi-media can challenge their system's memory capacities beyond reasonable limits. Small memory software is back!

But what is small memory software? Memory size, like riches or beauty, is always relative. Whether a particular amount of memory is small or large depends on the requirements the software should meet, on the underlying software and hardware architecture, and on much else. A weather-calculation program on a vast computer may be just as constrained by memory limits as a word-processor running on a mobile phone, or an embedded application on a smart card. Therefore:

Small memory software is any software that doesn't have as much memory as you'd like!

This book is written for programmers, designers and architects of small memory software. You may be designing and implementing a new system, maintaining an existing one, or merely seeking to expand your knowledge of software design.

In this book we've described the most important programming techniques we've encountered in successful small memory systems. We've analysed the techniques as *patterns* – descriptions in a particular form of things already known to work [Alexander 1977, 1979]. Patterns are not invented, but are identified or mined from existing systems and practices. To produce the patterns in this book we've investigated the design of many successful systems that run on small machines. This book distils the essence of the techniques that seem most responsible for the systems' success.

The patterns in this book consider only limitations on memory: Random Access Memory (RAM), and to a lesser extent Read Only Memory (ROM) and Secondary Storage, such as disk or battery backed RAM. A practical system will have many other limitations; there may be constraints on graphics and output resources, network bandwidth, processing power, real-time responsiveness, to name just a few. Although we focus on memory requirements, some of these patterns may help with these other constraints; others will be less appropriate: compression, for example, may be unsuitable where there are significant constraints on processor power; paging is unhelpful for real-time performance. We've indicated in the individual patterns how they may help or hinder supporting other constraints.

The rest of this chapter introduces the patterns in more detail. The sections are as follows:

How to use this book	Suggests how you might approach this book if you don't want to read every page.
Introduction to Small Memory	Describes the problem in detail, and contrasts typical kinds of memory-constrained software.
Introduction to Patterns	Introduces patterns and explains the pattern format used in this book
The Patterns in this Book	Suggests several different ways of locating, relating and contrasting all the patterns in the book.

How to use this book

You can, of course, start reading this book at page one and continue through to the end. But many people will prefer to use this book as a combination of several things:

- A programmer's introduction to small memory software.
- A quick overview of all the techniques you might want to use.
- A reference book to consult when you have a problem you need to solve.
- An implementation guide showing the tricks and pitfalls of using common – or less common – patterns.

The following sections explain how to use this book for each of the above, and also for other more specialised purposes:

- Solving a particular strategic problem
- Academic study
- Keeping the boss happy

A Programmer's Introduction to Small Memory Software.

Perhaps you're starting as a new developer on a memory-constrained project, and haven't worked on these kinds of projects before.

If so, you'll want to read about the programming and design-level patterns that will affect your daily work. Often your major design concern will initially be class design, so start with the straightforward and fun PACKED DATA (ppp). You can then continue exploring several other SMALL DATA STRUCTURE patterns used a lot in memory-limited systems: SHARING (PPP), COPY-ON-WRITE (PPP) and EMBEDDED POINTER (PPP).

The immediate choices in coding are often how to allocate data structures, so next compare the three common forms of memory allocation: FIXED ALLOCATION (PPP), VARIABLE ALLOCATION (PPP) and MEMORY DISCARD (PPP). Equally important is how you'll have to handle running out of memory, so have a look at the important PARTIAL FAILURE (PPP) pattern, and perhaps also the simple MEMORY LIMIT (PPP) one.

Finally, most practical small memory systems will use the machine hardware in different ways to save memory. So explore the possibilities of READ-ONLY MEMORY (PPP), APPLICATION SWITCHING (ppp) and DATA FILES (PPP).

The description of each of these patterns discusses how other patterns complement them or provide alternatives, so by reading these patterns you can learn the most important techniques and get an overview of the rest of the book.

Quick Overview of all the Techniques

A crucial benefit of a collection of patterns is that it creates a shared *language* of pattern names to use when you're discussing the topic [Gamma Helm Johnson Vlissides 1995, Coplien 1996]. To learn this language, you can scan all the patterns quickly, reading the main substance but ignoring all the gritty details, code and implementation notes. We've structured this book to make this easy to do. Start at the first pattern (SMALL ARCHITECTURE, page XXX) and read through each pattern down to the first break:

❖ ❖ ❖

This first part of the pattern provides all you really need to know about it: the problem, its context, a simple example, and the solution. Skimming all the patterns in this way takes a careful reader couple of hours, and provides an overview of all the patterns with enough detail that you can begin to remember the names and basic ideas of the patterns.

Reference for Problem Solving

Perhaps you're already working on a project, in a desperate hurry, but faced with a thorny problem with no simple answer. In this case, first consult the brief summaries of patterns in the front cover. If one or more patterns look suitable, then turn to each one and read its bullet points and 'Therefore' paragraph to see if it's really what you're after.

If that approach doesn't produce a perfect match, then use the index to look for keywords related to your problem, and again scan the patterns to see which are suitable.

If none of the patterns you've found so far are quite what you want, then have a look at the summary pattern diagram in the back cover – there may be useful patterns related to the ones you've already checked. Check the 'See Also' sections at the end of the patterns that seem most useful; perhaps one of the related patterns might address your problem.

Implementation Guide

Perhaps you've already decided that one or more of the patterns are right for you. You may have known the technique all along, although you've not thought of it as a pattern, and you've decided – or been told – to use it to implement part of your system.

In this case you can consult the full text for each specific pattern you've chosen. Find the pattern using the summary in the front cover, and turn to the Implementation section for a discussion of many of the issues you'll come across in using the pattern, and some of the techniques other implementers have successfully used to solve them. If you prefer looking at specifics like code, turn to the Example section first and then move back to the Implementation section.

You can also look at the Known Uses section – perhaps the systems will be familiar and you can find out how they've implemented the pattern, and the 'See Also' section guides you to other patterns you may find useful.

Helping Define a Project Strategy

If you're defining the overall strategy for a software project [Goldberg and Rubin 1995], you'll probably be concerned about many other issues in addition to memory restrictions. Maybe you're worried about time performance, real-time constraints, a hurried delivery schedule or a need for the system to last for several decades.

In this case turn to the discussion in appendix XXX. These concerns are called 'forces' [Alexander 1979]. Scan the chapter to identify the forces you're interested in; the sections on each force will tell you which patterns will best suit your needs.

Academic Study

Of course many people still enjoy reading books from start to finish. We have written this book so that it can also be read in this traditional, second millennium style.

Each chapter starts with the simpler patterns that are easy to understand, and progresses towards the more sophisticated patterns; the patterns that come first in a chapter lead to the patterns that follow afterwards. The chapters make a similar progression, starting with the large-scale patterns you are most likely to need early in a project (Architectural Patterns) and progressing to the most implementation-specific patterns (Memory Allocation).

Keeping the Boss Happy

Maybe you really don’t care about this stuff at all, but your manager has bought this book for you and you want to retain your credibility. In this case, leave the book open face down on a radiator for three days. The book will then look as though you’ve read it, without any effort required on your part [Covey 1990].

Introduction to Small Memory

What makes a system small? We expect that the patterns in this book will be most useful for systems with memory capacities roughly between 50K to 10M bytes total, although many of the patterns are frequently used with much smaller and even much larger systems.

Here are four different kinds of projects that can have difficult meeting their memory requirements, and consequently can benefit from the patterns in this book:

1. Mobile Computing

Palmtops, pagers, mobile phones, and similar devices are becoming increasingly important. Users of these mobile machines are demanding more complex and feature-ridden software, ultimately comparable to that on their desktops. But, compared to desktop systems, a portable device's hardware resources, particularly memory, are quite limited. Because of their ubiquitous nature [Norman 1998] these machines also need to be more robust than desktop machines — a digital diary with no hard disk cannot be restarted without loosing its data.

Developments for such machines must take far more care with memory constraints than in similar applications for PCs and Workstations. Virtually all the patterns in this book may be relevant to any given project.

2. Embedded Systems

A second category of physically small device is embedded systems such as process control systems, medical systems and smart-cards. When a posh new car can have more than a hundred microprocessors in it, and with predictions that we'll all have several embedded microprocessors in our bodies within the next decade, this is a very important area.

Embedded devices are limited by their memory, and have to be robust, but in addition they often have to meet hard real-time processing deadlines. If they are life critical or mission critical they must meet stringent quality control and auditing requirements too.

In systems with memory capacity is much below about 50 Kbytes, the software must be tightly optimised for memory, and typically must make drastic tradeoffs of functionality to fit into the available memory. In particular, the entire object-orientated paradigm, though possible, becomes less helpful as heap allocation becomes inappropriate. When implementing systems below 50K, the Allocation and Data Structure patterns are probably the most important.

3. Small Slice of a Big Pie

Many ostensibly huge machines — mainframes, minicomputers or PC servers— can also face problems with memory capacity. These very large machines are most cost-effective when supporting hundreds, thousands, or even hundreds of thousands of simultaneous sessions. Even though they are physically very large, with huge physical memory capacities and communication bandwidths, their large workloads often leave relatively modest amounts of memory for each individual session.

For example, most Java virtual machines in 2000 require at least 10Mb of memory to run. Yet a Java-based web server may need to support ten thousand simultaneous sessions. Naively replicating a single user virtual machine for each session would require a real hardware server with 100 Gigabytes of main memory, not counting the memory required for the application on each virtual machine. The patterns in this book, particularly the Data Structure patterns, can increase the capacity of such servers to support large numbers of users.

4. Big Problems on Big Machines

In single-user systems with a memory capacity greater than 10 Mbytes, memory is rarely the major concern for most applications, but general-purpose computers can still suffer from limited memory capacity.

For example, organisations may have a large investment in particular hardware with a set memory capacity that it is not feasible to increase. What happens if you're a bank with twenty thousand three-year old PCs sitting on your tellers' desks and would like to upgrade the software? What happens if you bought a new 1 Gigabyte server last year, can't afford this year's model, but need to process 2 Gigabytes this year? Even if you could afford to upgrade the machines, other demands (such as your staff bonus) may have higher priority.

Alternatively, you may be working on an application that must handle very large amounts of data, such as multi-media editing, video processing, pattern recognition, weather prediction, or maintaining a collection of detailed bitmap images of the entire world. Any such application could easily exhaust the RAM in even a large system, so you'll need careful design to limit its memory use. For such applications, the Secondary Storage and Compression patterns are particularly important.

Ultimately, no computer can ever have enough memory. Users can always run more simultaneous tasks, process larger data sets, or simply choose a less expensive machine with a lower physical memory capacity. In a small way, every machine has small memory.

Types of Memory Constraint

Imagine you're just starting a new project in a new environment. How can you determine which memory constraints are likely to be a problem, and what types of constraint will give the most trouble?

Hardware Constraints.

Depending on your system, you may have constraints on one or more of the following types of memory:

RAM Memory	Used for executing code, execution stacks, transient data and persistent data.
ROM Memory	Used for executing code and read-only data.
Secondary Storage	Used for code storage, read-only data and persistent data.

You may also have more specific constraints: for example stack size may be limited, or you may have both dynamic RAM, which is fast but requires power to keep its data, and static RAM, which is slower but will keep data with very little power.

RAM is usually the most expensive form of memory, so many types of system keep code on secondary storage and load it into RAM memory only when it's needed; they may also share the loaded code between different users or applications.

Software Constraints

Most software environments don't represent their memory use in terms of main memory, ROM and secondary storage. Instead, as a designer, you'll usually find yourself dealing with heap, stack and file sizes. Table XXX below shows typical attributes of software and how each maps to the types of physical memory discussed above.

Attribute	Where it Lives
Persistent data	Secondary storage or RAM.
Heap and static data	RAM
Code Storage	Secondary storage or ROM
Executing Code	RAM or ROM
Stack	RAM

Different Types of Memory-Constrained System

Different kinds of systems have different resources and different constraints. Table 3 describes four typical kinds of system: embedded systems, mobile phones or digital assistants, PCs or workstations, and large mainframe servers. This table is intended to be a general guide, and few practical systems will match it exactly. An embedded system for a network card will most certainly have network support, for example; many mainframes may provide GUI terminals; a games console might lie somewhere between an embedded system and a PDA.

	Embedded System	Mobile Phone PDA	PC, Workstation	Mainframe or Server Farm
Typical Applicat-ions	Device control, protocol conversion, etc.	Diary, Address book, Phone, Email	Word processing, spreadsheet, small database, accounting.	E-commerce, large database applications, accounting, stock control.
UI	None.	GUI; libraries in ROM	GUI, with several possible libraries as DLLs on disk	Implemented by clients, browsers or terminals
Network	None, Serial Connection, or Industrial LAN	TCP/IP over a wireless connection	10MBps LAN	100 MBps LAN
Other IO	As needed – often the main purpose of device	Serial connections	Serial and parallel ports, modem, etc.	Any, accessed via LAN

Table 3: Comparison of different kinds of system

All these environments will normally keep transient program and stack data in RAM, but differ considerably in their other memory use. Table XXX below shows how each of these kinds of system typically implements each kind of software memory.

	Embedded System	Mobile Phone, PDA	PC, Workstation	Mainframe or Server Farm
Vendor-supplied Code	ROM	ROM	On disk, loaded to RAM	Disk, loaded to RAM

3rd Party Code	None	Loaded to RAM from flash memory	As vendor-supplied code	As vendor-supplied code
Shared Code	None	DLLs shared between multiple applications	DLLs shared between multiple applications	DLL and application code shared between multiple users
Persistent Data	None, or RAM	RAM or flash memory.	Local hard disk or network server.	Secondary disk devices.

Table 1: Memory Use on each Type of System

Note how mobile phones and PDAs treat third-party code differently from vendor-supplied code, since the former cannot live in ROM.

Relative Importance of Memory Constraints

Table 4 shows the importance of the different constraints on memory for typical applications on each kind of system discussed above. Three stars mean the constraint is usually the chief driver for a typical project architecture; two stars mean it is an important design consideration. One star means the constraint that may need some effort from programmers but probably won't affect the architecture significantly; and no stars mean it's virtually irrelevant to development.

	Embedded System	Wireless PDA	PC, Workstation	Mainframe or Server Farm
Code Storage	✱✱	✱✱		
Code Working Set		✱✱	✱	
Heap and Stack	✱✱✱	✱✱	✱	✱
Persistent Data	✱✱✱	✱		

Table 4: Importance of Memory Constraints

In practice, every development is different; there will be some smart-card applications that can virtually ignore the restrictions on heap and stack memory, just as there will be some mainframe applications where the main constraint is on persistent storage.

Introduction to Patterns

What, then, actually *is* a pattern? The short answer is that a pattern is a "*solution to a problem in a context*" [Alexander 1977,Coplien 1996]. This focus on context is important, because with a large number of patterns it can be difficult to identify the best patterns to use. All the patterns in this book, for example, solve the same problem – too little memory – but they solve it in many different ways.

A pattern is not just a particular solution to a particular problem. One of the reasons programming is hard is that no two programming problems are exactly alike, so a technique that solves one very specific problem is not much use in general [Jackson 1995]. Instead, a pattern is a generalised description of a solution that solves a general class of problems — just as an algorithm is a generalised description of a computation, and a program is a particular implementation of an algorithm. Because patterns are general descriptions, and because they are higher-level than algorithms, you should not expect the implementation to be the same every time they are used. You can't just cut out some sample code describing a pattern in this book, paste it into your program and expect it to work. Rather, you need to understand the general idea of the pattern, and to apply that in the context of the particular problem you face.

How can you trust a pattern? For a pattern to be useful, it must be known to work. To enforce this, we've made sure each pattern follows the so-called Rule of Three: we've found at least three known uses of the solution in practical systems. The more times a pattern has been used, the better, as it then describes a better proven solution. Good patterns are not invented; rather they are identified or 'mined' from existing systems and practices. To produce the patterns in this book we've investigated the design of many successful systems that run on small machines. This book distils the essence of the techniques that seem most responsible for the systems' success.

Forces

A good pattern should be intellectually rigorous. It should present a convincing argument that that its solution actually solves its problem, by explaining the logic that leads from problem to solution. To do this, a good pattern will enumerate all the important *forces* in the context and enumerate the positive and negative consequences of the solution. A force is '*any aspect of the problem that should be considered when solving it'* [Buschmann et al 1996], such as a requirement the solution must meet, a constraint the solution must overcome, or a desirable property that the solution should have.

The most important forces the patterns in this book address are: *memory requirements*, the amount of memory a system occupies; and *memory predictability*, or whether this amount can be determined in advance. These patterns also address many other forces, however, from real-time performance to usability. A good pattern describes both its benefits (the forces it *resolves)*, and its disadvantages (the forces it *exposes).* If you use such a pattern you may have to address the disadvantages by applying another pattern [Meszaros and Doble 1998].

The Appendix discusses the major forces addressed by the patterns in this collection, and describes the main patterns that can resolve or expose each force. There's also a summary in the table printed inside the back cover.

Collections of Patterns

Some patterns may stand alone, describing all you need to do, but many are *compound patterns* [Vlissides 1998; Riehle 1997] and present their solution partially in terms of other patterns. Applying one pattern resolves some forces completely, more forces partially, and also exposes some forces that were not yet considered. Other, usually smaller-scale, patterns can address problems left by the first pattern, resolving forces the first pattern exposes.

Alexander organised his patterns into a *pattern language*, a sequence of patterns from the highest level to the lowest, where each pattern explicitly directed the reader to subsequent patterns. By working through the sequence of patterns in the language, an architect could produce a complete design for a whole room, building, or city [Alexander 1977].

The patterns in this book are not a pattern language in that sense, and we certainly have not set out to describe every programming technique required to build a complete system! Where practical, however, we have described how the patterns are related, and how using one pattern can lead you to consider using other patterns. The most important of these relationships are illustrated in the diagram printed inside the back cover.

A Brief History of Patterns

Patterns did not originate within programming. A Californian architect, Christopher Alexander developed the pattern form as a tool for recording knowledge of successful building practices (architectural folklore) [Alexander 1977, 1979]. Kent Beck and Ward Cunningham adapted patterns to software, writing a few patterns than were used to design user intefaces at Textronix. The 'Hillside Group' of software engineers developed techniques to improve pattern writing, leading to the PLoP series of conferences.

Gamma, Helm, Johnson and Vlissides, developed patterns for object-oriented frameworks in their 1995 book '*Design Patterns*'. Many other valuable pattern books have followed, particularly *Patterns of Software Architecture* [Buschmann 1996], *Analysis Patterns* [Fowler 1997], the Addison-Wesley *Pattern Languages of Program Design* series, and more specialist books, such as the *Smalltalk companion to Design Patterns* [Alpert, Brown and Woolf 1998], etc. You can now find patterns on virtually any aspect of software development using the *Patterns Almanac* [Rising 2000].

How We Wrote This Book

To produce this particular collection of patterns, we built up a comprehensive list of memory saving techniques that we'd seen in software, been told about, seen on the web, or just known about all along. We then pruned out any techniques that seemed related but that didn't actually save memory (such as bank switching or power management), and any techniques outside the scope of this book (UI design, project management). The result was several hundred different ideas, known uses and examples.

We then grouped them together, looking for a number of underlying themes or ideas that provided a set of underlying techniques. Each technique formed the basis for a pattern, and we wrote them up as a draft that was presented at a pattern writers workshop in 1998 [Noble and Weir 1998, Noble and Weir 2000]. As we built up a set of draft patterns, and received comments, criticism and suggestions, we 'refactored' [Fowler 1999] the patterns to give a more complete picture, expanding the scope of one pattern and reducing or changing the thrust of another. We analysed the major forces addressed by each pattern, and the relationships between the patterns, to find a good way to arrange the patterns into a coherent collection. The result is what you're reading now.

Like any story after the fact, this step-by-step approach wasn't exactly what happened in practice. The reality was much more organic and creative; it took place over several years, and most of the steps happened in parallel throughout that time: but in principle it's accurate.

Our Pattern Format

All the patterns in this book use the same format. Consider the example (overleaf/on the facing page), an abbreviated version of one of the data structure patterns. The full pattern is on page XXX.

Packed Data

Also Known As: Bit Packing

How can you reduce the memory needed to store a data structure?

- You have a data structure (a collection of objects) that has significant memory requirements.
- You need fast random access to every part of every object in the structure…

No matter what else you do in your system, sooner or later you end up having to design low-level data structures to hold the information your program needs…

For example, the Strap-It-On's Insanity-Phone application needs to store all of the names and numbers in an entire local telephone directory (200,000 personal subscribers)….

Because these objects (or data structures) are the core of your program, they need to be easily accessible as your program runs…

Therefore: *Pack data items within the structure so that they occupy the minimum space.*

There are two ways to reduce the amount of memory occupied by an object…

Consider each individual field in turn, and consider how much information that field really needs to store. Then, chose the smallest possible language-level data type that can store than information, so that the compiler (or assembler) can encode it in the minimum amount of memory space…

Considering the Insanity-Phone again, the designers realised that local phone books never cover more than 32 area codes – so each entry requires only 5 bits to store the area code…

Consequences

Each instance occupies less memory reducing the total *memory requirements* of the system, even though the same amount of data can be stored, updated, and accessed randomly…

However: The *time performance* of a system suffers, because CPUs are slower at accessing unaligned data…

❖ ❖ ❖

Figure 1: Excerpt from a Pattern

The example shows the sections of the pattern, which are as follows:

Pattern Name	Every pattern has a unique name, which should also be memorable
Cartoon	A cartoon provides an amusing visual representation of the solution.
Also Known As	Any other common names for the pattern, or for variants of the pattern.
Problem Statement	A single sentence summarises the main problem this pattern solves.
Context Summary	Bullet points summarise each main force involved in the problem, giving an at-a-glance answer to 'is this pattern suitable for my particular problem?'

Context Discussion	Longer paragraphs expand on the bullet points: when might this pattern apply; and what makes it a particularly interesting or difficult problem? This section also introduces a simple example problem – often fanciful or absurd – as an illustration.
Solution	**Therefore:** A single sentence summarises the solution.
Solution Description	Further paragraphs describe the solution in detail, sometimes providing illustrations and diagrams. This section also shows how the solution solves the example problem.
Consequences	This section identifies the typical consequences of using the pattern, both advantages and disadvantages. To distinguish the two, the positive consequences are first, and the negative ones second, partitioned by the word '**However:**'. In this discussion, the important forces are shown in *italic*; these forces are cross-referenced in the discussion in Chapter XXX.
Separator	Three '❖' symbols indicate the end of the pattern description.

Throughout the text we refer to other patterns using SMALL CAPITALS: thus 'PACKED DATA.'

This is all you need to read for a basic knowledge of each pattern. For a more detailed understanding – for example if you need to apply the pattern – every pattern also provides much more information, in the following sections:

Implementation	A collection of 'Implementation Notes' discuss the practical details of implementing the pattern in a real system. This is usually the longest section extending into several pages. The instructions in the implementation notes are not obligatory; you may find alternative, and better, ways to implement the pattern in a particular context. The notes capture valuable experience and it's worth reading them carefully before starting an implementation.
Example	This section provides code samples from a particular, usually fairly simple, implementation, together with a detailed discussion of what the code does and why. We recommend reading this section in conjunction with the Implementation section. Our web site [www.smallmemory.com] provides the full source for most of the samples.
Known Uses	This section describes several successful existing systems that use this pattern. This section validates our assertion that the pattern is useful and effective, and it also suggests places where you might look for further information.
See Also	This section points the reader to other related patterns in this book or elsewhere. This section may also refer you to books and web pages with more information on the subject or to help further with implementation.

Major Techniques and Pattern Relationships

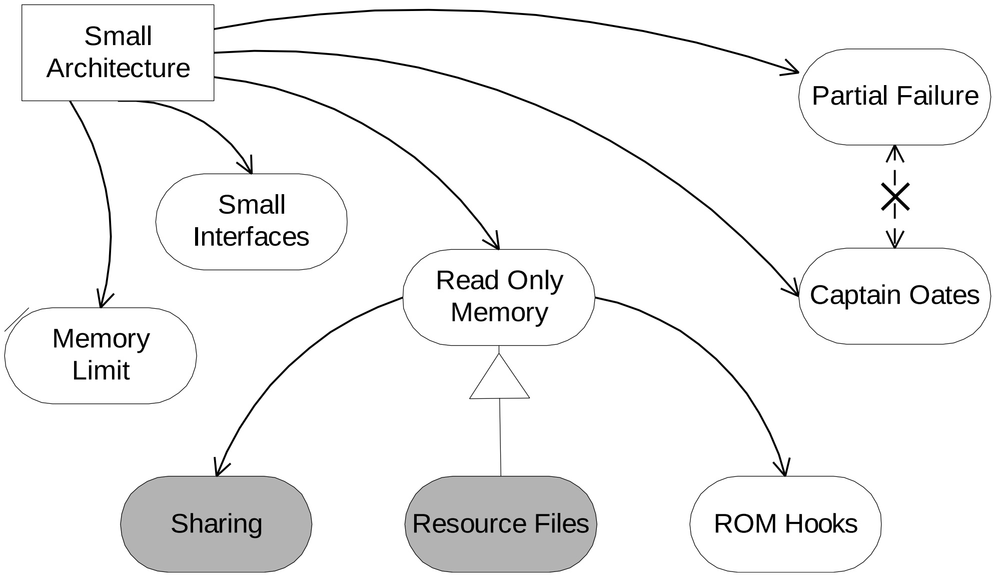
We've grouped the patterns in this book into five chapters. Each chapter presents a Major Technique for designing small memory software: Architecture, Secondary Storage, Compression, Data Structures, and Allocation.

Each Major Technique is, itself, a pattern, though more abstract than the patterns it contains. We've acknowledged that by using a variant of the same pattern format for Major Techniques as for the normal patterns.

In each Major Technique, a 'Specialised Patterns' section summarises each pattern in the chapter, replacing the 'Example' section. (see Figure XXX).

Specialised Patterns

The following sections describe six specialised patterns that describing ways architectural decisions can reduce RAM memory use. The figure below shows how they interrelate…



The patterns in this chapter are as follows:

SMALL INTERFACES Design the interfaces between components to manage memory explicitly, minimising the memory required for their implementation…

Figure 2: Excerpt from a Major Technique

This diagram illustrates the relationships between the patterns in the chapter. In the diagram, the rectangle represents the Major Technique pattern; white ovals represent patterns in the chapter; and grey ovals represent patterns in other chapters. The relationships between the patterns are shown as follows [Noble 1998]:

	Uses	If you're using the left-hand pattern, you should also consider using the right-hand one. The smaller-scale pattern on the right resolves *forces* exposed by the larger pattern on the left. For example if you are using the READ-ONLY MEMORY pattern, you should consider use the HOOKS pattern.

◁———	*Specialises*	If you are using the left hand pattern, you may want the right-hand pattern in particular situations. The right-hand pattern is a more specialised version of the left-hand one, resolving similar *forces* in more particular ways. For example, RESOURCE FILES are a special kind of READ-ONLY MEMORY.
<– –X– –>	*Conflicts*	Both patterns provide alternative solutions to the same problem. They resolve similar *forces* in incompatible ways. For example, PARTIAL FAILURE and CAPTAIN OATES both describe how a component can deal with memory exhaustion; either by providing a reduced quality of service, or by terminating another component.

The Running Example

We've illustrated many of the patterns with examples taken from a particularly memory-challenged system, the unique Strap-It-On^{TM} wrist-mounted PC from the well-known company StrapItOn. This product, of course, includes the famous Word-O-Matic word-processor, with its innovative Morse Code keypad and Voice User Interface (VUI).

Figure 3: The Strap-it-on

If you're foolish enough to implement any of the applications we suggest, and make money out of it, well good luck to you!

The Patterns in this Book

We've chosen one particular order for the patterns in this book (see the table in the front inside cover). This order makes the patterns easy to learn, working top-down so that the first patterns set the scenes for the patterns in later chapters.

There are many other valid ways to arrange or discuss the list patterns. This section examines from several different perspectives: the list of Major Techniques, the forces addressed by each technique, different approaches to saving memory, and via case studies. You may also like to consult the discussion '**Error! Reference source not found.**' on page XXX, which recommends patterns according to the types of small software involved.

The Major Techniques

The patterns are organised into five Major Techniques, summarised in the following table:

Small Data Structures	Defining data structures and algorithms that contrive to reduce memory use.
Memory Allocation	Mechanisms to assign a data structure from the 'primordial soup' of unstructured available memory, and to return it again when no longer required by the program.
Compression	Processing-based techniques to reduce data sizes by automatically compressing data.
Secondary Storage	Using disk, or equivalent, as an adjunct to RAM.
Small Architecture	Memory-saving techniques that require co-operation between several components in a system.

The Forces Addressed by the Patterns

Another way to look at the patterns is to compare the forces addressed by each pattern.

Chapter XXX (Forces) discusses the forces in detail, and discusses which patterns address each force. Meanwhile the following two tables provide a partial summary of that chapter. Table XXX summarises ten of the forces we consider most important, and table XXX shows how the patterns address each one.

Memory Requirements	Does the pattern reduce the absolute amount of memory required to run the system?
Memory Predictability	Does the pattern make it easier to predict the amount of memory a system will require in advance?
Real-time Response	Does the pattern decrease the latency of the program's response to events, usually by making the run-time performance of the program predictable?
Start-up Time	Does the pattern reduce the time between the system receiving a request to start the program, and the program beginning to run?
Time Performance	Does the pattern tend to improve the run-time speed of the system?
Local vs. Global	Does the pattern tend to help encapsulate different parts of the application, keeping them more independent of each other?
Secondary	Does the pattern tend to shift memory use towards cheaper secondary storage in

Storage	preference to more expensive RAM?
Maintainability	Does the pattern encourage better design quality? Will it be easier to make changes to the system later on?
Programmer Effort	Does the pattern reduce the total programmer effort to produce a given system?
Testing cost	Does the pattern reduce the total testing effort for the application development?

Table 2: Ten Important Forces

The following table shows how each pattern addresses these forces. If the pattern generally benefits you as far as this force is concerned ('resolves the force'), it's shown with a 'Y'. If it's generally a disadvantage ('exposes the force') that's shown with an 'N'. Appendix XXX explores these forces in much more detail.

[Typesetter to replace N in table and paragraph with sad face ☹ , and make those cells white font on black background.

Typesetter to replace Y in table and paragraph with happy face ☺, black font on white background.

Typesetter to replace O in table and paragraph with face 😐 - black font on light grey background

This table is duplicated in the inside back cover.]

		Time Performance	Real-time	Start-up time	Local vs. Global	Predictability	Quality and Maintainability	Programmer Effort	Testing cost	Usability
Architecture					O	Y	Y	O		Y
	Memory Limit				Y	Y			O	
	Small Interfaces	N	Y		Y	Y	Y	O	O	
	Partial Failure				N	Y	Y	N	N	Y
	Captain Oates	N			Y	O		N	N	Y
	Read-only Memory		Y	Y	O		N	N	Y	
	Hooks	N					Y	Y	N	
Secondary Storage			N		N			N		N
	Application Switching	N	Y	Y	O	Y	Y	O	Y	N
	Data Files	N		N	O	Y		N	Y	N
	Resource Files	N		N			Y	O		
	Packages		N	Y			Y	N	N	O
	Paging	N	N		Y		Y	Y	Y	Y
Compression		N	O			N	N	N	N	
	Table Compression	O							N	
	Sequence Compression	O	Y			Y			N	
	Adaptive Compression	N	N					N	N	
Data Structures			O		Y	O	Y	N	O	Y
	Packed Data	N			Y		N	N		
	Sharing	Y		Y	N		O	N	N	
	Copy-on-Write	O	N	Y		N		N	N	
	Embedded Pointer	Y	Y		N	Y	N	N		
	Multiple Representations	Y			Y	N	Y	O	N	

		Time Performance	Real-time	Start-up time	Local vs. Global	Predictability	Quality and Maintainability	Programmer Effort	Testing cost	Usability
Allocation		O	O	O	O	Y				
	Fixed Allocation	Y	Y	N		Y		O	Y	N
	Variable Allocation	N	N	Y	N	N	Y	Y	N	
	Memory Discard	Y	Y	Y	Y	Y		Y	N	
	Pooled Allocation	Y	Y	N		Y			N	
	Compaction	N	N					N	N	
	Reference Counting	N	Y		Y		Y			
	Garbage Collection	Y	N		Y	N	Y	Y		

Reduce Reuse Recycle

Environmentalists have identified three strategies to reduce the impact of human civilisation on the natural environment:

- Reduce consumption of manufactured products and the production of waste products.
- Reuse products for uses other than that for which they were intended.
- Recycle the raw material of products to make other products.

Of these, reduction is the most effective; if you reduce the amount of waste produced you don't have to worry about how to handle it. Recycling is the least effective; it requires a large expenditure of energy and effort to produce new finished products from old waste. The patterns we have described can be grouped in a similar way. They can:

- **Reduce** a program's memory requirements. Patterns such as PACKED DATA, SHARING, and COMPRESSION reduce the amount of absolute memory required, by reducing data sizes and removing redundancy. In addition the SECONDARY STORAGE and READ ONLY MEMORY patterns reduce RAM memory requirements by using alternative storage.
- **Reuse** memory for a different purpose. Memory used within a FIXED ALLOCATION or in POOLED ALLOCATION is generally (re)used to store a number of different objects of roughly the same type, one after another. HOOKS allow software to reuse existing read-only code rather than replacing it.
- **Recycle** memory for different uses at different types. VARIABLE ALLOCATION, REFERENCE COUNTING, COMPACTION, GARBAGE COLLECTION and CAPTAIN OATES all help a program to make vastly different uses of the same memory over time.

Case Studies

This section looks at three simple case studies, and looks at the patterns you might use to deliver a successful implementation in each case.

1. Hand-held Application

Consider working on an application for a hand-held device such as Windows CE, PalmOs or an EPOC smart-phone.

The application will have a GUI, and will need much of the basic functionality you'd expect of a full-scale PC application. Memory is, however, more limited than for a PC; acceptable maximums might be 2Mb of working set in RAM, 700Kb of (non-library) code, and a couple of Mb or so of persistent data according to the needs of each particular user.

Memory is limited for all applications on the device, including your own. The SMALL INTERFACES architectural pattern is ubiquitous and vital. Applications will use RESOURCE FILES, as dictated by the operating system style guide, and to keep code sizes and testing to a minimum you'll want to access the vendor-supplied libraries in READ ONLY MEMORY using the libraries' HOOKS. Since the environment supports many processes and yours won't be running all the time, you'll need persistent data stored in DATA FILES.

The environment may mandate other architectural patterns: PalmOs requires APPLICATION SWITCHING; CE expects CAPTAIN OATES; and EPOC expects PARTIAL FAILURE. If you're not working for the system vendor then yours will be a 'third party' application loaded from secondary storage, so you may use PACKAGES to reduce the code working set.

Most of your application's objects will use VARIABLE ALLOCATION or MEMORY DISCARD. Components where real-time performance is important –a communications driver, for example – will use FIXED ALLOCATIONand EMBEDDED POINTERS.

Classes that have many instances may use PACKED DATA, MULTIPLE REPRESENTATIONS, SHARING or COPY-ON-WRITE to reduce their total memory footprint. Objects shared by several components may use REFERENCE COUNTING.

2. Smart-card Project

Alternatively, consider working on a project to produce the software for a smart-card – say a PCMCIA card modem to fit in a PC. Code will live in ROM (actually flash RAM used as ROM), and there's about 2Mb of ROM in total, however there's only some 500K of RAM. The only user interface is the Hayes 'AT' command set available via the serial link to the PC; the modem is also connected to a phone cable.

The system code will be stored in the READ-ONLY MEMORY, along with static tables required by the modem protocols. You'll only need a single thread of control and a single, say 50K, stack.

The real-time performance of a modem is paramount, so most long lived objects will use FIXED ALLOCATION. Transient data wil use MEMORY DISCARD, being stored on the stack. The system will need lots of buffers for input and output, and these can use POOLED ALLOCATION; you may also need REFERENCE COUNTING if the buffers are shared between components.

Much of the main processing of the modem is COMPRESSION of the data sent on the phone line. Simple modem protocols may use SEQUENCE COMPRESSION; more complicated protocols will use TABLE COMPRESSION and ADAPTIVE COMPRESSION. To implement these more complicated protocols you'll require large and complicated data structures built up in RAM. To minimise the memory they use and improve performance, you can implement them with PACKED DATA and EMBEDDED POINTERS.

3. Large Web Server Project

Finally you might be working on a Java web server, which will provide an E-commerce Web and WAP interface to allow users to buy products or services. The server will connect to internal computers managing the pricing and stock control and to a database containing the details of individual users, via a local-area network.

RAM memory is relatively cheap compared with development costs, but there are physical limits to the amounts of RAM a server can support. The sever's operating system provides PAGING to increase the apparent memory available to the applications, but since most transactions take a relatively short time you won't want to have much of the memory paged out at any time. There will be many thousands of simultaneous users, so you can't afford simply to assign dozens of megabytes to each one.

You can ues SHARING so that all the users share just one, or perhaps just a few Java virtual machine instances. Where possible data will be READ ONLY, to make it easy to share. If the transaction with each user can involve arbitrarily complex data structures you can enforce a MEMORY LIMIT for each user. Maintainability and ease of programming are important so virtually all objects use VARIABLE ALLOCATION and GARBAGE COLLECTION.

The Internet connections to each user are a significant bottleneck, so you'll use ADAPTIVE COMPRESSION to send out the data, wherever the Web or WAP protocols support it. Finally, you may need to support different languages and page layouts for different users via RESOURCE FILES.

Major Technique: Small Architecture

How can you manage memory use across a whole system?

- Memory limitations restrict entire systems
- Systems are made up of many components
- Each component can be fabricated by a different team.
- Components' memory requirements can change dynamically.

A system's memory consumption is a global concern. Working well in limited memory isn't a feature that you can incorporate into your program in isolation: you can't ask a separate team of programmers to add code your system hoping to reduce its memory requirements. Rather, memory constraints cross-cut the design of your system, affecting every part of it. This is why designing software for systems for limited memory is difficult. [Buschmann, Meunier, Rohnert, Sommerlad, and Stal, 1996; Shaw and Garlan 1996; Bass, Paul, and Kazman 1998; Bosch 2000].

For example, the Strap-It-On wrist-top PC's has an email application supporting text in a variety of fonts. Unfortunately in early implementations it cached every font it ever loaded, to improve performance there; but stored every email compressed, threw away attachments and crashed if memory ran out loading a font, giving poor new performance and awful usability. There's no sense in one function limiting its memory use to a few hundred bytes when another part of the program wastes megabytes, and then brings the system down when it fails to receive them.

You could simply design your system as a monolithic single component: a "big ball of mud" [Foote and Yoder 2000]. Tempting though this approach might be, it tends to be unsatisfactory for any but the simplest systems, for several reasons: it's difficult to split the development of such a system between different programmers, or different programming teams; the resulting system will be difficult to understand and maintain, since every part of the system can affect every other part; and you loose any possibility of buying in existing reusable components.

To keep control over your system, you can construct it from components that you can design, build, and test independently. Components can be reused from earlier systems, purchased from external suppliers, or built new; some may need specialised skills to develop; some may even be commercially viable in their own right. Each component can be assigned to a single team, to avoid several teams working on the same code [Szyperski 1999].

These components may be of many different kinds, and interact in many different ways: source libraries to compile into the system; object libraries that must be compiled into an executable; dynamic-linked libraries to load at run-time; run-time objects in separate address-spaces using frameworks like CORBA, Java Beans or ActiveX; or simply separate executables running in their own independent process. All are logically separate components, and communicate, if they communicate at all, through interfaces.

Unfortunately, separating a program into components doesn't reduce its memory use. The whole systems' memory requirements will be the sum of the memory required by each component. Furthermore, the memory requirements for each component, and so for the whole system, will change dynamically as the system runs. Even though memory consumption affects the architecture globally, it is still important that components can be treated separately as much as possible. How can you make the system use memory effectively, and give the best service to its users, in a system is divided into components?

Therefore: *Make every component responsible for its own memory use.*

A system's architecture is more than just the design of its high-level components and their interconnections: it also defines the system's *architectural strategies* — the policies, standards and assumptions common to every component [Bass et al 1998; Brooks 1982]. The architecture for a system for limited memory must describe policies for memory management and ensure that each component's allocations are feasible in the context of the system as a whole.

In a system for limited memory, this means that each individual component must take explicit responsibility for implementing this policy: for managing its own memory use. In particular, you should take care to design **SMALL DATA STRUCTURES** that require the minimum memory to store the information your system needs.

Taking responsibility for memory is quite easy where a component allocates memory statically (**FIXED ALLOCATION**); a component simply owns all the memory that is fixed inside it. Where a component allocates memory dynamically from a heap (**VARIABLE ALLOCATION**) it is more difficult to assign responsibility; the heap is a global resource. A good start is to aim to make every dynamically allocated object or record be owned by one component at all times. [Cargill 1996]. You may need to implement a **MEMORY LIMIT** or allocate objects using **POOLED ALLOCATION** for a component to control its dynamic memory allocation. Where components exchange objects, you can use **SMALL INTERFACES** to ensure some component always takes responsibility for the memory required for the exchange.

A system architecture also needs to set policies for mediating between components' competing memory demands, especially when there is little or no unallocated memory. You should ensure that components suffer only **PARTIAL FAILURE** when their memory demands cannot be met, perhaps by sacrificing memory from low priority components (**CAPTAIN OATES**) so that the system can continue to operate until more memory becomes available.

For example, the software architecture for the Strap-It-On PC defines the Font Manager and the Email Display as separate components. The software architecture also defines a memory budget, constraining reasonable memory use for each component. The designers of the Font Manager implemented a **MEMORY LIMIT** to reduce their font cache to a reasonable size, and the designers of the Email Display component discovered they could get much better performance and functionality than they had thought. When the Email application displays a large email, it uses the **SMALL INTERFACE** of the Font Manager to reduce the size of the Font Cache. Similarly, when the system is running short of memory the font cache discards any unused items (**CAPTAIN OATES**).

Consequences

Handling memory issues explicitly in a program's architecture can reduce the program's *memory requirements,* increase the *predictability* of its memory use, and may make the program more *scalable* and more *usable.*

A consistent approach to handling memory reduces the *programmer effort* required since the memory policies do not have to be re-determined for each component. Individual modules and teams can co-ordinate smoothly to provide a consistent *global* effect, so users can anticipate the final system's behaviour when memory is low, increasing *usability.*

In general, explicitly describing a system's architecture increases its *design quality* improving *maintainability*.

However: designing a small architecture takes *programmer effort*, and then ensuring components are designed according to the architecture's rules takes *programmer discipline.* Making memory an architectural concern moves it from being a *local* issue for individual components and teams to a *global* concern, involving the whole project. For example, developers may try to minimise their components memory requirements at the expense of other components produced by other teams.

Incorporating external components can require large amounts *programmer effort* if they do not meet the standards set by the system architecture — you may have to re-implement components that cannot be adapted.

Designing an architecture to suit limited memory situations can restrict a program's *scalability* by imposing unnecessary restrictions should more memory become available.

❖ ❖ ❖

Implementation

The main ideas behind this pattern are 'consistency' and 'responsibility'. By splitting up your system into separate components you can design and build the system piece by piece; by having a common memory policy you ensure that the resulting pieces work together effectively.

The actual programming mechanism used to represent components is not particularly important. A component may be a class, a package or a namespace, a separate executable or operating system process, a component provided by middleware like COM or CORBA, or an ad-hoc collection of objects, data structures, functions and procedures. In an object-oriented system, a component will generally contain many different objects, often instances of different classes, with one or more objects acting as **FACADES** to provide an interface to the whole component.

Here are two further issues to consider when designing interfaces for components in small systems:

1. Tailorability.

Different clients vary in the memory requirements they place on other components that they use. This is especially the case for components that are designed to be reusable; such components will be used in many different contexts, and those contexts may have quite different memory requirements.

A component can address this by including parameters to tailor its memory use in its interface. Clients can adjust these parameters to adapt the component to fit its context. Components using **FIXED ALLOCATION**, for example, have to provide creation-time parameters to choose the number of items they can store. Similarly, components using **VARIABLE ALLOCATION** can provide parameters to tune their memory use, such as maximum capacity, initial allocation, or even the amount of free space (in a hash table, for example, leaving free space can increase lookup performance). Components can also support operations to control their behaviour directly, such as requesting a database to compact itself, or a cache to empty itself.

For example, the Java vector class has several methods that control its memory use. Vectors can be created with sufficient memory to hold a given number of items (say 10):

```
Vector v = new Vector(10);
```

This capacity can be increased dynamically (say to store twenty items):

```
v.ensureCapacity(20);
```

The capacity can also be reduced to provide only enough memory for the number of elements in the container, in this case one object.

```
v.addElement( new Object() );
v.trimToSize();
```

Allocating correctly sized structures can save a surprisingly large amount of memory and reduce the load a component places on a low level memory allocator or garbage collector. For example, imagine a Vector that will be used to store 520 items inserted one at a time. The vector class initially allocates enough space for 8 elements; when that is exhausted, allocates twice as much space as it is currently using, copies its current elements into the new space, and deallocates the old space. To store 520 elements, the vector will resize itself seven times, finally allocating almost twice the required memory, and having allocated about four times as much memory in total. In contrast, initialising the vector with 520 elements would have required one call to the memory system and allocated only as much memory as required [Soukup 1994].

2. Make clients responsible for components' memory allocation.

Sometimes a component needs to support several radically different policies for allocating memory — some clients might want to use **POOLED ALLOCATION** for each object allocated dynamically within the package; others might prefer a **MEMORY LIMIT** or to use **MEMORY DISCARD**; and still others might want the simplicity of allocating objects directly from the system heap. How can you cater for all of these with a single implementation of the component?

2.1. Callbacks to manage memory. A simple approach is to require the component to call memory management functions provided by the client. In non-OO environments, for example, you can make the component call a function supplied by its client, and linking the client and component together. In C, you might use function pointers, or make the component declare a function prototypes to be implemented by the library environment. For example, the Xt Window System Toolkit for the X Window System supports a callback function, the `XAlloc` function hook; clients may provide a function to do the memory allocation (and of course, another function to do the memory freeing) [Gilly and O'Reilly 1990].

2.2. Memory Strategy. In an object-oriented environment, you can apply the **STRATEGY** pattern: define an interface to a family of allocation algorithms, and then supply the component with the algorithm appropriate for the context of use. For example, in C++, a strategy class can simple provide operations to allocate and free memory:

```
class MemoryStrategy {
    virtual char* Alloc( size_t nBytes ) = 0; // returns null when exhausted.
    virtual void Free( char* anItem; ) = 0;
};
```

Particular implementations of the `MemoryStrategy` class then implement a particular strategy: a `PooledStrategy` implements **POOLED ALLOCATION**, a `LimitStrategy` applies a **MEMORY LIMIT**; a `TemporaryHeapStrategy` implements **MEMORY DISCARD**; and a `HeapStrategy` simply delegates the `Alloc` and `Free` operations straight to the system `malloc` and `free` functions.

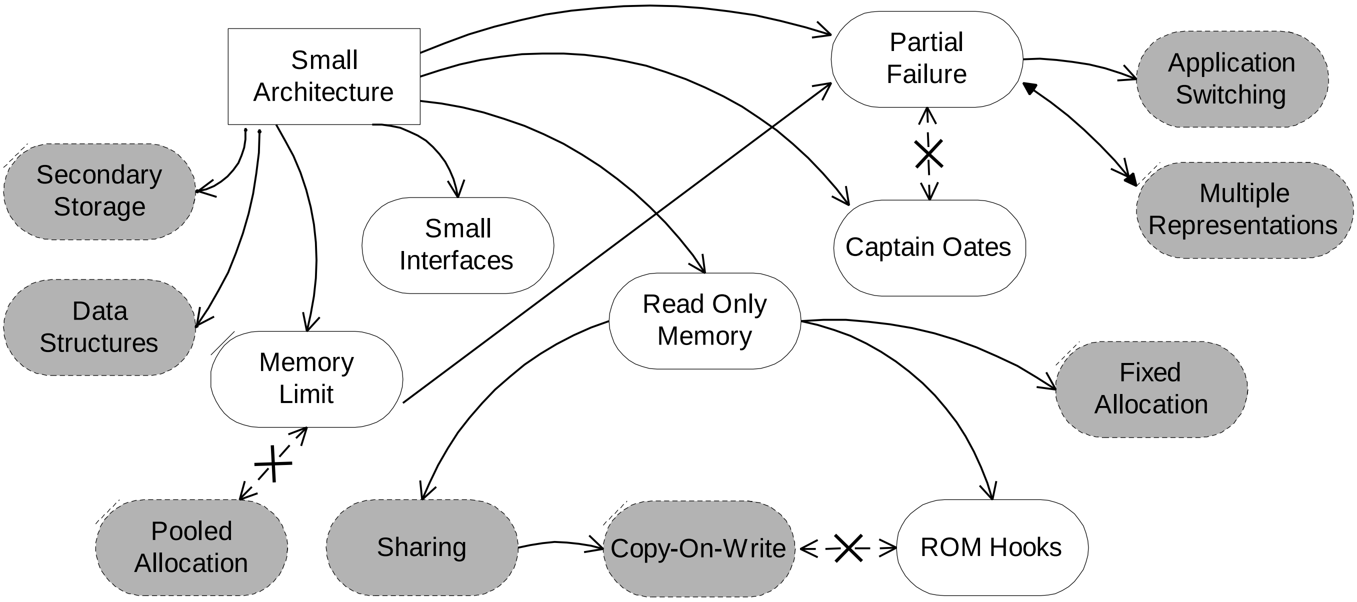
An alternative C++ design uses compile-time template parameters rather than runtime objects. The C++ STL collection and string templates accept a class parameter (called `Allocator`) that provides allocation and freeing functions [Stroustrup 1997]. They also provide a default implementation of the `allocator` that uses normal heap operations. So the definition of the STL 'set' template class is:

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator> class set;
```

Note how the `Allocator` template parameter defaults to `allocator`, the strategy class that uses normal heap allocation.

Specialised Patterns

The following sections describe six specialised patterns that describing ways architectural decisions can reduce RAM memory use. The figure below shows how they interrelate. Two other patterns in this book are closely related to the patterns in this chapter, and these patterns are shown in grey.



This chapter contains the following patterns:

MEMORY LIMIT enforces a fixed upper bound on the amount of memory a component can allocate.

SMALL INTERFACES between components are designed to manage memory explicitly, minimising the memory required for their implementation.

PARTIAL FAILURE ensures a component can continue in a 'degraded mode', without stopping its process or losing existing data, when it cannot allocate memory.

CAPTAIN OATES improves the overall performance of a system, by surrendering memory used by less important components when the system is running low on memory.

READ-ONLY MEMORY can be used to store components that do not need to be modified, in preference to more constrained and expensive main memory.

HOOKS allow information stored in **READ-ONLY MEMORY** (or shared between components) to appear to be changed

Known Uses

Object-oriented APIs designed to support different memory strategies include the C++ standard template library [Stroustrup 1997] and the Booch components [Booch 1987]. These libraries, and to a lesser extent the standard Java and Smalltalk collection classes, also provide parameters that adjust components' strategies, for example, by preallocating the correct amount of memory to hold an entire structure.

See Also

Many small architectures take advantage of **SECONDARY STORAGE** to reduce requirements for main memory. Architectures can also design **SMALL DATA STRUCTURES** to minimise their memory use, and encourage **SHARING** of code and data between components.

Tom Cargill's patterns for *Localized Ownership* [Cargill 1996] describe how you can ensure every object is the responsibility of precisely one component at all times. The **HYPOTH-A-SIZED COLLECTION** pattern [Auer and Beck 1996] describes how collections should be created with sufficient capacity to meet their clients needs without extra allocations.

Software Requirements & Specifications [Jackson 1995] and *Software Architecture* [Shaw and Garlan 1996] describe ways to keep a coherent architecture while dividing an entire system into components. *Software Architecture in Practice* [Bass et al 1998] describes much about software architecture; *Design and Use of Software Architectures* [Bosch 2000] is a newer book that focuses in particular on producing product-lines of similar software systems. *Patterns in Software Architecture* has a number of architecture-level patterns to help design whole systems and is well worth reading [Buschmann et al 1996].

The *Practice of Programming* [Kernighan and Pike, 1999], the *Pragmatic Programmer* [Hunt and Thomas 2000] and the *High-Level and Process Patterns from the Memory Preservation Society* [Noble and Weir 2000] describe techniques for estimating the memory consumption of a system's components, and managing those estimates throughout a development project.

Memory Limit

Also Known As: Fixed-sized Heap, Memory Partitions

How can you share out memory between multiple competing components?

- Your system contains many components, each with its own demands on memory.
- Components' memory requirements change dynamically as the program runs.
- If one component hogs too much memory, it will prevent others from functioning.
- You can define reasonable upper bounds on the memory required for each task.

As part of designing a **SMALL ARCHITECTURE,** you will have divided up your system into architectural components, and made each component responsible for its own memory use. Each components' memory demands will change as the program runs, depending on the overall kind of load being placed on the system. If access to memory is unrestricted, then each component will try to allocate as much memory as it might need, irrespective of the needs of other components. As other components also allocate memory to tackle their work, the system as a whole may end up running out of memory.

For example, the Strap-It-On's Virtual Reality "Stair Wars 1" game has several components: virtual reality display, voice output, music overview, voice recognition, not to mention the artificial intelligence brain co-ordinating the entire game plan. Each of these tasks is capable of using as much memory as it receives, but if every component tries to allocate a large amount of memory there will not be enough to go round. You must apportion the available memory sensibly between each component.

You could consider implementing the Captain Oates pattern, allowing components low on memory to steal it from components with abundant allocations. Captain Oates relies on the goodwill of component programmers to release memory, however, and can be difficult and complex to implement.

You could also consider budgeting components' memory use in advance. Just planning memory consumption is also insufficient; however, unless there is some way to be sure that components will obey the plans. This is trivial for components that use **FIXED ALLOCATION** exclusively, but for others it can be difficult to model their dynamic behaviour to be sure they will not disrupt your plans.

Therefore: *Set limits for each component and fail allocations that exceed the limits.*

There are three steps to applying the memory limit pattern.

1. Keep an account of the memory currently allocated by each component. For example, you might modify a component's memory allocation routine to increase a memory counter when allocating memory, and decrease the counter when deallocating memory.
2. Ensure components cannot allocate more memory then an allotted limit. Allocation operations that would make a component exceed its limit should fail in exactly the same way that they would fail if there were no more memory available in the system. Components should support **PARTIAL FAILURE** so that they can continue running even when they are at the limit.
3. Set the limits for each component, ideally by experimenting with the program and examining the memory use counters for each component. Setting the limits last may seem to be doing things backwards, but, in practice, you will have to revise limits during

development, or alternatively allow users to adjust them to suit their work. So, build the accounting mechanisms first, experiment gathering usage information, and then set the memory use policies that you want enforced.

Should the sum of the limits for each component be equal or greater than the total available? The answer depends on whether all the tasks are likely to be using their maximum memory limit simultaneously. This is unlikely in practice, and the main purpose of the Memory Limit pattern is to prevent a single component from hogging all the memory. It is generally sufficient to ensure that the limit on each task is a reasonable fraction of the total memory available.

Note that it's only worth implementing a limit for components that make variable demands on memory. A memory limit provides little benefits for components where most data structures use **FIXED ALLOCATION** and the memory use doesn't vary significantly with time.

In the 'Stair Wars' program, for example, the artificial intelligence brain component uses memory roughly in proportion to the number of hostile and friendly entities supported. By experimenting with the game, the developers determined a maximum number of such entities, and then adjusted brain component's memory limit to provide enough memory to support the maximum. On the other hand, the screen display component allocates a fixed amount of memory, so Stair Wars doesn't apply an extra memory limit for this component.

Consequences

Because there are guaranteed limits on the memory use of each component, you can *test* each one separately, while remaining sure that it will continue to work the same way in the final system. This increases the *predictability* of the system.

By examining the values of the memory counters, it's easy to identify problem areas, and to see which components are failing due to insufficient memory at run-time, increasing the *localisation* of the system.

Implementing a simple memory counter takes only a small amount of *programmer effort.*

However: Some tasks may fail due to lack of memory while others are still continuing normally; if the tasks interact significantly, this may lead to unusual error situations which are difficult to reproduce and *test.* A component can fail because it's reached its memory limit even when there is plenty of memory in the system; thus the pattern can be wasteful of memory. Most simple memory counters mechanisms don't account for extra wastage due to *fragmentation* (see the **MEMORY ALLOCATION** chapter). On the other hand, more complex operating system mechanisms such as separate heaps for each component tend to increase this same *fragmentation* wastage.

❖ ❖ ❖

Implementation

There are several alternative approaches to implementing memory limits.

1. Intercepting Memory Management Operations.

In many programming languages, you can intercept all operations that allocate and release memory, and modify them to track the amount of memory currently allocated quite simply. When the count reaches the limit, further memory allocations can fail until deallocations return the count below the limit. In C++, for example, you can limit the total memory for a process by overriding the four global `new` and `delete` operators [Stroustrup 1995].

A memory counter doesn't need to be particularly accurate for this pattern to work. It can be sufficient to implement a count only of the major memory allocations: large buffers, for example. If smaller items of allocated memory are allocated in proportion to these larger items, then this limit indirectly governs the total memory used by the task. For example, the different entities in the Stair Wars program each use varying amounts of memory, but the overall memory use is roughly proportional to the total number of entities, so limiting them implemented an effective memory limit.

In C++ you can implement a more localised memory limit by overriding the `new` and `delete` operators for a single class – and thus for its derived classes. This approach also has the advantage that different parts of the same program can have different memory limits, even when memory is allocated from a single heap [Stroupstrup 1997].

2. Separate Heaps.

You can make each component use a separate memory heap, and manage each heap separately, restricting their maximum size. Many operating systems provide support for separate heaps (notable Windows and Windows CE) [Microsoft 97, Boling 1998].

3. Separate processes.

You can make each component an individual process, and use operating system or virtual machine mechanisms to limit each component's memory use. EPOC, and most versions of Unix allow you to specify a memory limit for each process, and the system prevents processes from exceeding these limits. Using these limits requires little *programmer effort*, especially the operating systems also provides tools that can monitor processes memory use so that you can determine appropriate limits for each process. Of course, you have to design or whole system so that separate components can be separate processes — depending on your system, this can be trivial or very complex.

Many operating systems implement heap limits using virtual memory. They allocate the full size heap in the virtual memory address space (see the **PAGING PATTERN**); the memory manager maps this to real memory only when the process chooses to access each memory block. Thus the heap sized is fixed in virtual memory, but until it is used there's no real memory cost at all. The disadvantage of this approach is that very few virtual memory systems can detect free memory in the heap and restore the unused blocks to the system. So in most VM systems a process that uses its full heap will keep the entire heap allocated from then on.

Examples

The following C++ code restricts the total memory used by a `MemoryResrictedClass` and its subclasses. Exceeding the limit triggers the standard C++ out of memory exception, `bad_alloc`. Here the total limit is specified at compile time, as `LIMIT_IN_BYTES`:

```
class MemoryRestrictedClass {
public:
    enum { LIMIT_IN_BYTES = 10000 };
    static size_t totalMemoryCount;

    void* operator new ( size_t aSize );
    void operator delete( void* anItem, size_t aSize );
};

size_t MemoryRestrictedClass::totalMemoryCount = 0;
```

The class must implement an `operator new` that checks the limit and throws an exception:

```
void* MemoryRestrictedClass::operator new ( size_t aSize ) {
    if ( totalMemoryCount + aSize > LIMIT_IN_BYTES )
        throw ( bad_alloc() );
    totalMemoryCount += aSize;
    return malloc( aSize );
}
```

And of course the corresponding `delete` operator must reduce the memory count again:

```
void MemoryRestrictedClass::operator delete( void* anItem, size_t aSize ) {
    totalMemoryCount -= aSize;
    free( (char*)anItem );
}
```

For a complete implementation we'd also need similar implementations for the array versions of the operators [Stroustrup 1995].

In contrast, Java does not provides allocation and deletion operations in the language. It is possible however to limit the number of instances of a given class by keeping a static count of the number of instances created. Java has no simple deallocation call, but we can use finalisation to intercept deallocation. Note that many Java virtual machines do not implement finalisation efficiently (if at all), so you should consider this code as an example of one possible approach, rather than as recommended good practice [Gosling et al 1996].

The following class permits only a limited number of instances. The class counts the number of its instances, increasing the count when a new object is constructed, and decreasing the count when it is finalized by the garbage collector. Now, since objects can only be finalized when the garbage collector runs, at any given time there may be some garbage objects that have not yet been finalised. To ensure we don't fail allocation unnecessarily, the constructor does an explicit garbage collection before throwing an exception if we are close to the limit.

```
class RestrictedClass
{
    static final int maxNumberOfInstances = 5;
    static int numberOfInstances = 0;

    public RestrictedClass() {
        numberOfInstances++;
        if (numberOfInstances > maxNumberOfInstances) {
            System.gc();
        }
        if (numberOfInstances > maxNumberOfInstances) {
            throw new OutOfMemoryError("RestrictedClass can only have " +
                                       maxNumberOfInstances + " instances");
        }
    }
```

There's a slight issue with checking for memory in the constructor: even if we throw an exception, the object is still created. This is not a problem in general, because the object will eventually be finalized unless one of the superclass constructors stores a reference to the object.

The actual finalisation code is trivial:

```
    public void finalize() {
        --numberOfInstances;
    }
};
```

❖ ❖ ❖

Known Uses

By default, UNIX operating systems put a memory limit on each user process [Card et al 1998]. This limit prevents any one process from hogging all the system memory as only processes with system privileges can override this limit. The most common reason for a process to reach the limit is a continuous memory leak: after a process has run for a long time a memory request will fail, and the process will terminate and be restarted.

EPOC associates a heap with each thread, and defines a maximum size or each heap. There is a default, very large, limit for applications, but server threads (daemons) are typically created with rather smaller limits using an overloaded version of the thread creation function `RRhread::Create` [Symbian 1999]. The EPOC culture places great importance on avoiding memory leaks, so the limit serves to limit the resources used by a particular part of the system. EPOC servers are often invisible to users of the system, so it is important to prevent them from growing too large. If a server does reach the memory limit it will do a **PARTIAL FAILURE**, abandoning the particular request or client session that discovered the problem rather than crashing the whole server [Tasker et al 2000].

Microsoft Windows CE and Acorn Archimedes RICS OS allow users to adjust the memory limits of system components at runtime. Windows CE imposes a limit on the amount of memory used for programs, as against data, and RISC OS imposes individual limits on every component of the operating system [Boling 1998, RISC OS 2000].

Java virtual machines typically provide run-time flags to limit the total heap size, so you can restrict the size of a Java process [Lindholm and Yellin 1999]. The Real-Time Specification for Java will support limits on the allocation of memory within the heap [Bollella et al 2000].

See Also

Since it's reasonably likely a typical process will reach the limit, it's better to suffer a **PARTIAL FAILURE** rather than failing the whole process. Using only **FIXED ALLOCATION** (or **POOLED ALLOCATION**) is a simpler, but less flexible, technique to apportion memory among competing components.

Small Interfaces

How can you reduce the memory overheads of component interfaces?

- You are designing a **SMALL ARCHITECTURE** where every component takes responsibility for its own memory use.
- Your system has several components, which communicate via explicit interfaces.
- Interface designs can force components or their clients to allocate extra memory, solely for inter-component communication.
- Reusable components require generic interfaces, which risk needing more memory than would be necessary for a specific example.

You are designing a **SMALL ARCHITECTURE,** and have divided your system into components with each component responsible for its own memory use. The components collaborate via their interfaces. Unfortunately the interfaces themselves require temporary memory to store arguments and results. Sending a large amount of information between components can require a correspondingly large amount of memory.

For example, the Strap-It-On 'Spookivity' ghost hunter's support application uses a compressed database in ROM with details of every known ghost matching given specifications. Early versions of the database component were designed for much smaller RAM databases, so they implemented a 'search' operation that simply returned a variable sized array of structures containing copies of full details of all the matching ghosts. Though functionally correct, this interface design meant that Spookivity required a temporary memory allocation of several Mbytes to answer common queries – such as "find ghosts that are transparent, whitish, floating and dead"- an amount of memory simply not available on the Strap-It-On.

Interfaces can also cause problems for a **SMALL ARCHITECTURE** by removing the control each component has over memory allocation. If an object is allocated in one component, used by another and finally deleted by a third, then no single component can be responsible for the memory occupied. In the Spookivity application, although the array of ghost structures was allocated by the database component it somehow became the responsibility of the client.

Reusable components can make it even more difficult to control memory use. The designer of a reusable component often faces questions about the trade-offs between memory use and other factors, such as execution speed or failure modes. For example, a component might pre-allocate some memory buffers to support fast response during normal processing: how much memory should it allocate? The answers to such questions depend critically on the system environment; they may also depend on which client is using the component, or even depend on what the client happens to be doing at the time. The common approach – for the designer to use some idea of an 'average' application to answer such questions – is unlikely to give satisfactory results in a memory limited system.

Therefore: *Design interfaces so that clients control data transfer.*

There are two main steps to designing component interfaces:

1. Minimise the amount of data transferred across interfaces. The principles of 'small interfaces' [Meyer 1997] and 'strong design' [Coplien 1994] say that an interface should present only the minimum data and behaviour to its client. A small interface should not transmit spurious information that most components or their clients will not need. You can reduce the amount of memory overhead imposed by interfaces by reducing the amount of data that you need to be transfer across them.

2. *Determine how best to transfer the data.* Once you have identified the data you need to pass between components, you can determine how best to transfer it. There are many different mechanisms for passing data across interfaces, and we discuss the most important of them in the Implementation section.

For example, later versions of the Spookivity Database 'search' method returned a database **ITERATOR** object [Gamma et al 1995]. The iterator's '`getNext`' function returned a reference to a '`GhostDetails`' result object, which provided methods to return the data of each ghost in turn. This also allowed the implementers of the database component to reuse the same `GhostDetails` object each time; their implementation contained only a database ID, which they changed on each call. The `GhostDetails` methods accessed their data directly from the high-speed database. The revised interface required only a few bytes of RAM to support, and since the database is itself designed to use iterators there was no cost in performance.

Consequences

By considering the memory requirements for each component's interface explicitly, you can reduce the *memory requirements* for exchanging information across interfaces, and thus for the system as a whole. Because much of the memory used to pass information across interfaces is transient, eliminating or reducing interface's memory overheads can make your program's memory use more *predictable*, and support better *real-time* behaviour. Reducing inter-component interface memory requirements reduces the overheads of using more components in a design, increasing *locality* and *design quality* and *maintainability*.

However: Designing small interfaces requires *programmer discipline*, and increases team co-ordination overheads. A memory-efficient interface can be more *complex, and so* require more code and *programmer effort* and increase *testing costs*. As with all designs that save memory, designing small interfaces may increase *time performance*.

❖ ❖ ❖

Implementation

There are a number of issues and alternatives to consider when using designing interfaces between components in small systems. The same techniques can be used whether information is passing 'inward' from a client to a server, in the same direction as control flow (an *efferent flow* [Yourdon and Constantine 1979]), or 'outward' from component to client (an *afferent flow).*

1. Passing data by value vs. by reference.

Data can be passed and returned either by value (copying the data) or by reference (passing a pointer to the data). Passing data by reference usually requires less memory than by value, and saves copying time. Java and Smalltalk programs usually pass objects by reference. Passing references does means that the components are now **SHARING** the data, so the two components need to co-operate somehow to manage the responsibility for its memory. On the other hand, in pass-by-value the receiving components must manage the responsibility for the temporary memory receiving the value. as well. Pass-by-value is common in C++, which can **DISCARD** stack memory.

2. Exchanging memory across interfaces.

There are three common strategies for a client to transfer memory across a component interface:

- **Lending** —some client's memory is lent to the supplier component for the duration of the clients call to the supplier (or longer).
- **Borrowing** —the client gets access to an object owned by the supplier component.
- **Stealing** — the client receives an object allocated by the supplier, and is responsible for its deallocation.

When information is passed inward the client can often *lend* memory to the component for the duration of the call. Returning information 'outward' from component to is more difficult. Although clients can *lend* memory to a supplier, it is often easier for the client to *borrow* a result object from the server, and easier still for the client to *steal* a result object and use it without constraint.

The following section describes and contrasts each of these three approaches. For convenience, we describe a component that returns a single result object; but the same sub-patterns apply when a number of objects are returned.

2.1. Lending: The client passes an object into the component method, and the component uses methods on the object to access its data. If the client keeps a reference to the result object, it can access the data directly, or the component can pass it back to the client. For example, the following Java code sketches how an object using a word processor component could create a new document properties object, and pass it to the word processor, which initialises it to describe the properties of the current document.

```
DocumentProperties d = new DocumentProperties();
wordProcessor.getCurrentDocumentProperties( d );
```

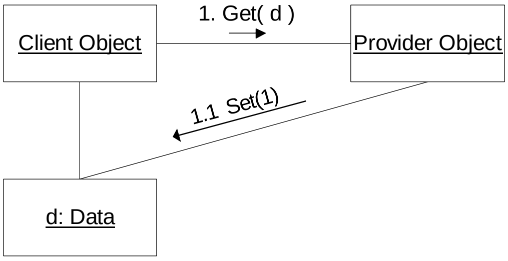
The client can then manipulate the document properties object:

```
long docsize = d.getSize();
long doctime = d.getEditTime();
```

The client must also release the document properties object when it is no longer useful:

```
d = null;
```

because it has kept the responsibility for the document properties object's memory.



When lending memory to a component, the client manages the allocation and lifetime of the data object (the document properties in this case), which may be allocated statically, or on the heap or the stack.

Consider using lending to pass arguments across interfaces when you expect the client to have already allocated all the argument objects, and when you are sure they will need all the results returned. Making the client own a result object obviously gives a fair amount of power and flexibility to the client. It does requires the client to allocate a new object to accept the results, and take care to delete the object when it is not longer needed, requiring *programmer discipline.* The component must calculate all the result properties, whether the client needs them or not.

In C++ libraries a common form of this technique is to return the result by value, copying from temporary stack memory in the component to memory lent by the client.

Another example of lending is where the client passes in a buffer for the component to use. For example in the **BUFFER SWAP** pattern, a component needs to record a collection of objects (e.g. sound samples) in real-time and return them to the client. The client begins by providing a single buffer to the main component, and then provides a new empty buffer every time it receives a filled one back. [Sane and Campbell 1996].

2.2. Borrowing: The component owns a simple or composite object, and returns a reference to that object to the client. The client uses methods on the object to access its data, then signals to the component when it no longer needs the object. For example, the word processor component could let its client borrow an object representing the properties of the current document:

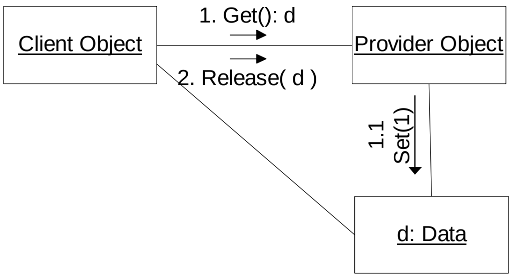
```
DocumentProperties d = wordProcessor.getDocumentProperties();
```

The client can then manipulate the document properties object:

```
long docsize = d.getSize();
long doctime = d.getEditTime();
```

but must tell the word processor when the properties object is no longer required.

```
wordProcessor.releaseDocumentProperties(d);
```



Like lending, borrowing can be used to transfer data both in to and out of a component. Having the component own the result object gives maximum flexibility to the component returning the result. The component can allocate a new data object each time (**VARIABLE DATA STRUCTURE**), or it can hold one or more instances permanently (**FIXED DATA STRUCTURE**), or some combination of the two.

On the other hand, the component now has to manage the lifetime of the result object, which is difficult if there are several clients or several data objects needed at a time. Alternatively, you can allocate only one result object statically, and recycle it for invocation. This requires the client to copy the information immediately it is returned (effectively similar to an ownership transfer). A static result object also cannot handle concurrent accesses, but this is fine as long as you are sure there will only ever be one client at a time.

Alternatively, the component interface can provide an explicit 'release' method to delete the result object. This is rarer in Java and Smalltalk, as these languages make it clumsy to ensure that the release method is called when an exception is thrown. This is quite common in C++ interfaces, as it allows the component to implement **REFERENCE COUNTING** on the object, or just to do `delete this` in the implementation of the `Release` function. For example, the EPOC coding style [Tasker et al 2000] is that all interfaces ('R classes') must provide a `Release` function rather than a destructor. Consider using borrowing when components need to create or to provide large objects for their clients, and clients are unlikely to retain the objects for long periods of time.

2.3. Stealing: The component allocates a simple or composite object, and transfers responsibility for it to the client. The client uses methods on the object to get data, then frees it (C++) or relies on garbage collection to release the memory. For example, the wordprocessor can let its client steal a document properties object:

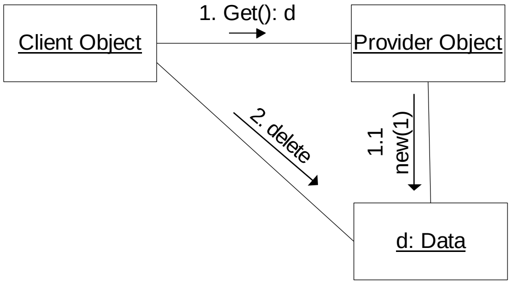
```
DocumentProperties d = wordProcessor.getDocumentProperties();
```

allowing the client to use it as necessary,

```
long docsize = d.getSize();
long doctime = d.getEditlong();
```

but the client now has the responsibility for managing or deleting the object

```
d = null;
```



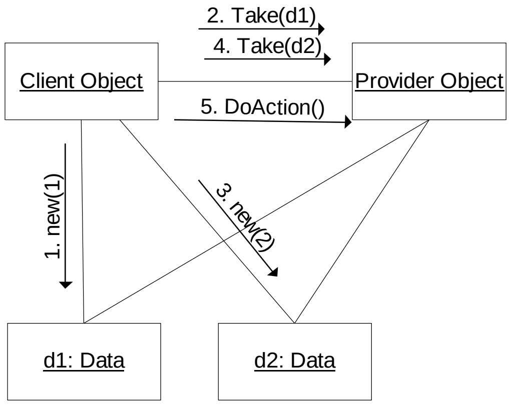
This example shows a client stealing an object originally belonging to a component, however, components can also steal objects belonging to their clients when data is flowing from clients to components. Transferring responsibility for objects (or ownership of objects) is simple to program, and is particularly common in languages such as Java and Smalltalk that support garbage collection and don't need an explicit `delete` operation. In C++ it's most suitable for variable size structures, such as unbounded strings. However in systems without garbage collection, this technique can cause memory leaks unless great *programmer discipline* is used to delete every single returned objects. Ownership transfer forces the server to allocate a new object to return, and this object needs memory. The server must calculate all the properties of the returned object, whether the client needs them or not, wasting *processing time* and memory. Consider using stealing when components need to provide large objects that their clients will retain for some time after receiving them.

3. Incremental Interfaces.

It is particularly difficult to pass a sequence or collection of data items across an interface. In systems with limited memory, or where memory is often fragmented, there may not be enough memory available to store the entire collection. In these cases, the interface needs to be made *incremental* — that is, information is transferred using more than one call from the client to a component, each call transferring only a small amount of information. Incremental interfaces can be used for both inward and outward data transfer. Clients can either make multiple calls directly to a component, or an **ITERATOR** can be used as an intermediate object. Consider using Iterator Interfaces when large objects need to be transferred across interfaces.

3.1. Client Makes Multiple Calls: The client makes several method calls to the component, each call *loaning* a single object for the duration of the call. When all the objects are passed, the client makes a further call to indicate to the component that it's got the entire collection, so it can get on with processing. For example, a client can insert a number of paragraphs into a word processor, calling `addParagraph` to ask the word processor to take each paragraph, and then `processAddedParagraphs` to process and format all the new paragraphs.

```
for (int i = 0; i < num_new_paras; i++) {
    wordProcessor.addParagraph(paras[i]);
};
wordProcessor.processAddedParagraphs();
```



The client making multiple calls is easy to understand, and so is often the approach chosen by novice programmers or used in non-OO languages. However, it forces the component either to find a way of processing the data incrementally (see **DATA CHAINING**), or to create its own collection of the objects passed in, requiring further allocated memory. Alternatively the client can *loan* the objects for the duration of the processing rather than for each call, but this forces the client to keep all the data allocated until the `DoAction` operation completes.

To return information from a component incrementally, the client again makes multiple calls, but the component signals the end of the using a return value or similar.

```
spookivity.findGhosts("transparent|dead");
while (spookivity.moreGhostsToProcess()) {
    ghostScreen.addDisplay(spookivity.getNextGhost());
};
```

3.2 Passing Data via an Iterator: Rather than make multiple calls, the client may *lend* an iterator to the component. The component then accesses further *loaned* objects via the iterator. For example, the client can pass an iterator to one of its internal collections:

```
ghostScreen.displayAllGhosts(vectorOfGhosts.iterator());
```

and the component can use this iterator to access the information from the client:

```
void displayAllGhosts(Iterator it) {
    while (it.hasNext()) {
        displayGhost((Ghost) it.next());
    }
}
```

Passing in an iterator reverses the control flow, so that the component is now invoking messages on the client.

Using an iterator is generally more flexible than making multiple calls to a special interface. The component doesn't have to store its own collection of objects, since it can access them through the iterator. It's important that the interface uses an abstract iterator or abstract collection class, however; a common interface design error is to use a specific collection class instead, which constrains the implementation of the client.

3.3. Returning Data with a Writeable Iterator. A *writable iterator* is an iterator that insert elements into a collection, rather than simply traverse a collection. A writeable iterator produced by the client can be used to implement outward flows from component to client, in just the same way that a normal iterator implements inward flows.

```
Vector retrievedGhosts = new Vector();
spookivity.findGhosts("transparent|dead");
spookivity.returnAllGhosts(retrievedGhosts.writeableIterator());
```

Note that at the time of writing, the Java library does not include writeable iterators.

3**.4. Returning data by returning an iterator.** Alternatively the client may *borrow* or *steal* an iterator object from the component, and access returned values through that:

```
Iterator it = spookivity.findGhostsIterator("transparent|dead");
while (it.hasNext()) {
    ghostScreen.displayGhost((Ghost) it.next());
}
```

Returning an iterator keeps the control flow from the client to the component, allowing the iterator to be manipulated by client code, or passed to other client components.

❖ ❖ ❖

Known Uses

Interfaces are everywhere. For good examples of interfaces suitable for limited memory systems, look at the API documentation for the EPOC or PalmOs operating systems [Symbian 1999, Palm 2000].

Operating system file IO calls have to pass large amounts of information between the system and user applications. Typically, they require buffer memory to be allocated by the client, and then read or write directly into their client side buffers. For example, the classic Unix [Ritchie and Thompson 1978] file system call:

```
read(int fid, char *buf, int nchars);
```

reads up to `nchars` characters from file `fid` into the buffer starting at `buf.` The buffer is simply a chunk of raw memory.

EPOC client-server interfaces always use lending, since the server is in a different memory space to its client, and can only return output by copying it into memory set aside for it within the client. This ensures that memory demand is typically small, and that the client 's memory requirements can be fixed at the start of the project.

Many standard interfaces use iterators. For example, the C++ iostreams library uses them almost exclusively for access to container classes [Stroustrup 1997], and Java'z `Zlib` compression library uses iterators (streams) for both input and output.

See Also

Interfaces have to support the overall memory strategy of the system, and therefore many other memory patterns may be reflected in the interfaces between components.

Interfaces can supply methods to set up simulating a memory failure in the component to allow **EXHAUSTION TESTING** of both client and component. Interfaces that return references to objects owned by the component may **SHARE** these objects, and may use **REFERENCE COUNTING** or **COPY ON WRITE**.

Interfaces, particularly in C++, can enforce constant parameters that refer to **READ ONLY MEMORY** and thus may not be changed. In other languages, such enforcement is part of the interface documentation. Where components use **RESOURCE FILES**, interfaces often specify strings or resources as resource IDs rather than structures. As well as reducing the amount of information passing across the interface, the memory costs of the resource can be charged to the component that actually instantiates and uses it.

If the component (or the programming environment) supports **MEMORY COMPACTION** using handles, then the interface may use handles rather than object references to specify objects in the component.

The patterns for *Arguments and Results* [Noble 2000] and *Type-Safe Session* [Pryce 2000] describe how objects can be introduced to help design interfaces between. Meyers' *Effective C++* [1998] and Sutter's *Exceptional C++* [2000] describe good C++ interface design. Tony Simons has described some options using borrowing, copying and stealing for designing C++ classes [Simons 1998].

Partial Failure

Also known as: Graceful degradation; Feast and Famine.

How can you deal with unpredictable demands for memory?

- No matter how much you reduce a program's *memory requirements,* you can still run out of memory.
- It is better to fail at a trivial task than to rashly abandon a critical task.
- It is more important to keep running that to run perfectly all the time…
- … And much more important to keep running than to crash.
- The amount of memory available to a system varies wildly over time.

No matter how much you do to reduce the *memory requirements* of your program, it can always run out of memory. You can silently discard data you do not have room to store, terminate processing with a rude error message, or continue as if you had received the memory you requested so that your program crashes in unpredictable ways, but you can't avoid the problem. Implicitly or explicitly, you have to deal with running out of memory. In a 'traditional' system, low memory conditions are sufficiently rare that it is not really worth spending programmer effort dealing with the situation of running out of memory. The default, letting the program crash, is usually acceptable. After all, there are lots of other reasons why programs may crash, and users will hardly notice one or two more! However in a memory-limited system, low memory situations happen sufficiently often that this approach would seriously affect the usability of the system, or even makes it unusable.

For example, the Word-O-Matic word processor provides voice output for each paragraph; adds flashing colours on the screen to highlight errors in spelling, grammar and political correctness; and provides a floating window that continuously suggests sentence endings and possible rephrasing. All this takes a great deal of memory, and frequently uses up all the available RAM memory in the system.

There is some good news, however. First, some system requirements are more important than others — so if you have to fail something, some things are better to fail at than others. Second, provided your system can keep running failing to meet one requirement does not have to mean that you will fail subsequent ones. Finally, you are unlikely to remain short of memory indefinitely. When a system is idle, its demands on memory will be less than when it is heavily loaded.

In the Strap-It-On PC, for example, it's more important that the system keeps running, and keeps its watch and alarm timers up to date, than that any fancy editing function actually works. Within Word-O-Matic, retaining the text users have laboriously entered with the two-finger keypad is more important even than displaying that text, and much more important than spelling or grammar checking and suggesting rephrasing.

Therefore:. *Ensure that running out of memory always leaves the system in a safe state.*

Ensure that for every memory allocation there is a strategy for dealing with failure before it propagates through your program.

When a program detects that its memory allocation has failed, its first priority must be to get back to a safe, stable state as soon as possible, and clean up any inconsistencies caused by the failure. As far as possible this should happen without losing any data. Depending on what was being allocated when memory ran out, it may be enough to back out of the action that required the extra memory. Alternatively you might reduce the functionality provided by one or more components; or even shut down the component where the error occurred.

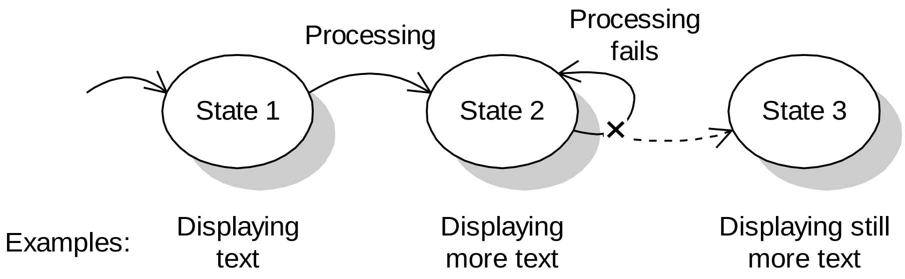


Figure 1: Failing the action that required the extra memory

What is vitally important, however, is to ensure that from the user's point of view, an action succeeds completely or fails completely, leaving the system in a stable state in either case. User interfaces, and component interfaces should make it clear when an important activity that affects the user has failed: if some data has been deleted, or an computation has not been performed.

Once a program has reached a stable state, it should continue as best it can. Ideally it should continue in a 'degraded mode', providing as much functionality as possible, but omitting less important memory-hungry features. You may be able to provide a series of increasingly degraded modes, to cater for increasing shortages of memory. Components can implement a degraded mode by hiding their memory exhaustion from their clients, perhaps accepting requests and queuing them for later processing, or otherwise offering a lower quality service. For example the Word-O-Matic's voice output module accepts but ignores commands from its clients in its 'out of memory' state, which makes programming its clients much simpler.

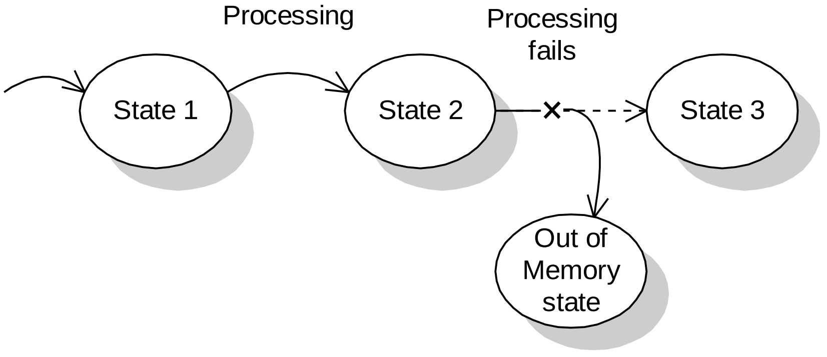


Figure 2: Failing to an Out of Memory state.

Finally, a system should return to full operation when more memory becomes available. Memory is often in short supply while a system copes with high external loads, once the load has passed its memory requirements will decrease. Users directly determine the load on multiprocessing environments like MS Windows and EPOC, so they can choose to free up memory by closing some system applications. A component running in a degraded mode should attempt to return to full operation periodically, to take advantage of any increase in available memory.

For example, when the Word-O-Matic fails to allocate the memory required for voice output of a document, its display screen continues to operate as normal. If the text checker fails, Word-O-Matic doesn't highlight any problems; if the floating window fails it doesn't appear, and the rest of the program carries on regardless. None of these fancier features are essential; and most users will be quite happy with just a text display and the means to enter more text.

Consequences

Supporting partial failure significantly improves a program's *usability*. With careful design, even the degraded modes can provide enough essential functionality that users can complete their work. By ensuring the program can continue to operate within a given amount of memory, partial failure decreases the program's minimum *memory requirements* and increases the *predictability* of those requirements.

Supporting partial failure increases the program's *design quality* – if you support partial failure for memory exhaustion, it's easy to support partial failure (and other forms of failure handling) for other things, like network faults and exhaustion of other resources. Systems that support partial failure properly can be almost totally *reliable*.

However: Partial Failure is hard work to program, requiring *programmer discipline* to apply consistently and considerable *programmer effort* to implement.

Language mechanisms that support partial failure – exceptions and similar – considerably increase the implementation complexity of the system, since programmers must cater for alternative control paths, and for releasing resources on failure.

Partial Failure tends to increase the *global* complexity of the systems, because *local* events – running out of memory – tend to have global consequences by affecting other modules in the system.

Supporting partial failure significantly increases the complexity of each module, increasing the *testing cost* because you must try to test all the failure modes.

❖ ❖ ❖

Implementation

Consider the following issues when implementing Partial Failure:

1 Detecting Memory Exhaustion.

How you detect exhaustion depends on the type of **MEMORY ALLOCATION** you are using. For example, if you are allocating memory from a heap, the operation that creates objects will have some mechanism for detecting allocation failure. If you are managing memory allocation yourself, such as using **FIXED ALLOCATION** or allocating objects dynamically from a pool, then you need to ensure the program checks to determine when the fixed structure or the memory pool is full. The **MEMORY ALLOCATION** chapter discusses this in more detail.

How you communicate memory exhaustion within the program depends on the facilities offered by your programming language. In many languages, including early implementations of C and C++, the only way to signal such an error was to return an error code (rather than the allocated memory). Unfortunately, checking the value returned by every allocation requires a very high level of programmer discipline. More modern languages support variants of exceptions, explicitly allowing functions to return abnormally. In most environments an out-of-memory exception terminates the application by default, so components that implement **PARTIAL FAILURE** need to handle these exceptions.

2 Getting to a Safe State.

Once you have detected that you have run out of memory, you have to determine how to reach a safe state, that is, how much of the system cannot continue because it absolutely required the additional memory being available. Typically you will fail only the function that made the request; in other situations the component may need a degraded mode, or, if a separate executable, may terminate completely.

To determine how much of the system cannot be made safe, you need to examine each component in turn, and consider their invariants, that is, what conditions must be maintained for them to operate successfully [Hoare 1981, Meyer 1997]. If a components' invariants are unaffected by running out of memory, then the component should be able to continue running as is. If the invariants are affected by the memory failure, you may be able to restore a consistent state by deleting or changing other information within the component. If you cannot restore a component to a safe state, you have to shut it down.

If you have to fail entire applications, you may be able to use **APPLICATION SWITCHING** to get to a safe state.

3 Releasing Resources.

A component that has failed to allocate the memory must tidy up after to ensure it has not left any side effects. Any resources it allocated but can no longer use (particularly memory) must be released, and its state (and that of any other affected components) must be restored to values that preserve its invariants.

In C++, exceptions 'unwind' the stack between a `throw` statement and a `catch` statement [Stroustrup 1997]. By default, all stack-based pointers between them are lost, and any resources they own are orphaned. C++ exceptions guarantee to invoke the destructor on any stack-based object, however, so any object on the stack can clean up in their destructors so that they will be tidied up correctly during an exception. The standard template class `auto_ptr` wraps a pointer and deletes it when the stack is unwound.

```
auto_ptr<NetworkInterfaceClass> p(new NetworkInterfaceClass);
p->doSomethingWhichCallsAnException();  // the instance is deleted
```

Although Java has garbage collection, you still have to free objects (by removing all references to them) and release external resources as the stack unwinds. Rather than using destructors, the Java '`try..finally`' construct will execute the '`finally`' block whenever the '`try`' block exits, either normally or abnormally. This example registers an instance of a **COMMAND** [Gamma et al 1995] subclass into a set, and then removes it from the set when an exception is thrown or the command's `execute` method returns normally.

```
Command cmd = new LongWindedCommand();
setOfActiveCommands.add(cmd);

try {
    cmd.execute();
}
finally {
    setOfActiveCommands.remove(cmd);
}
```

EPOC, as an operating system for limited memory systems, has Partial Failure as one of its most fundamental architectural principles [Tasker et al 2000]. Virtually every operation can to fail due to memory exhaustion; but such failure is limited as much as possible and never permitted to cause a memory leak. EPOC's C++ environment does not use C++ exceptions, rather an operating system TRAP construct. Basically, a call to the `leave` method unwinds the stack (using the C longjmp function), until it reach a TRAP harness call. . Client code adds and removes items explicitly from a 'cleanup stack', and then `leave` method automatically invokes a cleanup operation for any objects stored on the cleanup stack. The top-level EPOC system scheduler provides a `TRAP` harness for all normal user code. By default that puts up an error dialog box to warn the user the operation has failed, then continues processing.

Here's an example of safe object construction in EPOC. [Tasker et al 2000]. A **FACTORY METHOD** [Gamma et al 1995], `NewL`, allocates a zero-filled (i.e. safe) object using `new(Eleave)`, then calls a second function, `ConstructL`, to do any operations that may fail. By pushing the uninitialised object onto the cleanup stack, if `ConstructL` fails then it will be deleted automatically. Once the new object is fully constructed it can be removed from the cleanup stack.

```
SafeObject* SafeObject::NewL( CEikonEnv* aEnv )
{
  SafeObject* obj = new (ELeave) SafeObject( aEnv );
  CleanupStack::PushL( obj );
  obj->ConstructL();
  CleanupStack::Pop(); // obj is now OK, so remove it
  return obj;
}
```

The **CAPTAIN OATES** pattern includes another example of the EPOC cleanup stack.

4. Degraded Modes.

Once you've cleaned up the mess after your memory allocation has failed, your program should carry on running in a stable state, even though its performance will be degraded. For example:

- Loading a font may fail; in this case you can use a standard system font.
- Displaying images may fail; you can leave them blank or display a message.
- Cached values may be unavailable; you can get the originals at some time cost.
- A detailed calculation may fail; you can use an approximation.
- Undo information may not be saved (usually after warning the user).

Wherever possible components should conceal their partial failure from their clients. Such encapsulation makes the components easier to design and *localises* the effect of the failure to the components that detect it. Component interfaces should not force clients to know about these failure modes, although they can provide additional methods to allow interested clients to learn about such failure.

You can often use **MULTIPLE REPRESENTATIONS** to help implement partial failure.

5. Rainy Day Fund.

Just as you have to spend money to make money, handling memory exhaustion can itself *require* memory. C++ and Java signal memory exhaustion by throwing an exception, which requires memory to store the exception object; displaying a dialog box to warn the use about memory problems requires memory to store the dialog box object. To avoid this problem, set aside some memory for a rainy day. The C++ runtime system, for example, is required to preallocate enough memory to store the `bad_alloc` exception thrown when it runs out of memory [Stroustrup 1997]. Windows CE similarly sets aside enough memory to display an out-of-memory dialog box [Boling 1998]. The Prograph visual programming language takes a more sophisticated approach — it supplies a rainy day fund class that manages a memory reserve that is automatically released immediately after the main memory is exhausted [MacNeil and Proudfoot 1985].

Example

The following Java code illustrates a simple technique for handling errors with partial failure. The method `StrapFont.font` attempts to find a font and ensure it is loaded into main memory. From the client's point of view, it must always succeed.

We implement a safe state by ensuring that there is always a font available to return. Here, the class creates a default font when it first initialises. If that failed, it would be a failure of process initialisation – implemented by `new` throwing an uncaught `OutOfMemoryError` – preventing the user entering any data in the first place.

```
class StrapFont {

    static Font myDefaultFont =  new Font("Dialog",Font.PLAIN,12);

    public static Font defaultFont() {
        return myDefaultFont;
    }
```

The `StrapFont.font` method tries to create a new Font object based on the description `priavteGetFont` method, which can run out of memory and throw and `OutOfMemoryError`. If a new font object cannot be created then we return the default font. This mechanism also allows safe handling of a different problem, such as when the font does not exist:

```
    public static Font font(String name, int style, int size) {
        Font f;
        try {
            f = privateGetFont(name, style, size);
        }
        catch (BadFontException e) {
            return defaultFont();
        }
        catch (OutOfMemoryError e) {
            return defaultFont();
        }
        return f;
    }

}
```

The client must reload the font using `StrapFont.font` every time it redraws the screen, rather than caching the returned value; this ensures that when memory becomes available the correct font will be loaded.

❖ ❖ ❖

Known Uses

Partial Failure is an important architectural principle. If a system is to support Partial Failure, it must do so consistently. A recent project evaluated a third-party database library for porting to EPOC as an operating system service. Everything looked fine: the code was elegant; the port would be trivial. Unfortunately the library, designed for a memory-rich system, provided no support for partial failure; all memory allocations were assumed either to succeed or to terminate the process. In a service for simultaneous use by many EPOC applications that strategy was unacceptable; memory exhaustion is common in EPOC systems, and the designers couldn't allow a situation where it would cause many applications to fail simultaneously. The library was unsuitable because it did not support Partial Failure.

Degraded Modes are common in GUI applications. If Netscape fails to load a font due to insufficient memory, it continues with standard fonts. Microsoft PowerPoint will use standard fonts and omit images. PhotoShop warns the user and then stops saving undo information.

At a lower level, if the Microsoft Foundation Class framework detects an exception while painting a window, its default behaviour is to mark the window as fully painted. This allows the application to continue although the window display may be incorrect; the window will be repainted when it is subsequently changed by the application.

EPOC's Word Processor makes its largest use of memory when formatting part of a page for display. If this fails, it enters an out-of-memory mode where it displays as much of the text as has been formatted successfully. Whenever a user event occurs, (scrolling, or a redisplay command) Word attempts to reformat the page, leaves its degraded mode if it is successful. EPOC's architecture also has an interesting policy about safe states. The EPOC application framework is event-driven; every application runs by receiving repeated function calls from a central scheduler. Every application is in a safe state when it is not currently executing from the scheduler, so any EPOC application can fail independently of any other [Tasker et al 2000].

See Also

APPLICATION SWITCHING can fail an entire application and begin running another application, rather than terminating an entire system of multiple applications. **MULTIPLE REPRESENTATIONS** can also support partial failure, by replacing standard representations with more memory efficient designs.

An alternative to failing the component that needed the memory is to use the **CAPTAIN OATES** pattern and fail a different and less important component. The **MEMORY ALLOCATION** chapter describes a number of strategies for dealing with allocation failures, such as deferring requests, discarding information, and signalling errors.

Ward Cunningham's **CHECKS** pattern language discusses several ways of communicating partial failure to the user. [Cunningham 1995]. *Professional Symbian Programming* [Tasker et al 2000], *More Effective C++* [Meyers 1996] and *Exceptional C++* [Sutter 2000] describe in detail programming techniques and idioms for implementing Partial Failure with C++ exceptions.

Captain Oates

Also known as: Cache Release.

How can you fulfil the most important demands for memory?

- Many systems have components that run in the background.
- Many applications cache data to improve performance
- User's care more about what they are working on than background activities the system is doing for its own sake.

To the operating system all memory requirements appear equal. To the user, however, some requirements are more equal than others [Orwell 1945].

For example, when someone is using the Strap-It-On PC's word processor to edit a document, they don't care what the fractal screen background looks like. You can increase a system's *usability* by spending scarce resources doing what users actually wants.

Many systems include background components, such as screen savers, chat programs, cryptoanalysis engines [Hayes 1998], or Fourier analyses to search for extraterrestrial intelligence [Sullivan et al 1997]. Systems also use memory to make users' activities quicker or more enjoyable, by downloading music, caching web pages, or indexing file systems. Though important in the longer term, these activities do not help the user while they are happening, and take scarce resources from the urgent, vital, demands of the user.

Therefore: *Sacrifice memory used by less vital components rather than fail more important tasks.*

Warn every component in the system when memory is running out, but while there is still some space left. When a component receives this warning it should release its inessential memory, or in more extreme situations, terminate activities.

If there is no support for signalling memory conditions, processes can keep track of the free memory situation by regular polling, and free inessential resources (or close down) when memory becomes short.

For example when the Word-O-Matic is about to run out of memory the IP networking stack empties its cache of IP address maps and the web browser empties its page cache. Background service processes like the 'Fizzy^{TM}' fractal generator automatically closes down. Consequently, the word processor's *memory requirements* can be met. Figure XXX illustrates a system-wide implementation of the Captain Oates pattern:

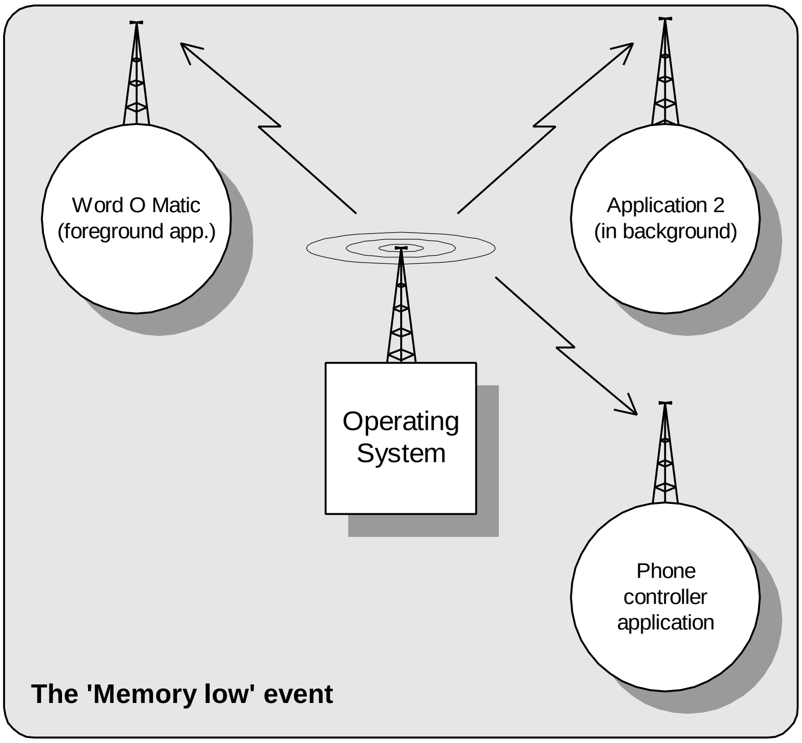


Figure 3: The Memory Low Event

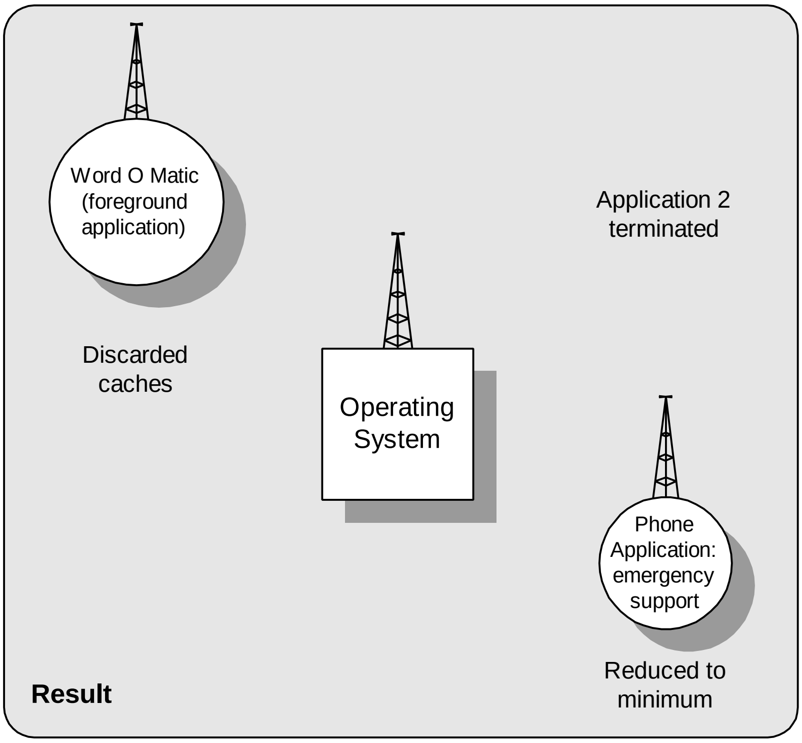


Figure 4: Result of the Memory Low Event

The name of this pattern celebrates a famous Victorian explorer, Captain Lawrence ‘Titus’ Oates. Oates was part of the British team led by Robert Falcon Scott, who reached the South

Pole only to discover that Roald Amundsen’s Norwegian team had got there first. Scott’s team ran short of supplies on the way back, and a depressed and frostbitten Oates sacrificed himself to give the rest of his team a chance of survival, walking out into the blizzard leaving a famous diary entry: “I may be some time”. Oates’ sacrifice was not enough to save the rest of the team, whose remains were found in their frozen camp the next year. Thirty-five kilograms of rock samples, carried laboriously back from the Pole, were among their remains [Limb and Cordingley 1982; Scott 1913].

Consequences

By allocating memory where it is most needed this pattern increases the systems *usability*, and reduces its *memory requirements*. Programs releasing their temporary memory also increase the *predictability* of the system’s memory use.

However: Captain Oates requires *programmer discipline* to consider voluntarily releasing resources. Captain Oates doesn’t usually benefit the application that implements it directly, so the motivation for a development team to implement it isn’t high –there needs to be strong cultural or architectural forces to make them do so. The pattern also requires *programmer effort* to implement and test.

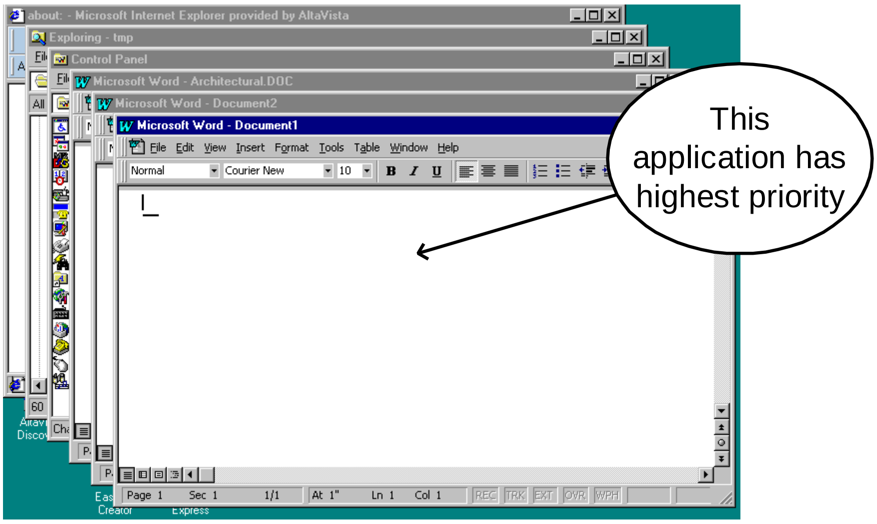
Captain Oates introduces coupling between otherwise unrelated components, which decreases the *predictability* of the system. Releasing resources can reduce the program’s *time performance*. Programs need to be *tested* to see that they do release resources, and that they continue to perform successfully afterwards. Because many programs must handle the memory low signal, Captain Oates is easier with *operating system support*. This is another *global mechanism* that introduces *local complexity* to handle the signal.

❖ ❖ ❖

Implementation

The main point of the Captain Oates pattern is that it releases memory from low priority activities so that high priority activities can proceed. It is inappropriate for a component to release memory if it is supporting high-priority activities. Yet mechanisms that detect low memory conditions are indiscriminate and notify all components equally. So how can you work out what components to sacrifice?

A user interface application can usually determine whether is the current application, i.e. whether it has the input focus so users can interact with it. If so, it should not sacrifice itself when it receives low memory warnings.


about: - Microsoft Internet Explorer provided by AltaVista
Exploring - tmp
Control Panel
Microsoft Word - Architectural.DOC
Microsoft Word - Document2
Microsoft Word - Document1
File Edit View Insert Format Tools Table Window Help
Normal
Courier New
10
Page 1 Sec 1 1/1 At 1" Ln 1 Col 1
This application has highest priority

A background process, though, cannot usually ask the system how important they are. In MS Windows, for example, high priority threads block waiting for some events — the Task manager has a high priority when waiting for Ctrl+Alt+Del key strokes. When the Task Manager detects an event, however, it changes its priority down to a normal. So, calling `GetThreadPriority` cannot give a true indication of how important the task is and whether it's being used.

Most processes, though, can determine how important they are from other information. A component managing network connections, for example, could check whether it had any active connections. Other background processes may not even have that information; a web page cache, for example, may have no direct information about the applications that it supports. Such processes, however must not be directly interacting with the user (otherwise they would have more information about users' activities) and so can usually quite safely release inessential resources when required.

1. Detecting Low Memory Conditions.

Many operating systems provide events that warn applications when memory is low. MS Windows and MS Windows CE send `WM_COMPACTING` and `WM_HIBERNATE` messages to all windows (though not, therefore, to background processes) to warn them that the system memory is getting low [Boling 1998,Microsoft 1997]. Rather than send events, some operating systems or language runtimes call back to system components when memory is low — one example, C++'s `new_handler`, is discussed in the **PARTIAL FAILURE** pattern.

As an alternative, if the system provides functions to show how much memory is in use, then each component can poll to see if memory is low, and release memory when it is. Polling can be unsatisfactory in battery-powered machines, however, since the processor activity uses battery power.

2. Handling low memory events.

When a low memory even occurs, it's useful if each component can determine how short of memory the system is. In the Java JDK 1.2 environment, the runtime object's `getMemoryAdvice()` call answers one of four modes: 'green' meaning there's no shortage, 'yellow' then 'orange' meaning memory is getting low, and 'red' meaning memory is critically low. MS Windows' event, WM_COMPACTING, sends an indication of the proportion of time spent paging memory: 1/8 is equivalent to 'yellow', and is when the message is first sent; anything over 1/4 is critically low [Microsoft 1997].

3. Good Citizenship.

Perhaps the simplest, and often the easiest, approach is for each process to voluntarily give up inessential resources they are not really using. By observing a simple timer, you can release latent resources after a specific time, regardless of the memory status of the rest of the system. For example, the EPOC Web browser loads dynamic DLLs to handle specific types of Web data. If a particular type of data occurs once, it may recur almost immediately, so the Browser DLL loader caches each DLL. If the DLL isn't reused within a few seconds, however, the loader releases it.

Example

This C++ example implements a piece of operating system infrastructure to support a simple Captain Oates mechanism for the EPOC operating system. The Captain Oates application runs in the background and closes applications not currently in use when memory becomes low. Since closing an EPOC application automatically saves its state (a requirement of the PC-synchronisation mechanisms), this does not lose any data. Transient editing state, such as the cursor position in a document or the current item displayed in a file browser, is not maintained, however.

The functionality is in class `COatesTerminator`, which is as follows (omitting function declarations):

```
class COatesTerminator : public CBase {
private:
    RNotifier  iPopupDialogNotifier;    // Provides user screen output
    CPeriodic* iTimer;                  // Timer mechanism
    CEikonEnv* iApplicationEnvironment; // User I/O Handler for this app.

    enum {
        EPollPeriodInSeconds = 10,      // How often to check memory
        EDangerPercentage = 5 };        // Close applications when less free
                                        // memory than this.
};
```

There are various construction and initialisation functions (not included here) to set up the periodic timer and dialog notifier.

The core of the application, however, is the `TimerTickL` function that polls the current memory status and closes applications when memory is low. The free memory reading can be deceptively low if other applications have allocated more memory then they are using. If free memory appears to be low on a first reading, we compress all the memory heaps; this claws back any free pages of memory at the end of each heap. Then a second reading will measure all free memory accurately. If the second reading is also low, we call `CloseAnApplicationL` to close an application.

```
void COatesTerminator::TimerTickL() {
    if ( GetMemoryPercentFree() <= EDangerPercentage ||
         (User::CompressAllHeaps(),
          GetMemoryPercentFree() <= EDangerPercentage ))  {
        CloseAnApplicationL();
    }
}
```

`CloseAnApplicationL` must first select a suitable application to terminate — we do not want to close the current foreground application, the system shell, or this process. Of the other candidates, we'll just close the one lowest in the Z order. Applications are identified to the system as 'window groups' (WG). To find the right window, we first get the identifiers of the window groups we don't want to close (`focusWg`, `defaultWg`, `thisWg`), get the

`WindowGroupList`, then work backwards through the list, and close the first suitable application we find.

Note also the use of the `CleanupStack`, as described in **PARTIAL FAILURE**. We push the array holding the `WindowGroupList` onto the stack when it is allocated, and then remove and destroy it as the function finishes. If the call to get the window group suffers an error, we immediately `leave` the `CloseAnApplicationL` function, automatically destroying the array as it is on the cleanup stack.

```
void COatesTerminator::CloseAnApplicationL() {
    RWsSession& windowServerSession = iApplicationEnvironment->WsSession();

    TInt foregroundApplicationWG =  windowServerSession.GetFocusWindowGroup();
    TInt systemShellApplicationWG = windowServerSession.GetDefaultOwningWindow();
    TInt thisApplicationWG =        iApplicationEnvironment->RootWin().Identifier();

    TInt nApplications=windowServerSession.NumWindowGroups(0);
    CArrayFixFlat<TInt>* applicationList=
                                    new (ELeave) CArrayFixFlat<TInt>(nApplications);
    CleanupStack::PushL( applicationList );
    User::LeaveIfError( windowServerSession.WindowGroupList(0,applicationList) );
    TInt applicationWG=0;
    TInt i= applicationList->Count();
    for (i--; i>=0; i--)  {
        applicationWG = applicationList->At( i );
        if (applicationWG != thisApplicationWG &&
            applicationWG != systemShellApplicationWG &&
            applicationWG != foregroundApplicationWG)
            break;
    }
```

If we find a suitable candidate, we use a standard mechanism to terminate it cleanly. Note that `_LIT` defines a string literal that can be stored in ROM – see the **READ ONLY MEMORY** pattern.

```
    if (i >= 0) {
        TApaTask task(windowServerSession);
        task.SetWgId(applicationWG);
        task.EndTask();
        _LIT( KMessage, "Application terminated" );
        iPopupDialogNotifier.InfoPrint( KMessage );
    }
    CleanupStack::PopAndDestroy(); // applicationList
}
```

This implementation has the disadvantage that it requires polling, consuming unnecessary CPU time and wasting battery power. A better implementation could poll only after writes to the RAM-based file system (straightforward), after user input (difficult), or could vary the polling frequency according to the available memory.

❖ ❖ ❖

Known Uses

The MS Windows application ‘ObjectPLUS’, a hypercard application by ObjectPLUS of Boston, responds to the WM_COMPACTING message. As the memory shortage becomes increasingly critical, it:

- Stops playing sounds
- Compresses images
- Removes cached bitmaps taken from a database

Though this behaviour benefits other applications in the system, it also benefits the HyperCard application itself by releasing memory for other more important activities. By implementing the behaviour in the Windows event handler, the designers have kept that behaviour architecturally separate from other processing in the application.

The Apple Macintosh memory manager (discussed in **COMPACTION**) supports "purgeable memory blocks" — that the memory manager reclaims when memory is low [Apple 1985]. They are used for **RESOURCE FILES**, and file system caches, and dynamically allocated program memory.

MS Windows CE Shell takes a two phase approach to managing memory [Microsoft 1998, Boling 1998]. When memory becomes low, it sends a `WM_HIBERNATE` message to every application. A CE application should respond to this message by releasing as many system resources as possible. When memory becomes even lower, it sends the message `WM_CLOSE` to the lowest priority applications, asking those applications to close — like EPOC, Windows CE requires applications to save their state on `WM_CLOSE` without prompting the user. Alternatively, if more resources become available, applications can receive the WM_`ACTIVATE` message, requesting them to rebuild the internal state they discarded for `WM_HIBERNATE`.

A number of distribued internet projects take advantage of Captain Oates by running as screensavers. When a machine is in use, the screensavers do not run, but after a machine is idle for a few minutes the screensaver uses the idle processor to search for messages from aliens [Hayes 1998] or crack encrypted messages [Sullivan et al 1997].

See Also

Where **CAPTAIN OATES** describes what a program should do when another process in the system runs out of memory, **PARTIAL FAILURE** describes what a process should do when it runs out of memory itself. Many of the techniques for **PARTIAL FAILURE** (such as **MULTIPLE REPRESENTATIONS** and **PROGRAM CHAINING**) are also appropriate for **CAPTAIN OATES**.

FIXED ALLOCATION describes a simple way to implement a form of **CAPTAIN OATES**, where each activity is merely a data structure – simply make new activities overwrite the old ones.

Scott and his team are popular heroes of British and New Zealand culture. See '*Captain Oates: Soldier and Explorer*' [Limb and Cordingley 1982], and '*Scott's Last Expedition: The Personal Journals of Captain R. F. Scott, R.N., C.V.O., on his Journey to the South Pole.*' [Scott 1913].

Read-Only Memory

Also known as: Use the ROM

What can you do with read-only code and data?

- Many systems provide read-only memory as well as writable memory
- Read-only memory is cheaper than writable memory
- Programs do not usually modify executable code.
- Programs do not modify resource files, lookup tables, and other pre-initialised data.

Programs often have lots of *read-only* code and data. For example, the Word-O-Matic word-processor has a large amount of executable code, and large master dictionary files for its spelling checker, which it never changes. Storing this static information in main memory will take memory from data that does need to change, increasing the *memory requirements* of the program as a whole.

Many hardware devices — particularly small ones —support read-only memory as well as writable main memory. The read-only memory may be primary storage, directly accessible from the processor, or indirectly accessible secondary storage. A wide range of technologies can provide read-only memory, from semiconductor ROMs and PROMS of various kinds, through flash ROMs, to read-only compact discs and even paper tape. Most forms of read-only memory are better in many ways than corresponding writable memory — simpler to build, less expensive to purchase, more reliable, more economical of power, dissipating less heat, and more resistant to stray cosmic radiation.

Therefore: S*tore read-only code and data in read-only memory.*

Divide your system code and data into those portions that can change and those that never change. Store the immutable portions in read-only memory and arrange to re-associate them with the changeable portions at run-time.

Word-O-Matic's program's code, for example, is contained in ROM memory in the Strap-It-On portable PC. Word-O-Matic's master dictionary and other resource files are stored in in read-only secondary storage (flash ROM); only user documents and configuration files are stored in writeable memory.

Consequences

This pattern trades off writable main storage for *read-only* storage, reducing the *memory requirements* for main storage and making the it easier to test.. Read-only storage is cheaper than writable storage, in terms of financial cost, power consumption and reliability. If the system can execute programs directly from read-only memory, then using read-only memory can decrease the system's *start-up time.*

Although you may need to copy code and data from *read-only secondary storage* to main memory, you can delete read-only information from main memory without having to save it back to secondary storage. Because they cannot be modified, read-only code and data can be shared easily between programs or components, further reducing the *memory requirements* of the system as a whole.

However: *programmer effort* is needed to divide up the program into read-only and writable portions, and then *programmer discipline* to stick to the division. The disctinction between read-only and writeable inforamtion is fundamentally a *global* concern, although it must be made *locally* for every component in the program.

Code or data in read-only memory is more difficult to *maintain* than information in writable secondary storage. Often, the only way to replace code or data stored in read-only memory is to physically replace the hardware component storing the information. Updating flash memory, which can be erased and rewritten, usually requires a complicated procedure particuarly if the operating system is stored in the memory being updated.

❖ ❖ ❖

Implementation

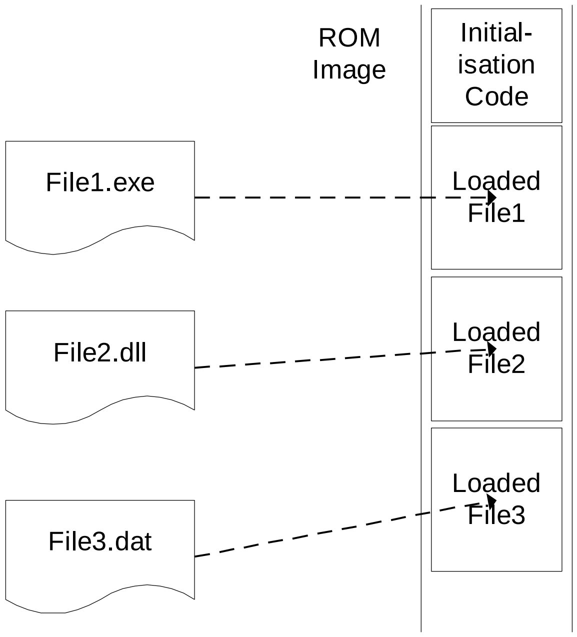
Creating a ‘ROM Image’ (a copy of the final code and data to be stored into read-only memory)is invariably a magical process, requiring major incantations and bizarre software ingredients that are specific to your environment. Across most environments, however, there are common issues to consider when using read-only memory.

1. Storing Executable Code.

If you can run programs directly from read only memory, then you can use it to store executable code. This generally poses two problems: how should the code be represented in read-only memory, and how can it get access to any data it needs?

Most environments store programs as object files — such as executables and dynamic linked libraries — that do not refer to any absolute addresses in memory, instead containing symbolic references to other files or libraries. Before object files can be executed, the operating system’s run-time loader must bind the symbolic references to create completely executable machine code.

To store this executable code in read-only memory you need an extra tool, the ‘ROM builder’, that does the job of the run-time loader, reading in object files and producing a ROM Image. A ROM Builder assigns each object file a base address in memory and copies it into the corresponding position in the ROM image, binding symbolic references and assigning writable memory for heap memory, static memory, and static data. For example, the EPOC system includes a Java ROM builder takes the ‘jar’ or ‘class’ files, and loads them into a ROM image, mimicking the actions of the Java run-time class loader.



If the system starts up by executing code in read-only memory, then the ROM image will also need to contain initialisation code to allocate main memory data structures and to bootstrap the whole system. The ROM Builder can know about this bootstrap code and install it in the correct place in the image.

2. Including Data within Code.

Most programs and programming languages include constant data as well as executable code — if the code is being stored in read-only memory, then this data should accompany it. To do this you need to persuade the compiler or assembler that the data is truly unchangeable.

The C++ standard [Ellis and Stroustrup 1990], for example, defines that instances of objects can be placed in the code segment — and thus in read-only memory — if:

- The instance is defined to be `const`, and
- It has no constructor or destructor.

Thus

```
const char myString[] = "Hello";   // In ROM
char* myString = "Hello";          // Not in ROM according to the Standard.
const String myString( "Hello" );  // Not in ROM, since it has a constructor
```

In particular you can create C++ data tables that can be compiled into ROM:

```
const int myTable[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  // In ROM
```

Note that non-`const` C++ strings are generally not placed in the code segment, since they can be modified, but some compilers support flags or `#pragma` declarations to change this behaviour.

The EPOC system uses a combination of C++ macros and template classes to create instances of strings in read-only memory, containing both a length and the text, as follows:

```
template <TInt S> class TLitC {
   // Various operators...
public:
   int iTypeLength;  // This is the structure of a standard EPOC string
   char iBuf[S];
   };

#define _LIT(name,s) const static TLitC<sizeof(s)> name={sizeof(s)-1,s}
```

This allows EPOC code to define strings in ROM using the `_LIT` macro:

```
_LIT( MyString, "Hello World" );
User::InfoPrint( MyString ); // Displays a message on the screen.
```

The linker filters out duplicate constant definitions, so you can even put `_LIT` definitions in header files.

2.1. Read-only objects in C++. C++ compilers enforce `const` as far as the bitwise state of the object is concerned: `const` member functions may not change any data member, nor may a client delete an object through a `const` pointer [Stroupstrup 1997]. A well-designed class will provide logical `const`-ness by ensuring that any public function is `const` if it doesn't change the externally visible state of the object. For example, the simple String class below provides both a 'logically `const'` access operator, and a non-`const` one. A client given using a `const String&` variable can use only the former.

```
class String {
public:
  // Constructors etc. not shown...
  char operator[]( int i ) const { return rep[i]; }
  char& operator[]( int i ) { return rep[i]; }
private:
  char* rep;
};
```

C++ supports passing parameters by value, which creates a copy of the shared object on the stack. If the object is large and if the function does not modify it, it's common C++ style to **SHARE** the representation by passing the object as a `const` reference. Thus:

```
void function( const String& p );
```

is usually preferable to

```
void function( String p );
```

because it will use less stack space.

2.2. Read-only objects in Java. Java lacks `const`, and so is more restrictive on what data can be stored with the code in read-only memory — only strings and single primitive values are stored in Java constant tables. For example, the following code

```
final int myTable[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  // Don't do this!
```

compiles to a very large function that constructs `myTable,` assigning values to an array in main memory element by element. Storing data for Java programs in read-only memory is thus quite complex. You can encode the data as two-byte integers and store in a string; use C++ to manage the data and access it via the Java Native Interface; or keep the data in a resource file and use file access calls [Lindholm and Yellin 1999].

3. Static Data Structures. Some programs require relatively large constant data structures, for example:

- Encryption algorithms, such as the US Data Encryption Standard (DES).
- Mathematical algorithms, such as log, sine and cosine functions.

- State transition tables, such as those generated by tools to support the Shlaer-Mellor object-oriented methodology [1991].

These tables can be quite large, so its usually not a good idea to store them in main memory, but since they are constant, they can be moved to read-only memory. Managing the development of these data structures can be quite a large task, however.

If the table data changes often during the development process, the best approach is to use a tool to generate the table as a separate file that is incorporated by the ROM Image builder. If the data changes very rarely, then it's usually easiest to copy the table manually into the code, and modify it or the surrounding code to ensure the compiler will place it into read-only memory.

4. Read-Only File Systems.

Some environments can treat read-only memory as if it were a file system. This has the advantage that file system structures can organise the read-only data, and that applications can read it through the normal file operations, although they cannot modify it. For example, EPOC supports a logical file system (Z:), normally invisible to users, which is stored in read-only memory and constructed by the EPOC ROM Builder. All the Resource Files for ROM-based applications are stored in this file system.

File system access is usually slower than direct memory access. If read-only memory can be mapped into applications' address spaces, the data in a ROM filing system can be made available directly, as an optimisation. For example, the EPOC Bitmap Server uses the function `User::IsFileInROM` to access bitmap data directly from ROM.

5. Version Control

Different versions of ROM images will place the same code or data at different addresses. You need to provide some kind of index so that other software in the system can operate with different ROM versions. For example, ROM images often begin with a table of pointers to the beginning of every routine and data structure: external software can find the correct address to call by indirection through this table [Smith 1985].

The **HOOKS** pattern describes how you can store the table in writable memory, so that routines can be extended or replaced with versions stored in writable memory.

Example

The following example uses a read-only lookup table to calculate the mathematical sine function for a number expressed in radians. Because the example is in Java, we must encode the table as a string (using hexadecimal values) because numeric arrays cannot be stored in Java constant tables. The following code runs on our development machine and calculates 256 values of the sine function as sixteen bit integers.

```
final int nPoints = 256;
for (int i = 0; i<nPoints; i++) {
    double radians = i * Math.PI / nPoints;
    int tableValue = (int)(Math.sin(radians) * 65535);
    System.out.print("\\u"+Integer.toHexString(tableValue));
}
```

This code doesn't produce quite correct Java: a few of the escape codes at the start and end lack of the table leading zeros, but it's easier to correct this by hand than to spend more time on a program that's only ever run once.

The `sin` function itself does linear interpolation between the two points found in the table. For brevity, we've not shown the whole table:

```
          static final String sinValues = "\u0000\u0324\u0648. . .\u0000";

      public static float sin(float radians) {
          float point = (radians / (float)Math.PI) * sinValues.length();
          int lowVal = (int) point;
          int hiVal = lowVal + 1;
          float lowValSin = (float)sinValues.charAt(lowVal) / 65535;
          float hiValSin = (float)sinValues.charAt(hiVal) / 65535;
          float result = ((float)hiVal - point) * lowValSin
              + (point - (float)lowVal) * hiValSin;
          return result;
      }
```

On a fast machine with a maths co-processor this `sin` function runs orders of magnitude more slowly than the native `Math.sin()` function! Nevertheless this program provides an accuracy of better than 1 in 20,000, and illustrates the lookup table technique. Lookup tables are widely used in environments that don't support mathematics libraries and in situations where you prefer to use integer rather than floating point arithmetic, such as graphics compression and decompression on low-power processors.

❖ ❖ ❖

Known Uses

Most embedded systems — from digital watches and washing machine controllers to mobile telephones and weapons systems — keep their code and some of their data in read-only memory, such as PROMs or EPROMs. Only run-time data is stored in writable main memory. Palmtops and Smartphones usually keep their operating system code in ROM, along with applications supplied with the phone. In contrast, third party applications live in secondary storage (battery backed-up RAM) and must be loaded into main memory to execute. Similarly, many 1980's home computers, such as the BBC Micro, had complex ROM architectures [Smith 1985].

Even systems that load almost all their code from secondary storage still need some 'bootstrap' initialisation code in ROM to load the first set of instructions from disk when the system starts up. PCs extend this bootstrap to be ROM-based Basic Input Output System (BIOS), which provides generic access to hardware, making it easy to support many different kinds of hardware with one (DOS or Windows) operating system [Chappel 1994].

See Also

Data in read-only storage can be changed using **COPY-ON-WRITE** and **HOOKS. COPY-ON-WRITE** and **HOOKS** also allow some kinds of infrequently changing (but not constant) data to be moved to read only storage.

Anything in read-only storage is suitable for **SHARING** between various programs and different components or for moving to **SECONDARY STORAGE.**

PAGING systems often distinguish between read-only pages and writable pages, and ignore or prevent attempts to write to read-only pages. Several processes can safely share a read-only page, and the paging system can discard it without the cost of writing it back to disk.

Hooks

Also known as: Vector table, Jump table, Patch table, Interrupt table.

How can you change information read-only storage?

- You are using read-only memory
- It is difficult or impossible to change read-only memory once created.
- Code or data in read-only memory needs to be maintained and upgraded.
- You need to make additions and relatively small changes to the information stored in read-only memory.

The main disadvantage of read-only storage is that it is *read-only*. The contents of read-only memory are set at manufacturing time, or possibly upgrade, time; whereupon they are fixed for eternity. Unfortunately, there are always bugs that need to be fixed, or functionality to be upgraded. For example, the released version of the Word-O-Matic code in the Strap-It-On's ROM is rather buggy, and fixes for these bugs need to be included into existing systems. In addition, Strap-It-On's marketing department has decreed that it needs an additional predictive input feature, to automatically complete users' input and so reduce the number of input keystrokes [Darragh, Witten, and James 1990].

If the information is stored in partly writable storage, such as EPROMs, your could issue a completely new ROM image and somehow persuade all the customers to invest the time and risk of upgrading it. Upgrading ROMs is painful for your customers, and often commercially impractical if you don't have control over the whole system. Moreover, a released ROM is unlikely to be so badly flawed as to demand a complete re-release. Often the amount of information that needs to be changed is small, even for significant changes to the system as a whole.

You could ignore the existing read-only memory, and store a new copy of the information in writable main memory. Even if there is enough writable memory in the system to hold a full copy of the contents of the read-only memory, you generally cannot afford to dedicate large amounts of main memory to storing copies of the ROM.

Therefore: *Access read-only information through hooks in writable storage and change the hooks to give the illusion of changing the information..*

The key to making read-only storage extensible is to link your system together through writeable memory, rather than read-only memory. When designing a system that uses read-only storage, do not access that storage directly. Allocate a 'hook' in writable memory for each entry point (to a function, component, data structure, or resource) that is stored in read-only memory, and initialise each hook to refer to its corresponding entry point. Ensure that every access to the entry point is via the writable hook — all accesses, whether from read-only memory or writable memory should use the hook.

To update the read-only memory you copy just that part of the memory you need to modify, and then make the required changes to the copy. Then, you can store the modified copy in writable store, and set the hooks to point to the modified portion. The modified portion can call other parts of the program, if necessary again by indirection through the hooks.

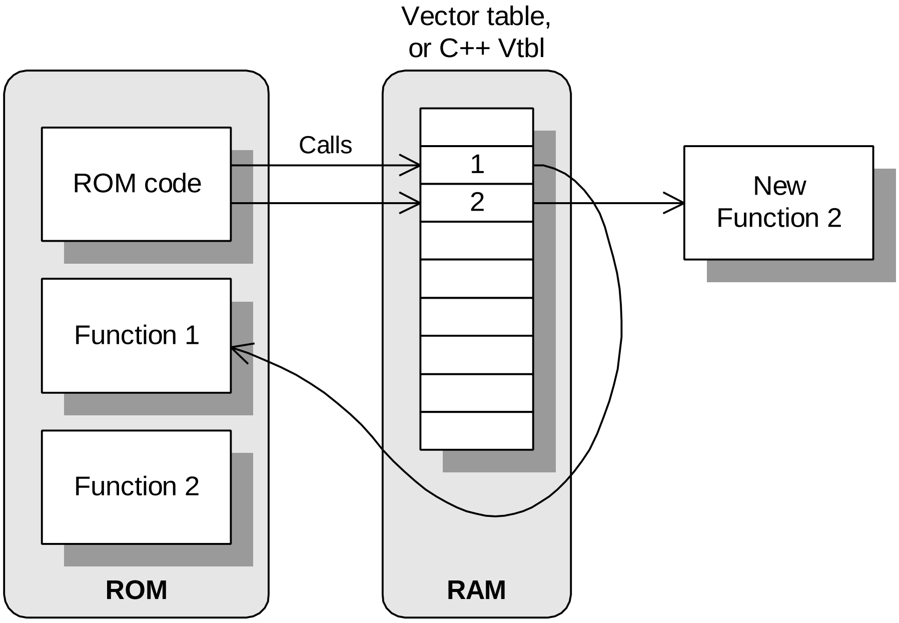


Figure 5: Code Hooks

For example, the Strap-It-On was carefully designed so that every major function is called indirectly through a table of hooks that are stored in RAM and initialised when the system is booted. Bug fixes, extensions, and third party code can be loaded into the system's main memory and the hooks changed to point to them. When an application uses a system function, the hooks ensures it finds the correct piece of code — either the original code in ROM, or the new code in RAM.

Consequences

Hooks let you extend read-only storage, and by making read-only storage easier to use, can reduce the program's writable *memory requirements*.

Providing good hooks increases the *quality* of the *program's design*, making it easier to *maintain* and extend in future. A ROM-based operating system that provides good hooks can enormously reduce the *programmer effort* required to implement any specific functionality.

However: Hooks require *programmer discipline* to design into programs and then to ensure they are used. They also increases the *testing* cost of the program, because the hooks have to be tested to see if they are called at the right times.

Indirect access via hooks is slower than direct access, reducing *time performance*; and the hook vectors take up valuable writable storage, slightly increasing *memory requirements*. Hook vectors are great places to attack system integrity, as any virus writer will tell you, so using hooks can make the system less reliable.

❖ ❖ ❖

Implementation

Consider the following issues when implementing the Hooks pattern:

1. Calling Writable Memory from Read-Only Memory. You can't predict the addresses or entry points of code and data stored in main memory — indeed, because the memory is writable memory addresses can change between versions of programs (or even as a program is running). This makes it difficult for code in ROM to call code or rely on data that is stored in writable memory.

You can address this by using additional hooks that are stored at known addresses in main memory — hooks that point to code and data in main memory, rather than into read-only memory. Code in ROM can follow these hooks to find the addresses of the main memory components that it needs to use.

2. Extending Objects in Read-Only Memory.

Object-oriented environments associate operations with the objects they operate upon – called 'dynamic dispatch', 'message sending' or 'ad-hoc polymorphism'. You can use this to implement rather more flexible hooks. For example, both EPOC and Windows CE support C++ derived classes stored in RAM that inherit from base classes stored in ROM. When the system calls a C++ virtual function, the code executed may be ROM or in RAM depending on the class of the object that the function belongs to. The compiler and runtime system ensures that the C++ virtual function tables (`vtbls`) have the correct entry for each function, so the `vtbls` behave like tables of hooks [Ellis and Stroustrup 1990,]. ROM programmers can use many object-oriented design patterns (such as **FACTORY METHOD** and **ABSTRACT FACTORY**) to implement extensible code [Gamma et al 1995] because the inheritance mechanism does not really distinguish because ROM and RAM classes.

This works equally well in a Java implementation. Java's dynamic binding permits ROM-based code to call methods that may be in ROM or RAM according to the object's class.

3. Extending Data in Read-Only Memory.

Replacing ROM-based data is simplest when the data exists as files in a ROM filing system. In this case, it is sufficient to ensure that application code looks for files in other file systems before the ROM one. EPOC, for example, scans for resource files in the same directory on each drive in turn, taking the drive letters in alphabetic order. Drive Z, the ROM drive, is therefore scanned last.

You can also use accessor functions to use data structures stored in read-only memory. Provided these functions are called through hooks, you can modify the data the rest of the system retrieves from read-only memory by modifying these accessor functions.

If you access read-only memory directly, then you need *programmer discipline* to write code that can use both ROM and RAM simultaneously. When reading data, you should generally search the RAM first, then the ROM; when writing data, you can only write into the RAM. This ensures that if you replace the ROM data by writing to RAM, the updated version in RAM will be found before the original in ROM.

Example

The Strap-It-On's operating system is mostly stored in ROM, and accessed via a table of hooks. The operating system can be updated by changing the hooks. This example describes C code implementing the creation of the hook table and intercepting the operating system function `memalloc`, that allocates memory.

The basic data type in the Strap-It-On operating system is called a `sysobj` — it may be a pointer to a block of memory, a single four-byte integer, two two-byte short integers and so on. Every system call takes and returns a single `sysobj`, so the hook table is essentially a table of pointers to functions taking and returning `sysobjs`.

```
typedef void* sysobj;
const int SIO_HOOK_TABLE_SIZE = 100;
typedef sysobj (*sio_hook_function) (sysobj) ;

sio_hook_function sio_hook_table[SIO_HOOK_TABLE_SIZE];
```

As the system begins running, it stores a pointer to the function that implements `memalloc` in the appropriate place in the hook table.

```
extern sysobj sio_memalloc( sysobj );
const int SIO_MEMALLOC = 0;
sio_hook_table[SIO_MEMALLOC] = sio_memalloc;
```

Strap-It-On applications make system calls, such as the function `memalloc`, by calling ‘trampoline functions’ that indirect through the correct entry in the hook table.

```
void *memalloc(size_t bytesToAllocate) {
    return (void*)sio_hook_table[SIO_MEMALLOC]((sysobj)bytesToAllocate);
}
```

1. Changing a function using a hook

To change the behaviour of the system, say to implement a memory counter, we first allocate a variable to remember the address (in read-only memory) of the original implementation of the `memalloc` call. We need to preserve the original implementation because our memory counter will just count the number of bytes requested, but then needs to call the original function to actually allocate the memory.

```
static sio_hook_function original_memalloc  = 0;

static size_t mem_counter = 0;
```

We can then write a replacement function that counts the memory requested and calls the original version:

```
sysobj mem_counter_memalloc(sysobj size) {
    mem_counter += (size_t)size;
    return original_memalloc( size );
}
```

Finally, we can install the memory counter by copying the address of the existing system `memalloc` from the hook table into our variable, and install our new routine into the hook table.

```
original_memalloc = sio_hook_table[SIO_MEMALLOC];
sio_hook_table[SIO_MEMALLOC] = mem_counter_memalloc;
```

Now, any calls to `memalloc` (in client code and in the operating system, as ROM also uses the hook table) will first be processed by the memory counter code.

❖ ❖ ❖

Known Uses

The Mac, BBC Micro, and IBM PC ROMs are all reached through hook vectors in RAM, and can be updated by changing the hooks. Emacs makes great use of hooks to extend its executable-only code — this way, many users can share a copy of the Emacs binary, but each one have their own, customised environment [Stallman 1984]. NewtonScript allows objects to inherit from read-only objects, using both hooks and copy-on-write so that they can be modified [Smith 1999].

The EPOC ‘Time World’ application has a large ROM-based database of world cities and associated time zones, dialling codes and locations. It also permits the user to add to the list; it stores new cities in a RAM database similar to the pre-defined ROM one, and searches both whenever the user looks for a city.

EPOC takes an alternative approach to updating its ROM. Patches to ROMs are supplied as device drivers that modify the virtual memory map of the system, to map one or more new pages of code in place of the existing ROM memory. This is awkward to manage as the new code must occupy exactly the same space as the code, and exactly the same entry points at exactly the same memory addresses.

See Also

COPY-ON-WRITE is a complementary technique for changing information in **READ-ONLY** STORAGE, and **COPY-ON-WRITE** and **HOOKS** can often be used together.

Using **HOOKS** in conjunction with **READ-ONLY** storage is a special instance of the general use of hooks to extend systems one cannot change directly. Many of the Object-Oriented Design Patterns [Gamma et al 1995] patterns are also concerned with making systems extensible without direct changes.

HOOKS form an important part of the hot-spot approach to systems design [Pree 1995].

Major Technique: Secondary Storage

Version 11/06/00 20:19 - 9

What can you do when you have run out of primary storage?

- Your *memory requirements* are larger than the available primary storage.
- You cannot reduce the system's *memory requirements* sufficiently.
- You can attach *secondary storage* to the device executing the system.

Sometimes your system’s primary memory is just not big enough to fulfil your program’s *memory requirements*.

For example, the Word-O-Matic™ word-processor for the Strap-It-On™ needs to be able to let users edit large amounts of text. Word-O-Matic also supports formatting text for display or printing, not to mention spelling checks, grammar checks, voice output, mail merging and the special StoryDone feature to write the endings for short stories. Unfortunately, the Strap-It-On has only 2Mb of RAM. How can the programmers even consider implementing Word-O-Matic when its code alone will occupy most of the memory space?

The are a number of other techniques in this book which can reduce a program’s *memory requirements*. COMPRESSION can store the information in a smaller amount of memory. Testing applications under a memory limit will ensure programs fit well into a small memory space. You can reduce the system functionality by deleting features or reducing their quality. In many cases, however, these techniques will not reduce the program’s memory requirements sufficiently: data that has to be accessed randomly is difficult to compress; programs have to provide the features and quality expected by the marketplace.

Yet for most applications there is usually some hope. Even in small systems, the amount of memory a program requires to make progress at any given time is usually a small fraction of the total amount of memory used. So the problem is not where to store the code and data needed by the program at any given moment; rather, the problem is where to store the rest of the code and data that may, or may not, be needed by the program in the future.

Therefore: *Use secondary storage as extra memory at runtime.*

Most systems have some form of reasonably fast *secondary storage*. Secondary storage is distinct from RAM, since the processor can’t write to each individual memory addresses directly; but it’s easy for applications to access secondary storage without user intervention. Most forms of secondary storage support *file systems* such that the data lives in files with text names and directory structures. Typically each file also supports *random access* to its data (“get me the byte at offset 301 from the start of the file”).

If you can divide up your program and data into suitable pieces you can load into main memory only those pieces of code and data that you need at any given time, keeping the rest of the program on secondary storage. When the pieces of the program currently in main memory are no longer required you can somehow replace them with more relevant pieces from the secondary store.

There are many different kinds of secondary storage that can be modified and can be accessed randomly: Floppy Disks, Hard Disks, Flash filing systems, Bubble Memory cards, CD-ROM drives, writable CD ROM file systems, and gargantuan file servers accessed over a network. Palm Pilot systems use persistent ‘Memory Records’ stored in secondary RAM. Other forms of secondary storage provide only sequential or read-only access: tape, CD-ROM and web pages accessed over the Internet.

For example the Strap-It-On comes with a CyberStrap, which includes a 32Mb bubble memory store built into its strap along with interfaces for wrist-mounted disk drives. So the Word-O-Matic developers can rely on plenty of 'disk' to store data. Thus Word-O-Matic consists of several separate executables for APPLICATION SWITCHING; it stores each unused document in a DATA FILE; dictionaries, grammar rules and skeleton story endings exist as RESOURCE FILES; optional features are generally shipped as PACKAGES; and the most complex operations use object PAGING to make it seem that the RAM available is much larger than in reality.

Consequences

Being able to use SECONDARY STORAGE can be like getting a lot of extra memory for free — it greatly reduces your program's *primary memory requirements.*

However: the secondary storage must be managed, and information transferred between primary and secondary storage. This management has a *time performance* cost, and may also cost programmer *effort* and *programmer discipline*, impose *local restrictions* to support *global mechanisms,* require hardware *or operating system support,* and reduce the program's *usability.* Most forms of secondary storage require additional devices, increasing the system's *power consumption*.

❖ ❖ ❖

Implementation

There are a few key issues you must address to use secondary storage effectively:

- What is divided up: code, data, configuration information or some combination?
- Who does the division: the programmer, the system or the user?
- Who invokes the loading and unloading: the programmer, the system or the user?
- When does loading or unloading happen?

Generally, the more the program is subdivided and the finer the subdivision, the less the program depends on main memory, and the more use the program makes of secondary storage. Coarser divisions, perhaps addressing only code or only data, may require more main memory but place less pressure on the secondary storage resources.

Making programmers subdivide the program manually requires more effort than somehow allowing the system, or the user, to subdivide the program; and a finer subdivision will require more effort than a coarser one. As a result very fine subdivisions are generally only possible when the system provides them automatically; but creating an automatic system requires significant effort. Making the user divide up the program or data imposes little cost for programmers, but reduces the *usability* of the system.

There are similar trade-offs in deciding who controls the loading and unloading of the divisions. If the system does it automatically this saves work for everyone except the system-builders; otherwise the costs fall on the user and programmer. Sequential loading and unloading is the simplest to implement (and often the worst for the user). More complex schemes that load and unload code or data on demand can be much more seamless to the user, and can even make the reliance on secondary storage transparent to both users and programmers.

❖ ❖ ❖

Specialised Patterns

The rest of this chapter contains five specialised patterns describing different ways to use secondary storage. Figure 1 shows the patterns and the relationships between them: arrows show close relationships; springs indicate a tension between the patterns.

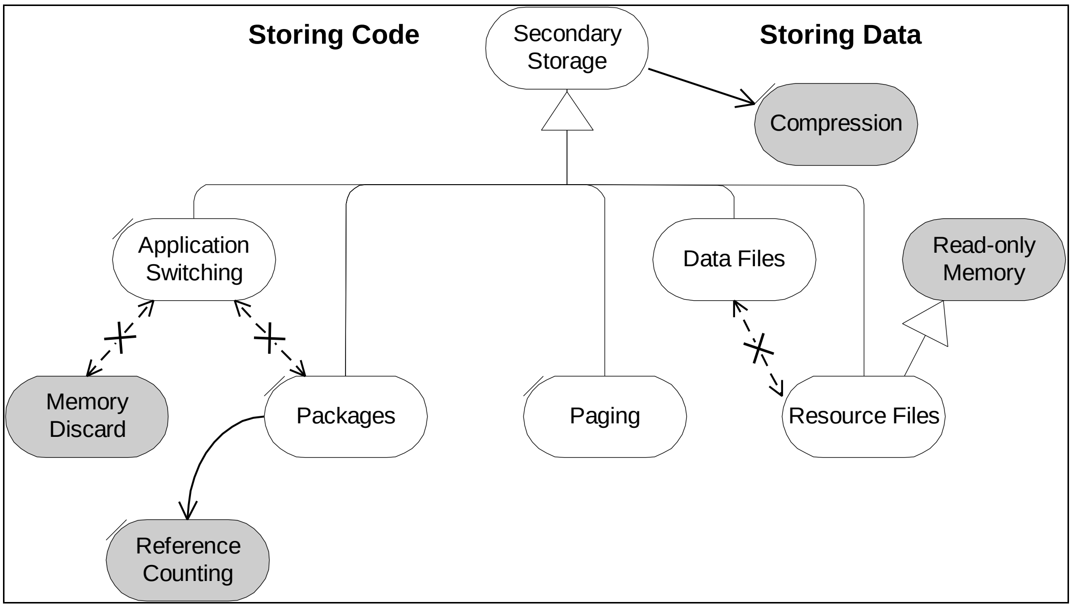


Figure 1: Secondary Storage Patterns

The patterns also form a sequence starting with simple patterns which can be implemented locally, relying only upon programmer discipline for correct implementation, and finishing with more complex patterns which require hardware or operating system support but require much less, if any, *programmer discipline*. Each pattern occupies a different place in the design space defined by the questions above, as follows:

APPLICATION SWITCHING requires the programmer to divide up the program into independent executables, only one of which runs at a time. The order in which the executables run can be determined by the executables themselves, by an external script, or by the user.

DATA FILES uses secondary storage as a location for inactive program data. These files may or may not be visible to the user.

RESOURCE FILES store static read-only data. When the program needs a resource (such as a font, an error message, or a window description), it loads the resource from file into temporary memory; afterwards it releases this memory.

PACKAGES store chunks of the program code. The programmer divides the code into packages, which are loaded and unloaded as required at runtime.

PAGING arbitrarily breaks the program down into very fine units (pages) which are shuffled automatically between primary and secondary storage. Paging can handle code and data, support read-only and shared information between different programs, and is transparent to most programmers and users.

All of these patterns in some sense trade facilities provided in the environment for work done by the programmer. The more complex the environment (compilation tools and runtime system), the less memory management work for the programmer; however a complex run-time environment takes both effort to develop, and has its own memory requirements at run-time. Figure 3 shows where each pattern fits in this scheme.

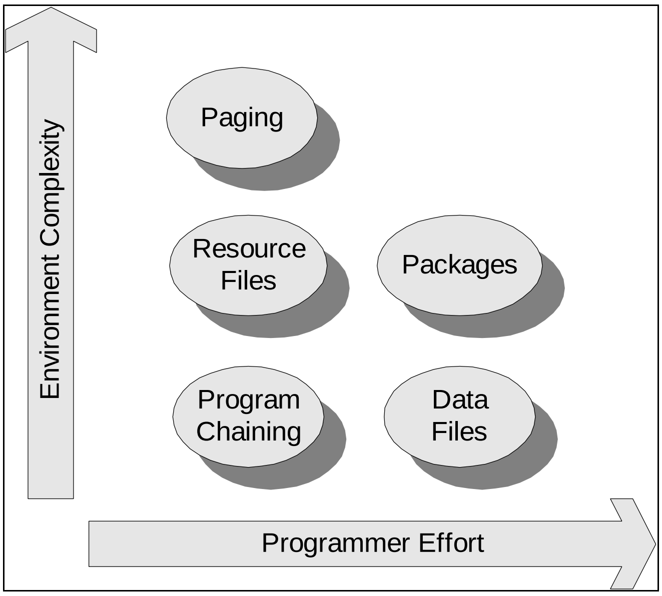


Figure 3: Implementation Effort vs. Environmental Complexity

See Also

READ-ONLY pieces of program or data can be deleted from memory without having to be saved back to secondary storage.

You can use COMPRESSION to reduce the amount of space taken on secondary storage.

Secondary Storage management is one of the primary functions of modern operating systems. More background information and detail on techniques for using Secondary Storage can be found in many operating systems textbooks [Tannenbaum 1992, Leffler, McKusik, Karels and Quarterman 1989, Goodheart and Cox 1994].

Application Switching

How can you reduce the memory requirements of a system that provides many different functions?

Also known as: Phases, Program Chaining, Command Scripts.

- Systems are too big for all the code and data to fit into main memory
- Users often need to do only one task at a time
- A single task requires only its own code and data to execute; other code and data can live on secondary storage.
- It's easier to program only one set of related tasks – one application – at a time.

Some systems are big – too big for all of the executable code and data to fit into main memory at the same time.

For example a Strap-It-On user may do word-processing, run a spreadsheet, read Web pages, do accounts, manage a database, play a game, or use the 'StrapMan' remote control facilities to manage the daily strategy of a large telecommunications network. How can the programmers make all this functionality work in the 2 Mb of RAM they have available – particularly as each of the StrapMan's five different functions requires 1Mb of code and 0.5 Mb of temporary RAM data?

Most systems only need a small subset of their functionality – enough to support one user task – at any given time. Much of the code and data in most systems is unused much of the time, but all the while it occupies valuable main memory space.

The more complex the system and the bigger the development team, the more difficult development becomes. Software developers have always preferred to split their systems architecture into separate components, and to reduce the interdependencies between these components. Components certainly make system development manageable, but they do not reduce main memory requirements.

Therefore: *Split your system into independent executables, and run only one at a time.*

Most operating systems support independent program components in the form of executable files on secondary storage. A running executable is called a process and its code and data occupies main memory. When a process terminates, all the main memory it uses is returned to the system.

Design the system so that behaviour the user will use together or in quick succession will be in the same executable. Provide facilities to start another executable when required, terminating the current one. The new process can reuse all the memory released by the terminated process.

In many operating systems this is the only approach supported; only one process may execute at a time. In MS-DOS the executable must provide functionality to terminate itself before another executable can run; in MacOS and PalmOs there is control functionality shared by all applications to support choosing another application and switching to it. [Chappell 1994, Apple 1985, Palm 2000]. In multi-tasking operating systems this pattern is still frequently used to reduce main memory requirements.

For example, no Strap-It-On user would want to do more than one of those tasks at any one time; it's just not physically possible given the small size of the screen. So each goes in a separate executable (word-processor, spreadsheet, web browser, accounting, database, Doom), and the Strap-It-On provides a control dialog that allows the user to terminate the current application and start another. Each application saves its state on exit and restores it on restart, so that – apart from the speed of loading – the user has no way of telling the application has terminated and restarted. The StrapMan application, however, wouldn't fit in RAM as a single executable. So the StrapMan's authors split it into six different executables (one for the main program and one for each function), and made the main one 'chain' to each other executable as required.

Consequences

The *memory requirements* for each process are less than the *memory requirements* for the entire system. The operating system reclaims the memory when the process terminates, so this reduces *programmer effort* managing memory and reduces the effects of 'memory leaks'.

Different executables may be in different implementation languages, and be an interpreted or compiled as required. Some executables may also be existing 'legacy' applications, or utilities provided by the operating system. So APPLICATION SWITCHING may significantly reduce the *programmer effort* to produce the system, encouraging *reuse* and making *maintenance* easier. Script-based approaches can be very flexible, as scripts are typically very easy to modify. Also applications tend to be geared to stopping and starting regularly, so errors that terminate applications may not be so problematic to the user, increasing the system's robustness.

In single-process environments, such as PalmOs, each process occupies the same memory space, so the amount of memory required is easy to *predict*, which improves *reliability*, makes *testing easier* and removes the effects of the *global* memory use on each *local* application. You only need to start the first process to get the system running, reducing *start-up times*. It's also easy to know what's happening in a single-process environment, simplifying *real-time* programming.

However: Dividing a large program into a good set of processes can be difficult, so a multi-process application can require significant *programmer effort* to design, and *local complexity* in the implementation.

If you have many executables, the cost of starting each and of transferring data can dominate the system's *run time performance*; this is also a problem if the control flow between different processes is complex — if processes are started and terminated frequently.

In single-process environments the user can use only the functionality in the current executable, so chaining tends to reduce the system's *usability*. If the user has to manage the processes explicitly, that also reduces the program's *usability*.

This pattern does not support background activities, such as TCP/IP protocols, interfacing to a mobile phone, or background downloading of email. Such activities must continue even when the user switches tasks. The code for background tasks must either be omitted (reducing *usability*), live in a separate process (increasing *programmer effort*), or be implemented using interrupt routing (requiring large amounts of specialised *programmer effort)*.

❖ ❖ ❖

Implementation

To implement Application Switching you have to divide up the system into separate components (see the SMALL ARCHITECTURE pattern). Communication between running processes can be difficult, so in general the split must satisfy these rules:

- The control flow between processes is simple
- There is little transient data passed between the processes.
- The division makes some kind of sense to the user.

Figure 3 shows the two main alternatives that you can use to implement process switching:

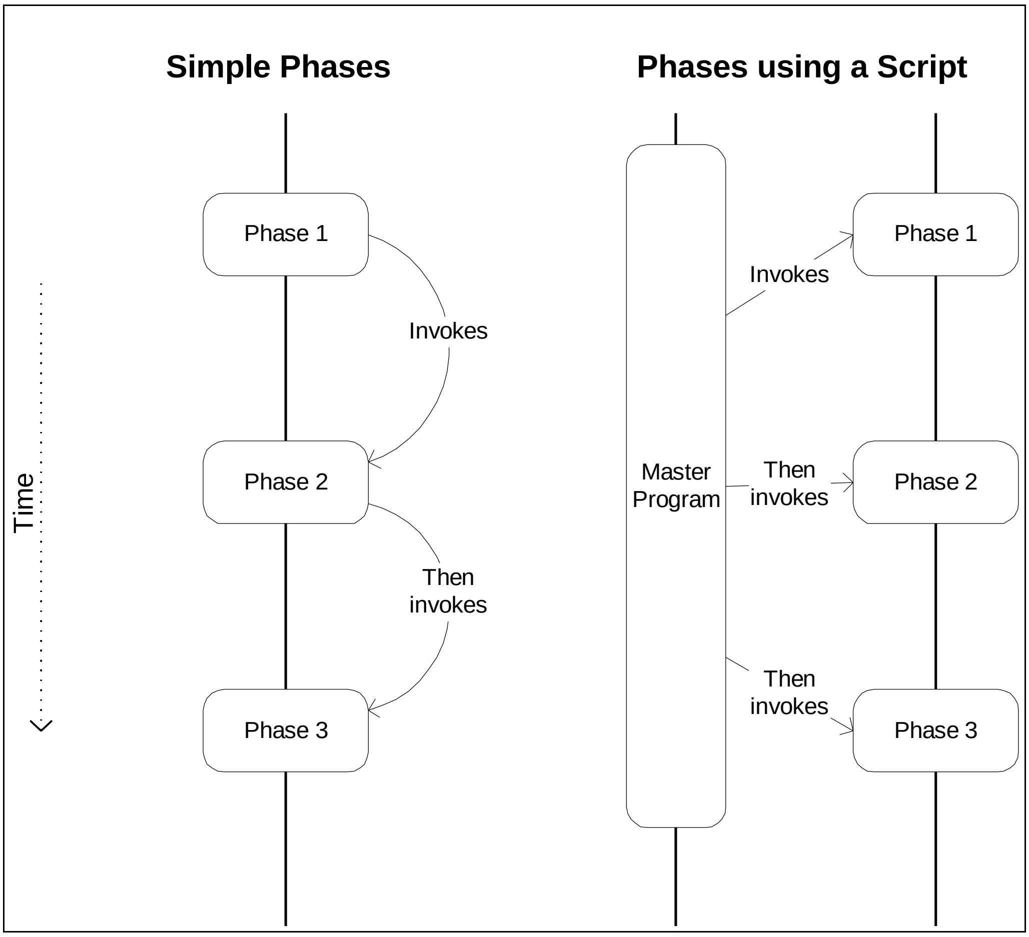


Figure 5: Two different approaches to implementing Phases

1. Program Chaining.

One process can pass control explicitly to the following process. This is called ‘program chaining’, after the ‘`CHAIN`’ command in some versions of the BASIC programming language [Digital 1975; Steiner 1984]. Program Chaining requires that each executable to know which executable to go to next. This can be programmed explicitly by each application, or as part of an application framework library. Given such an application framework, each executable can use the framework to determine which application to switch to next, and to switch to that application, without requiring much programmer effort. The MacOs (task switcher) and PalmOs application frameworks do this [Apple 1984, Palm 2000].

2. Master Program.

Alternatively, a script or top-level command program can invoke each application in turn. A master program, by contrast, encourages reuse because each executable doesn't need to know much about its context and can be used independently. The UNIX environment pioneered the idea of small interoperable tools designed to work together in this way [Kernighan and Pike 1984]. Even with a master program, the terminating program can help determine which application to execute next by passing information back to the master program using exit codes, or by producing output or temporary files that are read by the master program.

3. Communicating between Processes.

How can separate components communicate when only one process is active at a time? You can't use main memory, because that is erased when each process terminates. Instead you need to use one or more of the following mechanisms:

- Command line parameters and environment variables passed into the new process.
- Secondary storage files, records or databases written by one process and read by another.
- Environment-specific mechanisms. For example, many varieties of Basic complimented the `CHAIN` command with a `COMMON` keyword that specifies data preserved when a new process overwrites the current one [Steiner 1984].

4. Managing Data.

How do you make it seem to the user that an application never terminates, even when it is split up in separate processes? Many environments only support a small number of processes, maybe just one, but users don't want to have to recreate all their state each time they start up a new application. They want the illusion that the application is always running in the background.

The solution is for the application to save an application's state to SECONDARY STORAGE on exit, and to restore this state when the application's restarted. Many OO libraries and environments support ways of 'streaming' all the important objects – data and state – as a single operation. The approach requires a binary 'file stream', which defines stream functions to read and write primitive types (e.g. int, char, float, string). Each class representing the application's state then defines its own streaming functions.

When you are streaming out object-oriented applications, you need to ensure each object is streamed only once, no matter how many references there may be to it. A good way to deal with this is to have the 'file stream' maintain a table of object identifiers. Each time the stream receives a request to stream out an object it searches this table, and if it finds the object already there, it just saves a reference to the file location of the original instead of saving it again.

The Java libraries support persistence through the Serialization framework [Chan et al 1998]. Any persistent class implements the `Serializable` interface; it needs no other code – the runtime environment can serialize all its data members, following object references as required (and writing each object only once, as above). The classes `ObjectOutputStream` and `ObjectInputStream` provide methods to read and write a structure of objects: `writeObject` and `readObject` respectively. By convention the files created usually have the extension '`.ser`'; some applications ship initial '`.ser`' files with the Java code in the JAR archive.

Examples

Here's a very trivial example from an MS Windows 3.1 system. We cannot use the disk-checking program, scandisk, while MS Windows is running, so we chain it first, then run Windows:

```
@REM AUTOEXEC.BAT Command file to start MS Windows 3.1 from DOS
@REM [Commands to set paths and load device drivers omitted]
C:\WINDOWS\COMMAND\scandisk /autofix /nosummary
win
```

The following Java routine chains to a different process, terminating the current process:

```
void ChainToCommand(String theCommand) throws IOException {
    Runtime.getRuntime().exec(theCommand);
    Runtime.getRuntime().exit( 0 );
}
```

Note that if this routine is used to execute another Java application, it will create a new Java virtual machine before terminating the current one, and the two VMs will coexist temporarily, requiring significant amounts of memory.

The Unix `exec` family of functions is more suitable for single process chaining in low memory; each starts a new process in the space of the existing one [Kernighan and Pike 1984]. The following C++ function uses Microsoft C++'s `_execl` variant [Microsoft 1997]. It also uses the Windows environment variable '`COMSPEC`' to locate a command interpreter, since only the command interpreter knows where to locate executables and how to parse the command line correctly.

```
void ChainToCommand( string command )
{
   const char *args[4];
   args[0] = getenv( "comspec" );
   args[1] = "/c";
   args[2] = command.c_str();
   args[3] = 0;
   _execv( args[0], args );
}
```

The function never returns. Note that although all the RAM memory is discarded, `execl` doesn't close file handles, which remain open in the chained process. See your C++ or library documentation for '`execl'` and the related functions.

The following is some EPOC C++ code implementing streaming for a simple class, to save data to files while the application is switched. The class, `TSerialPortConfiguration`, represents configuration settings for a printer port. Most of its data members are either C++ `enum's` with a small range of values, or one-byte integers (`char` in C++, `TInt8` in EPOC C++); `TOutputHandshake` is a separate class:

```
class TSerialPortConfiguration {
   // Various function declarations omitted…
   TBps iDataRate;
   TDataBits iDataBits;
   TStopBits iStopBits;
   TParity iParity;
   TOutputHandshake iHandshake;
   };
```

The functions `InternalizeL` and `ExternalizeL` read and write the object from a stream. Although the values `iDataRate` are represented internally as 4-byte integers and enums, we know we'll not loose information by storing them as PACKED DATA, in a single byte. The class `TOutputHandshake` provides its own streaming functions, so we use them:

```
EXPORT_C void TSerialPortConfiguration::InternalizeL(RReadStream& aStream)
   {
   iDataRate = (TBps) aStream.ReadInt8L();
   iDataBits = (TDataBits) aStream.ReadInt8L();
   iStopBits = (TStopBits) aStream.ReadInt8L();
   iParity = (TParity) aStream.ReadInt8L();
   iHandshake.InternalizeL(aStream);
   }

EXPORT_C void TSerialPortConfiguration::ExternalizeL(RWriteStream& aStream) const
   {
   aStream.WriteInt8L(iDataRate);
   aStream.WriteInt8L(iDataBits);
   aStream.WriteInt8L(iStopBits);
   aStream.WriteInt8L(iParity);
   iHandshake.ExternalizeL(aStream);
   }
```

❖ ❖ ❖

Known Uses

The PalmOs and early versions of the MacOs environments both support only a single user process at any one time; both provide and framework functions to simulate multi-tasking for the user. MacOs uses persistence while PalmOs uses secondary storage 'memory records' to save application data [Apple 1985, Palm 2000].

The UNIX environment encourages programmers to use processes by supporting *scripts* and making them executable in the same way as binary executables, with any suitable scripting engine [Kernighan and Pike 1984]. In Windows and the DOS environments, the only fully-supported script formats are the fairly simple BAT and CMD formats, although it's trivial to create a simple Windows BAT file to invoke more powerful scripting language such as Tcl [Ousterhout 1994] and Perl [Wall 1996].

The Unix Make utility manages application switching (and the DATA FILES required) to compile a program, generally running any preprocessors and the appropriate compiler process for each input file in turn, then running one or more linker processes to produce a complete executable [Kernighan and Pike 1984].

See Also

PACKAGES provide similar functionality within a single process – by delaying code loading until it's required. However whereas in PROCESS SWITCHING the operating system will discard the memory, code and other resources owned by a task when the task completes, a PACKAGE must explicitly release these resources.

PAGING provides much more flexible handling of both code and data.

The executables can be stored on SECONDARY STORAGE, using COMPRESSION.

The MEMORY DISCARD pattern has a similar dynamic to this pattern but on a much smaller scale. Where APPLICATION SWITCHING recovers all the memory occupied by an process only when it terminates, MEMORY DISCARD allows an application to recover the memory occupied by a group of objects in the middle of its execution.

Data File Pattern

What can you do when your data doesn't fit into main memory?

Also known as: Batch Processing, Filter, Temporary File

- Systems are too big for all the code and data to fit into RAM together
- The code by itself fits into RAM (or can be fitted using other patterns)
- The data doesn't fit into RAM
- Access to the data is sequential.

Sometimes programs themselves are quite small, but need to process a large amount of data — the *memory requirements* mean that the program will fit into main memory, but the data requires too much memory.

For example, the input and output data for the Word-O-Matic Text Formatter can exceed the capacity of the Strap-It-On's main memory when formatting a large book. How should the Word-O-Matic designers implement the program to produce the output PostScript data, let alone to produce all the index files and update all the cross references, when it's physically impossible to get them all into RAM memory?

Dividing the program up into smaller phases (as in APPLICATION SWITCHING) can reduce the memory required by the program itself, but this doesn't help reduce the memory requirements for the input and output. Similarly, COMPRESSION techniques may reduce the amount of secondary storage required to hold the data, but don't reduce the amount of main memory need to process it.

Yet most systems don't need you to keep all data in RAM. Modern operating systems make it simple to read and write from files on Secondary Storage. And the majority of processing tasks do not require simultaneous access to all the data.

Therefore: *Process the data a little at a time and keep the rest on secondary storage.*

Use sequential or random file access to read each item to process; write the processed data sequentially back to one or more files. You can also write temporary items to secondary storage until you're ready to use them. If you are careful, the amount of main memory needed for processing each portion will be much less than the total memory that would be required to process all the data in main memory. You need to be able to store both input and output as files in SECONDARY STORAGE, so the input and output data must be partitioned cleanly.

For example Word-O-Matic stores its chapters as separate text files. The Word-O-Matic Text Formatter (nicknamed the 'Wombat') makes several passes over these files, see Figure XXX. The first pass scans all the chapter files in turn, locating the destinations of cross references and index entries in the file data, and writes all the information it needs to create each cross-reference to a temporary 'cross reference' file. Wombat's second pass then scans the cross-reference file to create an in-memory index to this file, then reads each chapter file, creating a transient version with the cross references and indexes included. It reads the cross-reference data by random access to the index file using the in-memory index. Since the page numbering changes as a result of the updates, Wombat also keeps an in-memory table showing how each reference destination has moved during this update. Finally Wombat's third pass reads each transient chapter file a bit at a time, and writes out the PostScript printout sequentially, making the corrections to the page numbers in the index and references using its in-memory table as it does. Using these techniques Wombat can format an entire book using as little as 50Kb of RAM memory.

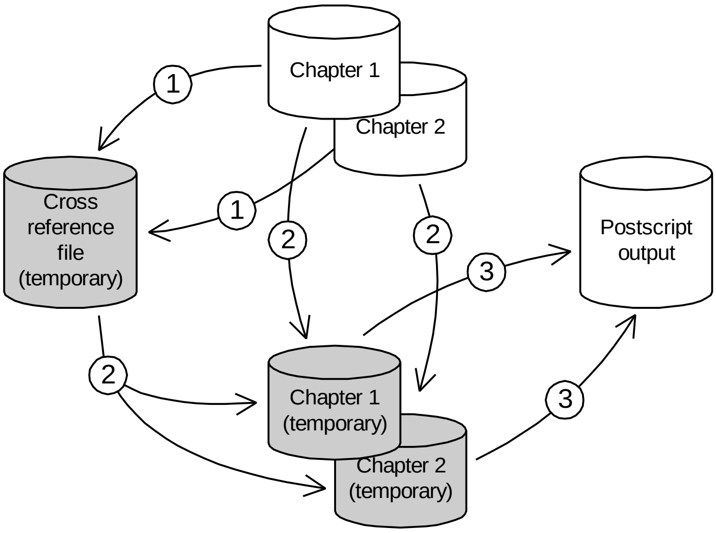


Figure 6: Wombat's Data Files and Phases

Consequences

The *memory requirements* for processing data piecemeal are reduced, since most of the data lives on secondary storage. The system's main memory requirements are also much more *predictable*, because you can allocate a fixed memory to support the processing, rather than a variable amount of memory to store a variable amount of data.

You can examine the input and output of functions in an application using utilities to look at the secondary storage files, which makes *testing* easier. Data Files also make it easy to split an application into different independent components linked only by their data files, reducing the *global* impact of *local* changes, and making *maintenance* easier. Indeed Data Files also make it much easier to implement phases, allowing APPLICATION SWITCHING; for example, Wombat's phase 1 is in a different executable from phases 2 and 3.

However: *Programmer effort* is required to design the program so that the data can be processed independently. Processing data incrementally adds *local complexity* to the implementation, which you could have avoided by processing the data *globally* in one piece. If you need to keep extra context information to process the data, then managing this information can add *global complexity* to the program.

Data chaining can provide slower *run-time performance* than processing all the input in one piece, since reading and writing many small data items is typically less efficient than reading or writing one large item. Repeated access to secondary storage devices can increase the system's *power consumption*, and can even reduce the lifetime of some secondary storage media, such as flash RAM and floppy disks. The limitations of data files – such as imposing ordering rules on the input, or needing the user or client software to manage files – can reduce the system's *usability*.

❖ ❖ ❖

Implementation

The Wombat example above illustrated the four main kinds of operation on data files:

1. Simple Sequential Input (reading each chapter in turn)

2. Simple Sequential Output (writing the final output file)
3. Random Access (reading and writing the cross-reference file)
4. Sequential output to several files (writing the temporary chapter files)

Here are some issues to consider when using data files:

1. Incremental Processing. One simple and common way to manipulate data files is to read an entire file sequentially from input, and/or to write a second file sequentially to the output (see figure XX). Incremental processing requires extra *programmer effort* to implement, because the program must be tailored specially to process its input file incrementally. Because the program processes one large file in small increments, the program is typically responsible for selecting the increments to process (although this can be left to the user by requiring them to indicate increment boundaries in the data file, or provided a collection of smaller data files).

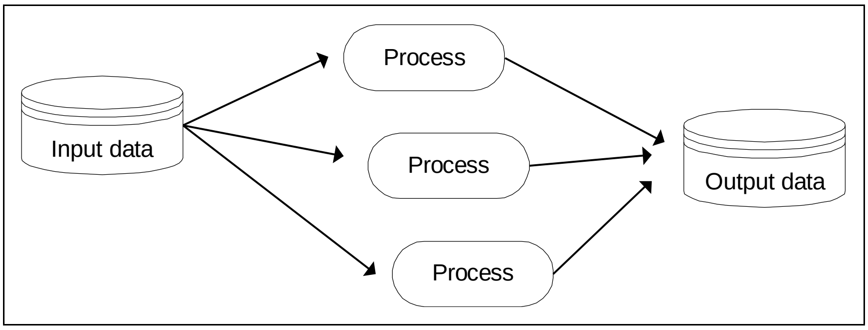


Figure 7: Incremental Processing

Because the whole input file is processed in a single operating systems process, incremental data chaining makes it easier to maintain global contextual information between each processing stage, and easier to produce the final output — the final output is just written incrementally from the program. Unfortunately, precisely because it works in one single long-running process, it can be more difficult to keep the *memory requirements* down to a minimum.

2. Subfile Processing. Rather than processing a single file sequentially, you can divide data up into a number of smaller subfiles. Write a program which processes one subfile, producing a separate output file. Run this program multiple times (typically sequentially) to process each subfile, and then combine the subfiles to produce the required output (see Figure YYY).

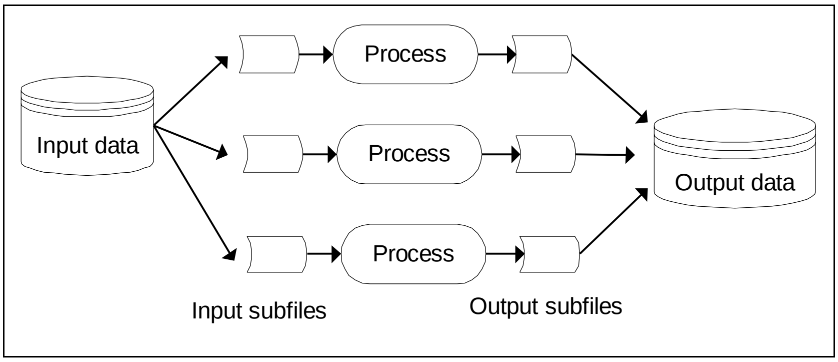


Figure 8: Subfile Processing

Subfile processing has several advantages, provided it’s easy to divide up the data. Subfile processing tends to require less memory, since only a subset of the data is processed at a time;

and it is more robust to corruption and errors in the data files, since each such problem only affects one file. Unfortunately, splitting the files requires effort either on the part of the program or on the part of the user: co-ordinating the processing and combining the subfiles requires programmer effort. See the pattern APPLICATION SWITCHING for a discussion of techniques for communication between such processes.

Many compilers use subfile processing: they compile each code file separately, and only combine the resulting temporary object files in a separate link phase. Because of its enormous potential for reducing memory use, subfile processing was ubiquitous in old-time batch tape processing [Knuth 1998].

3. Random Access. Rather than reading and writing files sequentially (whether incrementally or using subfiles) you can access a single file randomly, selecting information and reading and writing it in any order. Random access generally requires more *programmer effort* than incremental or subfile processing, but is much more flexible: you don't have to determine the order items can be processed (and possibly divide them into subfiles) in advance.

To use random access, each process needs to be able to locate individual data items within the files on secondary storage. Generally, you will need an index, a list of offsets from the start of the file for each item of data required. Because the index will be used for most accesses to the file, it needs to be stored in main memory, or easily accessible from main memory. Effective indexing of files is a major science in its own right, but for simple applications there are two straightforward options:

- The file may contain its own index, perhaps at the start of the file, which is read into RAM by the process. RESOURCE FILES often use this approach.
- The application may scan the file on start up, creating its own index. The Wombat text processor did this with its cross-reference file.

More complicated systems may use indexes in different files from the data, or even have indexes to the index files themselves. File processing is covered in texts such as Folk, Zoellick and Riccardi [1988], Date [1999] and Elmasri and Navathe [2000].

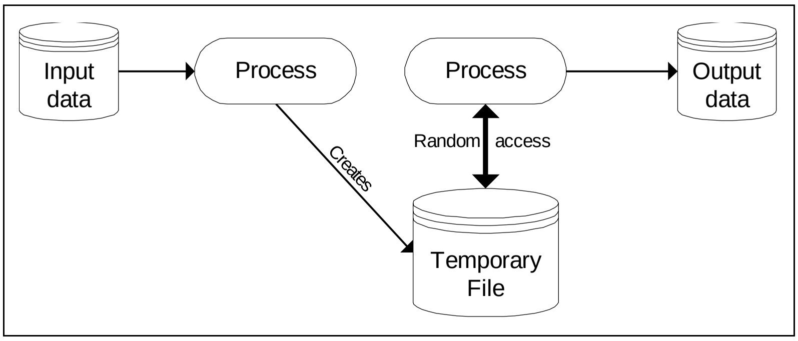


Figure 9: Random Access

Examples

1. Simple Subfile Processing

File compilation provides a typical example of subfile processing. The user splits each large program into a number of files, and the compiler processes each file individually. Then the linker '`ld`' combines the various '`.o`' output files into an executable program, `testprog`.

```
cc main.c
cc datalib.c
cc transput.c
ld –o testprog main.o datalib.o transput.o
```

2. Incremental Processing

The following Java code reverses the characters in each line in a file. It reads each line into a buffer, reverses the characters in the buffer, and then writes the buffer out into a second file. We call the `reverse` method with a `BufferedReader` and `BufferedWriter` to provide more efficient access to the standard input and output than direct access to the disk read and write functions, at a cost of some memory:

```
reverse(new BufferedReader(new InputStreamReader(System.in)),
        new BufferedWriter(new OutputStreamWriter(System.out)));
```

The `reverse` method does the work, using two buffers, a `String` and a `StringBuffer`, because `Strings` in Java cannot be modified.

```
public void reverse(BufferedReader reader, BufferedWriter writer)
   throws IOException {
   String line;
   StringBuffer lineBuffer;

   while ((line = reader.readLine())!=null) {
       lineBuffer = new StringBuffer(line);
       lineBuffer.reverse();
       writer.write(lineBuffer.toString());
       writer.newLine();
   }
   writer.close();
}
```

The important point about this example is that it requires only enough memory to hold the input and output buffers, and a single line of text to reverse, rather than the entire file, and so can handle files of any length without running out of memory.

3. Processing with Multiple Subfiles

Consider reversing all the bytes in a file rather than just the bytes in each line. The simple incremental technique above won't work, because it relies on the fact that processing one line does not affect any other lines. Reversing all the characters in a file involves the whole file, not just each individual line.

We can reverse a file without needing to store it all in memory by using subfiles on secondary storage. We first divide (`scatter`) the large file into a number of smaller subfiles, where each subfile is small enough to fit into memory, and then we can reverse each subfile separately. Finally, we can read (`gather`) each subfile in reverse order, and assemble a new completely reversed file.

```
public void run() throws IOException {
   scatter(new BufferedReader(new InputStreamReader(System.in)));
   gather(new BufferedWriter(new OutputStreamWriter(System.out)));
}
```

To scatter the file into subfiles, we read `SubfileSize` bytes from the input reader into a buffer, reverse the buffer, and then write it out into a new subfile

```
protected void scatter(BufferedReader reader) throws IOException {
   int bytesRead;
   while ((bytesRead = reader.read(buffer, 0, SubfileSize)) > 0) {
       StringBuffer stringBuffer = new StringBuffer(bytesRead);
       stringBuffer.append(buffer, 0, bytesRead);
       stringBuffer.reverse();
       BufferedWriter writer =
        new BufferedWriter(new FileWriter(subfileName(nSubfiles)));
       writer.write(stringBuffer.toString());
       writer.close();
       nSubfiles++;
   }
}
```

We can reuse the buffer each time we reverse a file (an example of FIXED ALLOCATION), but we have to generate a new name for each subfile. We also need to count the number of subfiles we have written, so that we can gather then all together again.

```
protected char buffer[] = new char[SubfileSize];

protected String subfileName(int n) {
   return "subxx" + n;
}

protected int nSubfiles = 0;
```

Finally, we need to gather all the subfiles together. Since the subfiles are already reversed, we just need to open each one starting with the last, read its contents, and write them to an output file.

```
protected void gather(BufferedWriter writer) throws IOException {
   for (nSubfiles--; nSubfiles >= 0; nSubfiles--) {
        File subFile = new File(subfileName(nSubfiles));
        BufferedReader reader =
             new BufferedReader(new FileReader(subFile));
        int bytesRead = reader.read(buffer, 0, SubfileSize);
        writer.write(buffer, 0, bytesRead);
        reader.close();
        subFile.delete();
       }
   writer.close();
}
```

❖ ❖ ❖

Known Uses

Most programming languages compile using subfile processing. C, C++, FORTRAN and COBOL programs are all typically compiled one file at a time, and the output object files are then combined with a single link phase after all the compilation phases. C and C++ also force the programmer to manage the 'shared data' for the compilation process in the form of header files [Kernighan and Ritchie 1988]. Java takes the same approach for compiling each separate class file; instead of a link phase Java class files are typically combined into a 'JAR' archive file using COMPRESSION [Chen et al 1998].

The UNIX environment encourages programmers to use data files by providing many simple 'filter' executables: `wc, tee, grep, sed, awk, troff`, for example [Kernighan and Pike 1984]. Programmers can combine these using 'pipes'; the operating system arranges that each filter need only handle a small amount of data at a time.

Most popular applications use data files, and make the names of the current files explicit to the user. Microsoft's MFC framework enshrines this application design in its Document-View architecture [Prosise 1999], supporting multiple documents, where each document normally corresponds to a single data file. EPOC's AppArc architecture [Symbian 1999] supports only one document at a time; depending on the look and feel of the particularly environment this file name may not be visible to the user.

Some word processors and formatters support subfiles – for example Microsoft Word, TeX, and FrameMaker. [Microsoft Word 1997, Lamport 1986, Adobe 1997]. The user can create a *master document* that refers to a series of subdocuments. These subdocuments are edited individually, but when the document is printed each subdocument is loaded into memory and printed in turn. The application need keep only a small amount of global state in memory across subdocuments.

EPOC supports a kind of sub-file within each file, called a stream; each stream is identified using an integer ID and accessed using a simple persistence mechanism. This makes it easy to create many output subfiles and to access each one separately, and many EPOC applications use this feature. Components that use large objects generally persist each one in a separate stream; then they can defer loading each object in until it's actually required – the template class `TSwizzle` provides a MULTIPLE REPRESENTATION to make this invisible to client code [Symbian 1999]. EPOC's relational database creates a new stream for every 12 or so database rows, and for every binary object stored. This makes it easy for the DBMS server to change entries in a database – by writing a new stream to replace an existing one and updating the database's internal index to all the streams [Thoelke 1999].

Printer drivers (especially those embedded in bitmap-based printers) often use 'Banding', where the driver renders and prints only a part of the page at a time. Banding reduces the size of the output bitmap it must store, but also reduces the printing speed, as each page must be rendered several times, once for each band.

See Also

RESOURCE FILES is an alternative for read-only data. PAGING is much simpler for the programmer, though much more complex to implement. DATA FILES make it easier to implement APPLICATION SWITCHING. Each subfile can be stored on SECONDARY STORAGE, using COMPRESSION.

You can use either or both of FIXED ALLOCATION and MEMORY DISCARD to process each item read from a DATA FILE.

PIPES AND FILTERS [Shaw and Garland 1996] describes a software architecture style based around filters.

Rather than a simple Data File, you may need a full-scale database [Connolly and Begg 1999; Date 1999; Elmasri and Navathe 2000]. Wolfgang Keller and Jens Coldewey [1998] provide a set of patterns to store objects from OO programs into relational databases.

Resource Files Pattern

How can you manage lots of configuration data?

- Much program data is *read-only* configuration information and is not modified by the program.
- The configuration data typically changes more frequently than program code.
- Data can be referenced from different phases of the program.
- You only need a few data items at any time.
- File systems support *random access*, which makes it easy to load a portion of a file individually.

Sometimes a program's *memory requirements* include space for a lot of *read-only* static data; typically the program only uses a small amount of this at any one time. For example Word-O-Matic needs static data such as window layouts, icon designs, font metrics and spelling dictionaries. Much of this information may be requested at any arbitrary time within the program, but when requested it is typically needed only for a short time. If the information is stored in main memory – if, for example, you hard-coded it into your program – it will increase the program's overall *memory requirements*.

Furthermore, you may need to change the configuration information separately from the program itself. What may seem to be different variants of the program (for different languages, or with different user interface themes) may use the same code but require different configuration data. Within a given configuration, many data items may be required at any time – window formats or fonts for example, so you cannot use APPLICATION SWITCHING techniques to bring this data in only for a given portion of the time. In general, however, much of the data will not be used at any given time.

Therefore: *Keep configuration data on secondary storage, and load and discard each item as necessary.*

Operating systems offer a simple way to store read-only static data: in a file on secondary storage. File systems provide random access, so it's easy to read just a single portion of a file, ignoring the remainder. You can load a portion of file into temporary memory, use it for a while, then discard it; you can always retrieve it again if you need it. In fact, with only a little additional complexity, you can make a file into a read-only database, containing data items each associated with a unique identifier.

Rather than hard-code each item of data specifically in the program code, you can give each item a unique identifier. When the program requires the data, it invokes a special routine passing the identifier; this routine loads the data from a 'resource file' and returns it to the program. The program may discard the loaded data item when it's no longer required. Typical resources are:

- Strings
- Screen Layouts
- Fonts
- Bitmaps, icons, and cursors.

For example, all of Word-O-Matic's window layouts, icon designs and text strings are stored in resource files. When they are required Word-O-Matic retrieves the data from the resource file, and stores in a temporary memory buffer. The memory can be reused when the data is no longer required.

Consequences

The read-only static data doesn't clutter primary storage, reducing the program's *memory requirements*. Multiple programs can share the same resource file, reducing the *programmer effort* involved. Some operating systems share the loaded resources between multiple instances of the same program or library, further decreasing *memory requirements.* This also makes it easy to change the data without changing the program (e.g. to support multiple language strings), increasing the program's *design quality.*

However: this approach requires *programmer discipline* to place resources into the resource files, and to load and release the resources correctly. Loading and unloading resource files reduces the program's *time performance* somewhat. In particular they can impact its *start-up time*. Resource files also need *programmer effort* to implement, because you need some mechanism to unload (and reload) the resources. It's best if the *operating system* environment provides this support.

❖ ❖ ❖

Implementation

Since resource files are accessed randomly, applications need an index to locate data items (see DATA FILES). Most implementations of resource files hold this index in the resource file itself; typically at the start of the file. However this means that the resource file cannot simply be human-readable text, but must be *compiled.* Resource Compilers also typically convert the resource data into binary formats that can easily used by managed by application code, reducing the memory occupied and improving application performance.

In practice you usually need a logical separation between different resources in a system: the resources for one component are distinct from those for another, and the responsibility of separate teams. Thus most resource file frameworks support more than one resource file at a time.

Here are some things to consider when implementing resource files:

1. Making it easy for the programmer.

The task of loading and interpreting a resource is not a trivial one, so most systems provide library functions. You need basic functions to load and release the raw resources; typically you can also use more sophisticated functions:

- To manage the loading and release of resources, often from multiple files.
- To build graphical dialogs and constructs from the information
- To transfer bitmap and drawing resources (fonts, icons, cursors, drawing primitives) directly from the file to the screen without exposing their structure to the program.
- To insert parameters into the resource strings.

It's not just enough to be able to load an unload resources into systems at runtime; you also have to create the resources in the first place. Programming environments also provide facilities to help you produce resources:

- A Resource compiler – which creates a resource file database from a text file representation.

- Dialog Editors. These allow programmers to 'screen paint' screens and dialogs with user controls; programmers can then create resource-file descriptions from the results. An example is Microsoft Developer Studio (see Figure XX), but there are very many others.

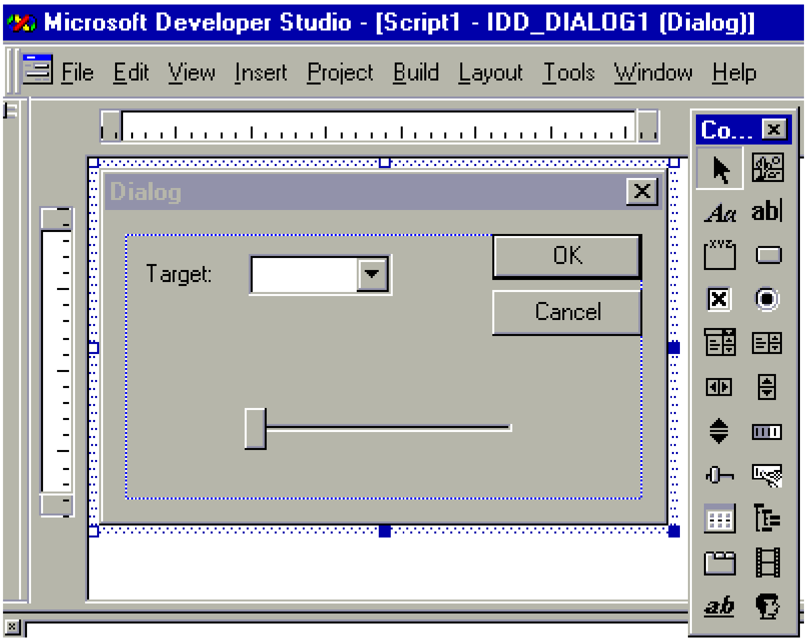

Microsoft Developer Studio - [Script1 - IDD_DIALOG1 (Dialog)]
File Edit View Insert Project Build Layout Tools Window Help
Co...
Dialog
Target:
OK
Cancel

Figure 10: A Dialog Editor

2. Working with Resource Files to Save Memory.

Some resource file systems support compression. This has a small time overhead for each resource loaded, but reduces the file system space taken by the files. ADAPTIVE COMPRESSION algorithms are inappropriate for compression whole files, though, as it must be possible to decode any data item independently of the rest of the file. You can compress individual resources if they are large enough, such as images or sound files

It's worthwhile to take some effort to understand how resource loading works on your specific system as this can often help save memory. For example, Windows also supports two kinds of resource: `PRELOAD` and `LOADONCALL`. Preloaded resources are loaded when the program is first executed; a `LOADONCALL` resource loads only when the user code requests the specific resource. Clearly to save memory, you should prefer `LOADONCALL`. Similarly Windows 3.1 doesn't load strings individually, but only in blocks of 16 strings with consecutive ID numbers. So you can minimise memory use by arranging strings in blocks, such that the strings in a single block are all used together. By way of contrast, the Windows `LoadIcon` function doesn't itself access the resource file; that happens later when a screen driver needs the icon – so calling `LoadIcon` doesn't in itself use much memory. Petzold [1998] discusses the memory use of Windows resource files in more detail.

3. Font Files

You often want to treat font resources very differently from other kinds of resources. For a start, all applications will share the same set of fonts, and font descriptions tend to be much larger than other resources. A sophisticated font handling system will load only portions of each file as required by specific applications: perhaps only the implementation for a specific font size, or only the characters required by the application for a specific string. The last approach is particularly appropriate for fonts for the Unicode characters, which may contain many thousands of images [Pike and Thompson 1993].

4. Implementing a resource file system.

Sometimes you need to implement your own resource file system. Here are some issues to consider:

4.1 Selecting variants. How will the system select which version of the resource files is loaded? There are various options. Some systems include only one resource file with each release. Others (e.g. most MS Windows applications) support variants for languages, but install only one; changing language means overwriting the files. Still other systems select the appropriate variant on program initialisation; for example chooses the variant by file extension (if the current language is number 01, application Word loads resource file `WORD.R01`). Other systems may even permit the system to change its language 'on the fly', although this is bound to require complex interactions between applications.

4.2 Inserting parameters into strings. The most frequent use of resources is in strings. Now displayed strings often contain variable parameters: "You have CC things to do, NN", where the number NN and the name CC vary according to the program needs. How do you insert these parameters?

A common way is to use C's `printf` format: "You have %d things to do, %s". This works reasonably, but has two significant limitations. First, the normal implementation of `printf` and its variants are liable to crash the program if the parameters required by the resource strings are not those passed to the strings. So a corrupt or carelessly constructed resource file can cause unexpected program defects. Second, the `printf` format isn't particularly flexible at supporting different language constructions – a German, for example, might want the two parameters in the other order: "%s: you have %d things to do.".

A more flexible alternative is to use numbered strings in the resource strings: "You have %1 things to do, %2". The program code has responsibility to convert all parameters to strings (which is simple, and can be done in a locale-sensitive way), and a standard function inserts the strings into the resource string. It is a trivial task to implement this function to provide default behaviour or an error message if the number of strings passed doesn't match the resource string.

Examples

Here's an example of an MS Windows resource file for an about box:

```
// About Box Dialog
//

IDD_ABOUTBOX DIALOG DISCARDABLE  34, 22, 217, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About DEMOAPP"
FONT 8, "MS Sans Serif"
BEGIN
    ICON            2,IDC_STATIC,11,17,18,20
    LTEXT           "Demonstration Application by Charles Weir",
                     IDC_STATIC,40,10,79,8
    LTEXT           "Copyright \251 1999",IDC_STATIC,40,25,119,8
    DEFPUSHBUTTON   "OK",IDOK,176,6,32,14,WS_GROUP
END
```

The C++ code to use this using the Microsoft Foundation Classes is remarkably trivial:

```
/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog

CAboutDlg::CAboutDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CAboutDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CAboutDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}
```

Note the explicit syntax of the comment, `{{AFX_DATA_INIT(CAboutDlg)`; this allows other Microsoft tools and 'wizards' to identify the location; the Wizard can determine any variable fields in the dialog box, and insert code to initialise them and to retrieve their values after the dialog has completed. In this case there are no such variables, so no code is present.

❖ ❖ ❖

Known Uses

Virtually all Apple Macintosh and MS Windows GUI programs use resource files to store GUI resources, especially fonts [Apple 1985; Petzold 1998]. EPOC stores all language-dependent information (including compressed help texts) in resource files, and allows the system to select the appropriate language at run-time. EPOC's Unicode font handling minimises the memory use of the font handler with a FIXED-SIZE MEMORY buffer to store a cached set of character images.EPOC16 used compression to reduce the size of its resource files [Edwards 1997].

Many computer games use resource files – from hand-helds with extra static ROM, to early microcomputers backed with cassette tapes and floppies, and state-of-the-art game consoles based on CD-ROMs. The pattern allows them to provide many more screens, levels, or maps than could possibly fit into main memory. Each level is stored as a separate resource in secondary storage, and then loaded when then user reaches that level. Since the user only plays on one level at any time, *memory requirements* are reduced to the storage required for just one level. This works well for arcade-style games where users play one level, then proceed to the next (if they win) or die (if they lose), because the sequence of levels is always predictable. Similarly many of the variations of multi-user adventure games keep the details of specific games: locations, local rules, monsters, weapons as resource files; as they tend to be large, they are often stored COMPRESSED**.**

See Also

DATA FILES provide writable data storage; APPLICATION SWITCHING and PACKAGES do for code what RESOURCE FILES does for unmodifiable data.

Each resource file can be stored in SECONDARY STORAGE or READ-ONLY MEMORY, and may use COMPRESSION..

Petzold [1998] and Microsoft [1997] describe Microsoft Windows Resource files. Tasker et al [2000] describes using EPOC resource files.

Packages

Also known as: Components, Lazy Loading, Dynamic Loading, Code Segmentation.

How can you manage a large program with lots of optional pieces?

- You don't have space in memory for all the code and is static data.
- The system has lots of functionality, but not all will be used simultaneously
- You may require any arbitrary combination of different bits of functionality.
- Development works best when there's a clear separation between developed components.

Some big programs are really small programs much of the time — the *memory requirements* of all the code are much greater the requirements for the code actually used in any given run of the program. For example Strap-It-On's Spin-the-Web^{TM} web browser can view files of many different kinds at once, but it typically reads only the StrapTML local pages used by its help system. Yet the need to support other file types increases the program's code *memory requirements* even when they are not needed.

In these kinds of programs, there is no way of predicting in advance which features you'll need, nor of ordering them so that only one is in use at the same time. So the APPLICATION SWITCHING pattern cannot help, but you still want the benefits of that pattern – that the memory requirements of the system are reduced by not loading all of the program in to main memory at the same time.

Therefore: *Split the program into packages, and load each package only when it's needed.*

Any run-time environment which stores code in disk files must have a mechanism to activate executables loaded from disk. With a relatively small amount of effort, you can extend this mechanism to load additional executable code into a running program. This will only be useful, though, if most program runs do not need to load most of this additional code.

You need to divide the program into a *main program* and a collection of independently loaded *packages*. The main program is loaded and starts running. When it needs to use a facility in a package, a code routine somewhere will load the appropriate package, and call the package directly.

For example, the core of Spin-the-Web is a main program that analyses each web page, and loads the appropriate viewer as a package.

Consequences

The program will *require less memory* because some of its code is stored on SECONDARY STORAGE until needed.

The program will *start up quicker*, as only the small main program needs to be loaded initially, and can begin running with *less memory* than would otherwise be required. Because each package is fairly small, subsequent packages can be loaded in quickly without pauses for changing phases (as would be caused by the APPLICATION SWITCHING pattern).

Because packages aren't statically linked into the application code, dynamic loading mechanisms allow third parties or later developers to add functionality without changing or even stopping the main program. This significantly increases the system's *usability* and *maintainability*.

However: *Programmer effort* is needed to divide up the program into packages.

Many environments never unload packages, so the program's *memory requirements* can steadily increase, and the program can still run out of memory unless any given run uses only a small part of its total functionality. It takes *programmer effort* to implement the dynamic loading mechanism and to make the packages conform to it, and to define the strategy of when to load and unload the packages; or to optimise the package division and minimise the loading overhead. This mechanism can often be reused across programs, or it may be provided by the *operating system;* on the other hand many environments provide no support for dynamic loading.

Because a package isn't loaded until it's required dynamic loading means that the system may not detect a missing package until well after the program has loaded; this slightly reduces the program's *usability*. Also if access to the package is slow (for example, over the Web), the time taken to load a package can reduce the program's responsiveness, which also reduces the program's *usability.* This arbitrary delay also makes PACKAGES unsuitable for *real-time* operations.

Packages can be located remotely and changed independently from the main program. This produces *security* implications – a hostile agent may introduce viruses or security loopholes into the system by changing a package.

❖ ❖ ❖

Implementation

To support packages, you need three things:

1) A system that loads code into RAM to execute it.

2) A partition of the software into packages such that normally only a subset of the packages need be active.

3) Support for dynamically loadable packages — usually position independent or relocatable object code.

Here are some issues to consider when using or implementing packages as part of your system.

1. Processes as Packages

Perhaps the simplest form of a package is just a separate process. With careful programming, two processes that run simultaneously can appear to users as a single process, although there can be a significant cost in performance and program complexity to achieve this. Implementing each package in separate processes has several key advantages:

- The package and main program will execute in separate address spaces, so a fatal error in the package will not necessarily terminate the main program.
- The memory allocated to the package can be discarded easily, simply by terminating the process when the package is no longer needed.
- In some cases, the desired package may already exist as an application in its own right. For example we may want packages to do word-processing, drawing, or spreadsheet managing. Such applications exist already, and are implemented as separate processes.

There are two common approaches to use processes as packages:

1. The client can execute the process in the same way that, say, the operating system shell might do. It runs the process until its complete, perhaps reading its standard output (see DATA FILES)

2. The client can use operating system Inter-Process Communication (IPC) mechanisms to communicate with the process.

This second approach is taken by some forms of the Microsoft ActiveX ('COM') frameworks, by IBM's System Object Model (SOM) and by frameworks based on CORBA [Box 1998; Szyperski 1997; Henning and Vinoski 1999; Egremont 1998]. Each uses some form of PROXY [Gamma et al 1995, Buschmann et al 1996] to give the client access to objects in the package object. The *Essential Distributed Objects Survival Guide* [Orfali 1996] for a discussion and comparison of these environments.

2.Using Dynamically Linked Libraries as C++ Packages

You can also consider using Shared, or Dynamically Linked, Libraries (DLLs) as packages. Normally an executable loads all its DLLs during initialisation, so DLLs do not behave as packages by default. Most environments, however, provide additional mechanisms to load and unload DLLs 'on the fly'.

Some frameworks support delayed loading of DLLs: you can implement Microsoft COM objects, for example, as DLLs that load automatically when each object is first accessed. Although COM's design uses C++ virtual function tables, many other languages have provided bindings to access COM objects [Box 1998].

Other environments simply provide mechanisms to load the DLL file into RAM, and to invoke a function within the DLL. How can you use this to implement PACKAGES?

Typically you can identify their externally callable, *exported*, functions in DLLs either by function name or by function ordinal (the first exported function is ordinal 0, the second, 1, etc.). With either approach it would be quite a task to provide stubs for all the client functions and patch each to the correct location in the DLL.

Instead you can use object-orientation's dynamic binding to provide a simpler solution. This requires just a single call to one DLL entry point (typically at index 0 or 1). This function returns a pointer to a single instance of a class that supports an interface known to the client. From then on the client may call methods on that instance; the language support for dynamic linking ensures that the correct code executes. Typically this class is an ABSTRACT FACTORY or provides FACTORY METHODS [Gamma et al 1995]. Figure 11 shows such a library and the classes it supports.

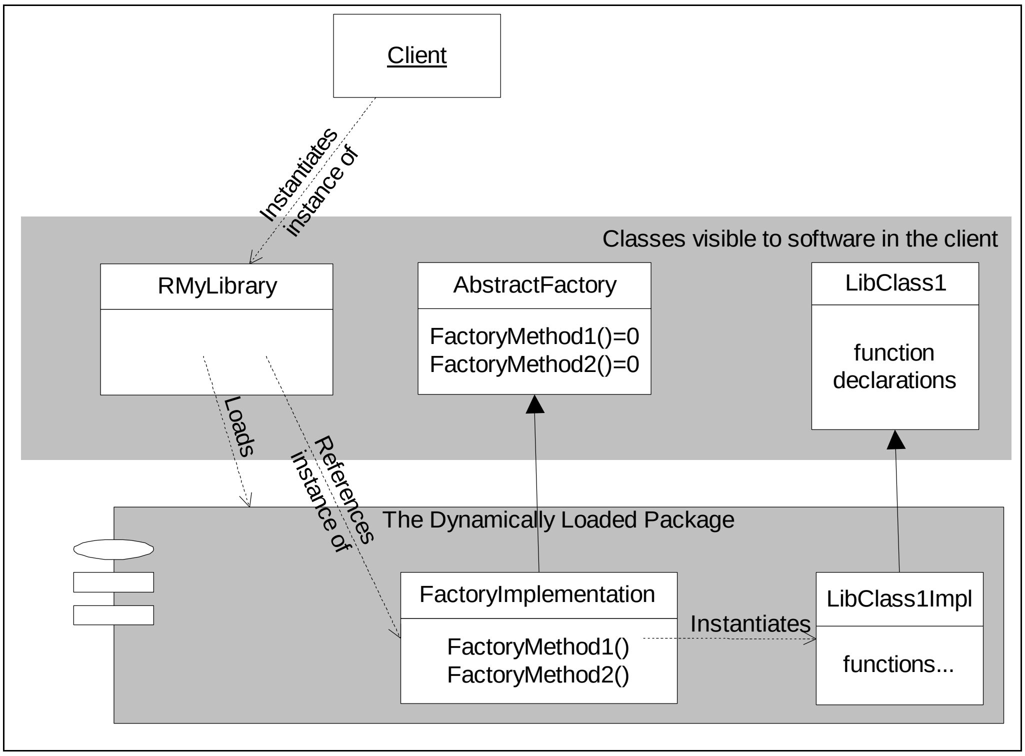


Figure 11: Dynamically loaded package with abstract factory

3. Implementing Packages using Code Segmentation

Many processor architectures and operating systems provide *code segmentation.* This supports packages at the machine code or object code level. A segmented architecture considers a program and the data that it accesses to be made up of some number of independent segments, rather than one monolithic memory space [Tannenbaum 1992].

Typically each segment has its own memory protection attributes — a data segment may be readable and writable by a single process, where a code segment from a shared library could be readable by every process in the system. As with packages, individual segments can be swapped to and from secondary storage by the operating system, either automatically or under programmer control. Linkers for segmented systems produce programs divided up into segments, again either automatically or following directives in the code.

Many older CPUs supported segmentation explicitly, with several *segment registers* to speed-up access to segments, and to ensure that the code and data in segments can be accessed irrespective of the segment's physical memory. Often processor restrictions limited the maximum size of each segment (64K in the 8086 architecture). More modern processor architectures tend to combine SEGMENTATION with PAGING.

4. Loading Packages

If you're not using segmentation or Java packages, you'll have to write some code somewhere in each application to load the packages. There are two standard approaches to where you put this code:

4.1 Manual loading: The client loads the package explicitly. This is best when:

1) The client must identify which it requires of several packages with the same interface. (E.g. loading a printer driver), or

2) The library provides relatively simple functionality, and it's clear when it needs to be unloaded.

4.2 Autoloading: The client calls any function supported by the library. This function is actually a stub provided by the client; when called it loads the library and invokes the appropriate entry point. This is better when:

1) You want a simple interface for the client, or

2) There are many packages with complicated interdependencies, so there's no easy algorithm to decide when to load a package.

Both approaches are common. For example, the Microsoft's COM framework and most EPOC applications do explicit loading; the Emacs text editor does autoloading [Box 1998; Tasker 2000; Stallman 1984].

5. Unloading packages

You'll save most memory if there's a mechanism to unload packages that are no longer required. To do this you need also a way to detect when there is no longer a need for the loaded code. In OO-environments this is easy to decide: the loaded code is no longer needed when there are no instances of objects supported by the package. So you can use REFERENCE COUNTING or GARBAGE COLLECTION to decide when to unload the code.

Loading a package takes time, so some implementations choose to delay unloading packages even when clients notify them that they may do so. Ideally they must unload these cached packages when system memory becomes short – the CAPTAIN OATES PATTERN.

6. Version Control and Binary Compatibility

You need to make sure that each package loaded works correctly with the component that loaded it – even if the two pieces of code were developed and released at different times. This requirement is often called 'binary compatibility', and is distinct from 'source code compatibility'. The requirements of 'binary compatibility' depend both on the language and on the compilation system used, but typically include:

- New versions of the clients expect the same externally visible entry points, parameters and return values; new services support the same ones.
- New clients don't add extra parameter values; new services don't add extra returned values. This is related to the rules for sub-typing – see Meyer [1997].
- New services support the same externally visible state as before.
- New services don't add new exceptions or error conditions unless existing clients have a way to handle them.

The problem of version control can become a major headache in development projects, when teams are developing several packages in parallel. Java, for example, provides no built-in mechanism to ensure that two packages are binary compatible; incompatible versions typically don't fail to load, but instead produce subtle program defects. To solve this problem, some environments provide version control in the libraries. Solaris, for example, supports major and minor version numbers for its DLLs. Minor version changes retain binary compatibility; major ones do not.

Drossopoulou et al [1998] discusses the rules for Java in more detail. [Symbian Knowledgebase 2000] discusses rules for C++ binary compatibility.

7. Optimising Packages

If you are using PACKAGES, you'll only have a fraction of the total code and data in memory at any given time – the *working set*. What techniques can you use to keep this working set to a minimum? You need to ensure that code that is used together is stored in the same package. Unfortunately, although organizing compiled code according to classes and modules is a good start, it doesn't provide an optimum solution. For example each of the many visual objects in the Strap-it-On's Mind-Mapping application have functionality to create themselves from vague text descriptions, to render animated pictures on the screen, to interact in weird and stimulating ways, to save themselves to store and to restore themselves again. Yet a typical operation on a mind-map will use only one of these types of functionality – but in every class (see figure XX).

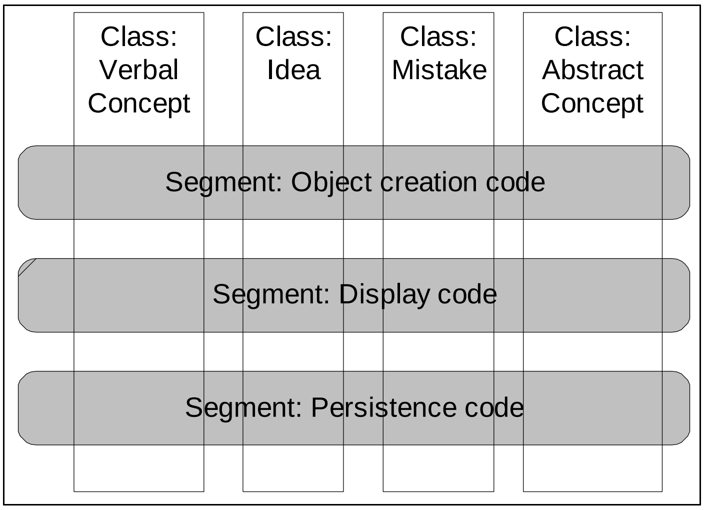


Figure 13: Example - Class divisions don't give appropriate segments

You could reorganise the code so that the compilation units correspond to your desired segments – but the results would be difficult to manage and for programmers to maintain. Using the terminology of Soni et al [1995], the problem is that we must organise the compiled code according to the *execution architecture* of the system, while the source code is organised according to its *conceptual architecture.* Most development environments provide *profilers* that show this execution architecture, so it's possible for programmers to decide a segmentation structure – at the cost of some *programmer effort* – but how should they implement it?

Some compilation environments provide a solution. Microsoft's C++ Compiler and DEC's FORTRAN compiler, for example, allow the user to partition each compilation unit into separate units of a single function, called '`COMDATs`'. Programmers can then order these into appropriate segments using a Link option: `/ORDER:@filename` [Microsoft 1997]. Sun's SparcWorks' `analyzer` tool automates the procedure still further, allowing 'experiments' with different segmentation options using profiling data, and providing a utility (`er_mapgen`) to generate the linker map file directly from these experiments.

For linkers without this option, an alternative is to pre-process the source files to produce a single file for each function, and then to order the resulting files explicitly in the linker command line. This requires additional *programmer discipline,* since it prevents us making code and data local to each source file.

Example

This EPOC C++ example implements the 'Dynamically loaded package with abstract factory' approach illustrated in Figure 11. This component uses animated objects in a virtual reality application. The animated objects are of many kinds (people, machinery, animals, rivers, etc.), only a few types are required at a time, and new implementations will be added later. Thus the implementations of the animated objects live in Packages, and are loaded on demand.

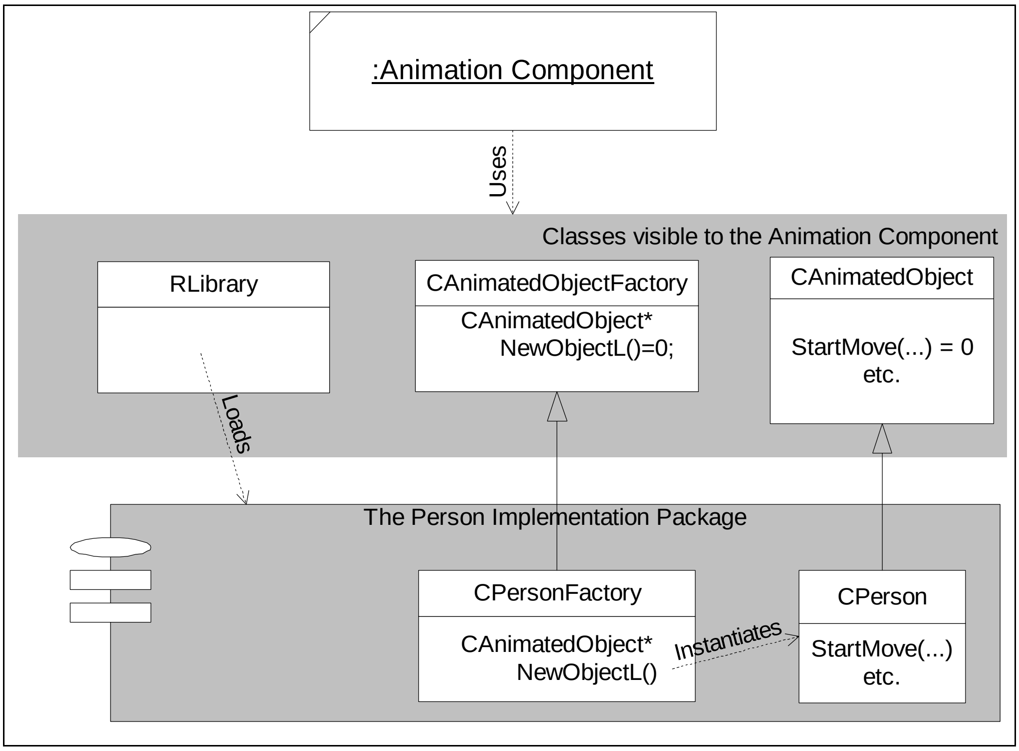


Figure 14: The Example Classes

1. Implementation of the Animation Component

EPOC implements packages as DLLs. The code in the animation component must load the DLL, and keep a handle to if for as long as it has DLL-based objects using its code. It might create a new `CAnimatedObject` using C++ something like the following (where the current object has a longer lifetime than any of the objects in package, and `iAnimatedObjectFactory` is a variable of type `CAnimatedObjectFactory*`)

```
iAnimatedObjectFactory = CreateAnimatedObjectFactoryFromDLL( fileName );
    CAnimatedObject* newAnimatedObject =
        iAnimatedObjectFactory->NewAnimatedObjectL();
```

The implementation of `CreateAnimatedObjectFactoryFromDLL` is as follows. It uses the EPOC class `RLibrary` as a handle to the library; the function `RLibrary::Load` loads the library; `RLibrary::Close` unloads it again. As with all EPOC code, it must implement PARTIAL FAILURE if loading fails. Also `libraryHandle` is a stack variable, so we must ensure it is `Close`'d if any later operations do a PARTIAL FAILURE themselves, using the cleanup stack function, `CleanupClosePushL`.

```
CAnimatedObjectFactory* CreateAnimatedObjectFactoryFromDLL(const TDesC& aFileName)
{
    RLibrary libraryHandle;
    TInt r=libraryHandle.Load(aFileName);
    if (r!=KErrNone)
        User::Leave(r);
    CleanupClosePushL(libraryHandle);
```

We must ensure that the library is the correct one. In EPOC every library (and data file) is identified by three Unique Identifier (UID) integers at the start of the file. The second UID (index 1) specifies the type of file:

```
if(libraryHandle.Type()[1]!=TUid::Uid(KUidAnimationLibraryModuleV01))
        User::Leave(KErrBadLibraryEntryPoint);
```

EPOC DLLs export functions by ordinal rather than by name [Tasker 1999a]. By convention a call to the library entry point at ordinal one returns an instance of the FACTORY OBJECT, `CAnimatedObjectFactory`.

```
    typedef CAnimatedObjectFactory *(*TAnimatedObjectFactoryNewL)();
    TAnimatedObjectFactoryNewL libEntryL=
        reinterpret_cast<TAnimatedObjectFactoryNewL>(libraryHandle.Lookup(1));
    if (libEntryL==NULL)
        User::Leave(KErrBadLibraryEntryPoint);
    CAnimatedObjectFactory *factoryObject=(*libEntryL)();
    CleanupStack::PushL(factoryObject);
```

We'll keep this factory object for the lifetime of the package, so we pass the `RLibrary` handle to its construction function:

```
    factoryObject->ConstructL(libraryHandle);
    CleanupStack::Pop(2); // libraryHandle, factoryObject
    return factoryObject;
}
```

The `CAnimatedObjectFactory` factory object is straightforward. It merely stores the library handle. Like almost all EPOC objects that own resources, it derives from the `CBase` base class, and provides a `ConstructL` function [Tasker et al 2000]. Some of its functions will be called across DLL boundaries; we tell the compiler to generate the extra linkup code using the EPOC `IMPORT_C` and `EXPORT_C` macros.

```
class CAnimatedObjectFactory : public CBase {
public:
    IMPORT_C ~CAnimatedObjectFactory();
    IMPORT_C void ConstructL(RLibrary& aLib);
    IMPORT_C virtual CAnimatedObject * NewAnimatedObjectL()=0;
private:
    RLibrary iLibraryHandle;
};
```

The implementations of the construction function and destructor are simple:

```
EXPORT_C void CAnimatedObjectFactory::ConstructL(RLibrary& aLib) {
    iLibraryHandle = aLib;
}

EXPORT_C CAnimatedObjectFactory::~CAnimatedObjectFactory() {
    iLibraryHandle.Close();
}
```

2. Implementation of the Package

The package itself must implement the entry point to return a new factory object, so it needs a class that derives from `CAnimatedObjectFactory`:

```
class CPersonFactory : public CAnimatedObjectFactory {
public:
    virtual CAnimatedObject * NewAnimatedObjectL();
};

CAnimatedObject * CPersonFactory::NewAnimatedObjectL() {
    return new(ELeave) CPerson;
}
```

The package also needs the class to implement the `CAnimatedObject` object itself:

```
class CPerson : public CAnimatedObject {
public:
    CPerson();
    // etc.
};
```

Finally, the library entry point simply returns a new instance of the concrete factory object (or null, if memory fails). EXPORT_C ensures that this function is a library entry point. In MS C++ we ensure that the function corresponds to ordinal one in the library by editing the 'DEF' file [Microsoft 1997]

```
EXPORT_C CAnimatedObjectFactory * LibEntry() {
    return new CPersonFactory;
}
```

❖ ❖ ❖

Known Uses

Most modern operating systems (UNIX, MS Windows, WinCE, EPOC, etc.) support dynamically linked libraries [Goodheart and Cox 1994, Petzold 1998, Symbian 1999]. Many applications delay the loading of certain DLLs, particularly for *add-ins* – added functionality provided by third parties. Lotus Notes loads viewer DLLs when needed; Netscape and Internet Explorer dynamically load viewers such as Adobe PDF viewer; MS Word loads document converters and uses DLLs for add-in extensions such as support for Web page editing. Some EPOC applications explicitly load packages: the Web application loads drivers for each transport mechanism (HTTP, FTP, etc.) and viewers for each data type.

Printer drivers are often implemented as packages. This allows you to add new printer drivers without restarting any applications. All EPOC applications dynamically load printer drivers where necessary. MS Windows 95 and NT do the same.

Many Lisp systems use dynamic loading. GNU Emacs, for example, consists of a core text editor package plus auto-loading facilities. Most of the interesting features of GNU Emacs exist as packages: intelligent language support, spelling checkers, email packages, web browsers, terminal emulators, etc [Stallman 1984].

Java makes great use of dynamic loading. Java loads each class only when it needs it, so each class is effectively a package. Java implementations may discard classes once they don't need them any more, using garbage collection, although many environments currently do not. Java applets are also treated as dynamically loading packages by Web browsers. A browser loads and run applets on pages it is displaying, and then stops and unloads applets when their containing pages are no longer displayed. [Lindholm and Yellin 1999]. The Palm Spotless JVM loads almost all classes dynamically, even those like String that are really part of the Java Language [Taivalsaari et al 1999].

Many earlier processors supported segmentation explicitly in their architecture. The 8086 and PDP-11 processors both implement segment registers. Programmers working in these environments often had to be acutely aware of the limitations imposed by fixed segment sizes; MS Windows 1, 2 and 3 all reflected the segmented architecture explicitly in the programming interfaces [Hamacher 1984; Chappell 1994].

See Also

APPLICATION SWITCHING is a simpler alternative to this pattern, which is applicable when the task divides into independent phases**.** PAGING is a more complex alternative. Unloaded packages can live on SECONDARY STORAGE, and maybe use COMPRESSION.

ABSTRACT FACTORY provides a good implementation mechanism to separate the client interfaces from the package implementations. VIRTUAL PROXIES can be used to autoload individual packages [Gamma et al 1995]. You may need REFERENCE COUNTING or GARBAGE COLLECTION to decide when to unload a package.

Coplien's *Advanced C++ Programming Styles and Idioms* [1994] describes dynamically loading C++ functions into a running program.

Paging Pattern

Also known as: Virtual Memory, Persistence, Backing Store, Paging OO DBMS.

How can you provide the illusion of infinite memory?[1]

- The *memory requirements* for the programs code and data are too big to fit into RAM.
- The program needs *random access* to all it's code and data.
- You have a fast *secondary storage* device, which can store the code and data not currently in use.
- To decrease *programmer effort* and *usability,* programmers and users should not be aware that the program is using secondary storage.

Some systems' *memory requirements* are simply too large to fit into the available memory. Perhaps one program's data structures are larger than the system's RAM memory, or perhaps a whole system cannot fit into main memory, although each individual component is small enough on its own**.**

For example, the Strap-It-On's weather prediction system, Rain-Sight^{TM}, loads a relatively small amount of weather information from it's radio network link, and attempts to calculate whether the user is about to get rained on. To do that, it needs to work with some very large matrices indeed – larger than can fit in memory even if no other applications were present at all. So the Rain-Sight marketing team have already agreed to distribute a 5 Gb 'coin-disk' pack with every copy of the program, ample for the Rain-Sight data. The problem facing the Rain-Sight developers is how to use it.

You can manage data on secondary storage explicitly using a DATA FILE. This has two disadvantages.

- The resulting code needs to combine processing the data with shuffling it between primary and secondary storage. The result will be *complex* and *difficult to maintain,* costing *programmer effort* to implement and *programmer discipline* to use correctly, because programmers will have to understand both domain-specific requirements of the program, and the fine points of data access.
- In addition this approach will tend to be *inefficient* for *random access* to data. If you read each item each time it's accessed, and write it back after manipulating it, this will require a lot of slow secondary storage access.

Other techniques, such as COMPRESSION and PACKED DATA, will certainly reduce the RAM memory requirements, but can only achieve a finite reduction; ultimately any system can have more data than will fit into RAM.

Therefore: *Keep a system's code and data on secondary storage, and move them to and from main memory as required.*

No software is completely random in its access to memory; at any given time a typical system will be working with only a small subset of the code and data it has available. So you need to

[1] *Sometimes the program is just too big, to complex, or you are too lazy to segment, subdivide, chain, phase, slice, dice, vitamise, or food process the code any more. Why* ***should*** *programmers have to worry about memory! Infinite memory for all is a* ***right****, not a privilege! Those small memory guys are just* ***no-life-losers****!*

keep only a relatively small *working set* in memory; the rest of the system can stay on secondary storage. The software can access this working set very fast, since it's in main memory. If you need to access information on secondary storage, you must change the working set, reading the new data required, and writing or discarding any data that occupied that space beforehand.

You must ensure that the working set changes only slowly – that the software exhibits *locality of reference*, and tends to access the same objects or memory area in preference to completely random access. It helps that memory allocators will typically put items allocated together in the same area of memory. So objects allocated together will typically be physically near to each other in memory, particularly when you're using FIXED ALLOCATION.

There are three forms of this pattern in use today [Tannenbaum 1992, Goodheart 1994]:

Demand Paging is the most familiar form. The memory management hardware, or interpreter environment, implements *virtual memory* so that there is an additional *page table* that maps addresses used in software to *pages* of physical memory. When software attempts to access a memory location without an entry in the page table, the environment frees up a page by saving its data, and loads the new data from secondary storage into physical memory, before returning the address of the physical memory found.

Swapping is a simpler alternative to paging, where the environment stops a process, and writes out all its data to Secondary Storage. When the process needs to process an event, the environment reloads all the data from secondary storage and resumes. This approach is common on portable PCs, where the entire environment is saved to disk, though the intent there is to save power rather than memory.

Object Oriented Databases are similar to Demand Paging, but the unit of paged memory is an object and its associated owned objects (or perhaps, for efficiency, a cluster of such objects). This approach requires more programmer effort than demand paging, but makes the data persistent, and allows multiple processes to share objects.

So, for example, the Rain-Sight team decided to use Paging to make use of their disk. The Strap-OS operating system doesn't support hardware-based paging, so the team hacked a Java interpreter to implement paging for each Java object. The team then defined objects to implement each related part of the Rain-Sight matrices (which are always accessed together), giving them acceptable performance and an apparent memory space limited only by the size of the 'coin disk'.

Consequences

Paging is the ultimate escape of the memory-challenged programmer. The programmer is **much less** aware of paging than any other technique, since paging provides the illusion of essentially infinite memory — the program's *memory requirements* are no longer a problem. So paging tends to increase other aspects of a system's *design quality*, and *maintainability* because memory requirements are no longer an overriding issue.

Paging needs little *programmer effort* and *programmer discipline* to use, because it doesn't need a logical decomposition of the program. Because paging does not require any artificial division of programs into phases or data into files it can make systems more *usable*. Programs using paging can easily accommodate more memory by just paging less, so paging improves *scalability*, as well.

Paging can make good *local* use of the available memory where the program's memory use is distributed *globally* over many different components, since different components will typically use their data at different times.

However: Paging reduces a program's *time performance*, since some memory accesses require secondary storage reads and writes. It also reduces the predictability of response times, making it unsuitable for *real-time systems.* Paging performs badly if the memory accesses do not exhibit locality of reference, and this may require *programmer effort* to fix.

Paging needs fast secondary storage to perform well. Of course 'fast' is a relative term; lots of systems have used floppy disks for paging. Because paging tends to make lots of small data transfers rather than a few large ones, the latency of the secondary storage device is usually more important than its throughput. Furthermore, PAGING's continuous use of secondary storage devices increases the system's *power consumption*, and reduces the lifetime of storage media such as flash RAM and floppy disks.

Since paging doesn't require *programmer discipline,* a program's *memory requirements* can tend to increase in paged systems, requiring more secondary storage and impacting the program's *time performance*. Paging requires no *local* support from within programs, but requires low-level *global* support, often provided by the *hardware and operating system*, or an interpreter or data manager. Because intermediate information can be paged out to secondary storage, paging can affect the *security* of a system unless the secondary storage is as well protected as the primary storage.

❖ ❖ ❖

Implementation

Paging is typically supported by two main data structures, see Figure XXX.

P*age frames* live in main memory, and contain the 'paged in' RAM data for the program. Each page frame also has control information: the secondary storage location corresponding to the current data and a *dirty bit*, set when the page memory has been changed since loading from secondary storage.

The *page table* also lives in main memory, and has an entry for each page on secondary storage. It stores that page is resident in memory, and if so, in which page frame it is stored. Figure XXX below shows a Page Table and Page Frames.

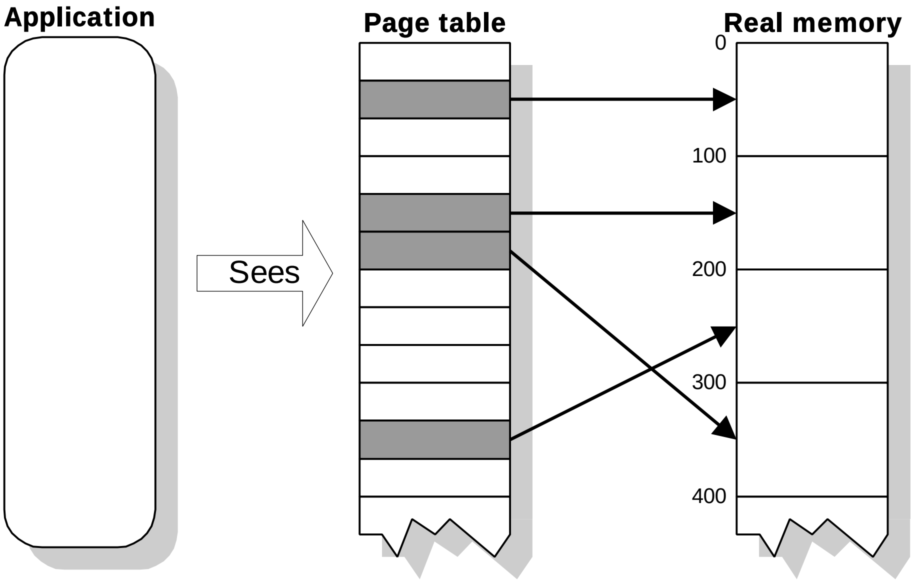


Figure 15: Page Table and Page Frames

As you run a paging application, it accesses memory via the page table. Memory that is paged in can be read and written directly: writing to a page should set the page's dirty bit. When you try to access a page that is 'paged out ' (not in main memory) the system must load the page from secondary storage, perhaps saving an existing page in memory back to secondary storage to make room for the new page. Trying to access a page in secondary storage page is called a *page fault*. To handle a page fault, or to allocate a new page, the system must find a free page frame for the data. Normally, the frame chosen will already contain active data, which must be discarded, and if the dirty bit is set, the system must write the contents out to secondary storage. Once the new frame is allocated, or its contents are loaded from secondary storage, the page table can be updated and the program's execution continue.

Here are some issues to consider when implementing paging.

1. Intercepting Memory Accesses

Probably the single most difficult part of implementing paging is the need to intercept memory accesses. In addition, this intercept must distinguish access for writing, which must set the 'dirty bit', from access for read, which doesn't.

There are several possible mechanisms:

MMU	Many modern systems have a Memory Management Unit (MMU) in addition to the Central Processing Unit (CPU). These provide a set of *virtual memory maps (*typically one for each process), which map the memory locations requested by the code to different real memory addresses. If the program accesses an address that hasn't been loaded, this causes a page fault interrupt, and the interrupt driver will load the page from secondary storage. The MMU also distinguishes pages as read-only and read-write. An attempt to write to a read-only page also causes a page-fault interrupt, which makes it easy to set the dirty bit for that page.

Interpreter	It's fairly straightforward to implement paging for interpreted environments. The run-time interpreter must implement any accesses to the program or its data, so it is relatively easy to intercept accesses and to distinguish reads from writes.
Process Swap	When you swap entire processes, you don't need to detect memory access as processes are not running when they are swapped out.
Data Manager	For programs in an environment with no built-in paging, we can use 'smart pointers' to classes to intercept each access to an object. Then a *data manager* can ensure that the object is in store and to manage loading, caching and swapping. In this case it's appropriate to page entire objects in and out, rather than arbitrarily sized pages.

2. Page Replacement

How can you select which page frame to choose to free up to take a new or loaded page? The best algorithm is to remove the page that will be needed the furthest into the future —the least important page for the system's immediate needs [Tannenbaum 1992]. Unfortunately this is usually impossible to implement, so instead you have to guess the future on the basis of the recent past. Removing the *least frequently used (LFU)* page provides the most accurate estimation, but is quite difficult to implement. Almost as effective but easier to implement is a *least recently used (LRU)* algorithm, which simply requires keeping a list of all page frames, and moving each page to the top of this list as it is used. Choosing a page to replace at random is easy to implement and provides sufficient performance for many situations.

Most implementations of MMU paging incorporate Segmentation techniques as well (see PACKAGES). Since you already have the process's virtual data memory split into pages, it's an obvious extension to do the same for code. Code is READ-ONLY, and typically needs only very trivial changes when it's loaded from secondary storage to memory. So there's no point in wasting space in the swap file; you can take the code pages directly from the code file when you want them and discard them when they're no longer needed.

3. Working Set Size

A program's *working set size* is the minimum amount of memory it needs to run without excessive page faults. Generally, the larger the page size, the larger the working set size. A program's working set size determines whether it will run well under any given paging system. If the working set size is larger than the real memory allocated to page frames, then there will be an excessive number of page faults. The system will start *thrashing*, spending its time swapping pages from main memory to secondary storage and back but making little progress executing the software.

To avoid thrashing, do less with your program, add real memory to the system, or optimise the program's memory layout using the techniques discussed in the PACKAGES PATTERN.

4. Program Control of Paging

Some programs do not have *locality of reference*. For example a program might traverse all its data reading each page exactly once in order. In this case, each page will be paged in only once, and the best page to replace will be the *most* frequently or recently used. To help in such a case, some systems provide an interface so that the programmer can control the paging system. For example, the interface might support a request that a particular page be paged out.

Alternatively it might allow a request that a particular page be *paged in* — for example the program above will know which page will be needed *next* even before processing the current page.

Other program code may have real-time constraints. Device drivers, for example, must typically respond to interrupt events within microseconds. So device driver data must not be paged out. Most systems support this by 'tagging' certain areas of code as with different attributes. For example, Microsoft's Portable Executable Format supports Pre-load and Memory-resident options for its 'virtual device driver' executable files [Microsoft 1997]

Example

The following code implements a simple framework to page individual C++ objects.

It's very difficult indeed to intercept all references to a C++ object without operating system support – in particular we can't intercept the '`this`' pointer in a member function. So it's a bad idea to page instances of any C++ class with member functions. Instead we make each object store its data in a separate data structure and access that structure through special member functions that control paging. The object itself acts as a PROXY for the data, storing a page number rather than a pointer to the data. Figure XX below shows a typical scenario:

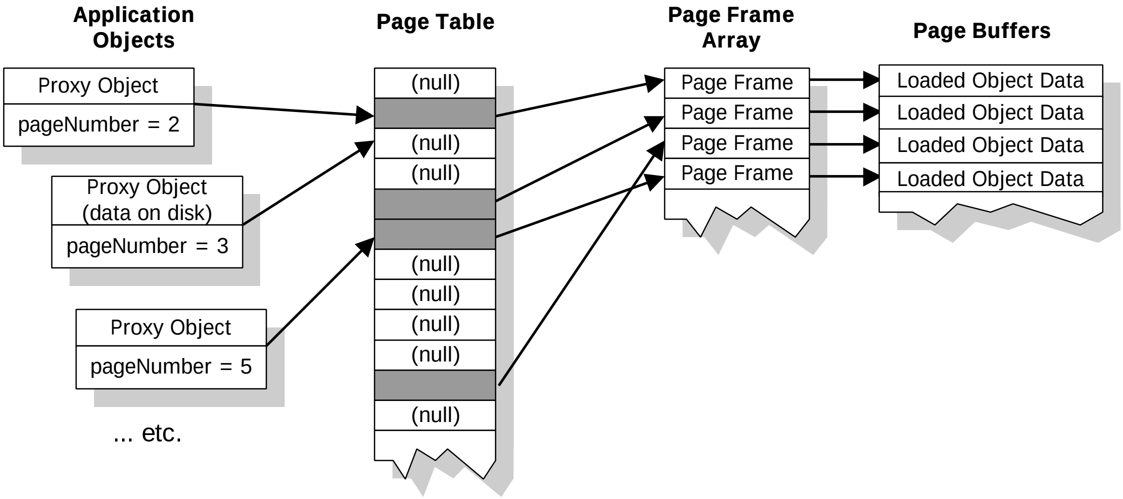


Figure 16: Objects in memory for the Paging Example

The Page Table optimises access to the data in RAM: if its entry is non-null for a particular page, that page is loaded in RAM and the application object can access its data directly. If an application object tries to access a page with a null Page Table entry, it means that object's data isn't loaded. In that case the paging code will save or discard an existing page frame and load that object's data from disk.

1. Example Client Implementation

Here's an example client implementation that uses the paging example. It's a simple bitmap image containing just pixels. Other paged data structures could contain any other C++ primitive data types or `structs`, including pointers to objects (though not pointers to the paged data structure instances, of course, as these will be paged out).

```
typedef char Pixel;

class BitmapImageData {
    friend class BitmapImage;
    Pixel pixels[SCREEN_HEIGHT * SCREEN_WIDTH];
};
```

The PROXY class, `BitmapImage`, derives its paging functionality from the generic `ObjectWithPagedData`. The main constraint on its implementation is that all accesses to the data object must be through the base class `GetPagedData` functions, which ensure that the data is paged into RAM. It accesses these through functions to cast these to the correct type:

```
class BitmapImage : public ObjectWithPagedData {
private:
    BitmapImageData* GetData()
        { return static_cast<BitmapImageData*>(GetPagedData()); }
    const BitmapImageData* GetData() const
        { return static_cast<const BitmapImageData*>(GetPagedData()); }
```

The constructor must specify the `PageFile` object and initialise the data structure. Note that all these functions can be inline:

```
public:
    BitmapImage( PageFile& thePageFile )
        : ObjectWithPagedData(thePageFile) {
        memset( GetData(), 0, sizeof(BitmapImageData) );
    }
```

And all functions use the `GetData` functions to access the data. Note how the C++ `const`-correctness ensures that we get the correct version of the data function; non-`const` accesses to `GetData`() will set the ‘dirty bit’ for the page so it gets written back to file when paged out.

```
    Pixel GetPixel(int pixelNumber) const {
        return GetData()->pixels[pixelNumber];
    }
    void SetPixel(int pixelNumber, Pixel newValue) {
        GetData()->pixels[pixelNumber] = newValue;
    }
};
```

And that’s the full client implementation. Simple, isn’t it?

To use it we need to set up a page file – here’s one with just four page buffers:

```
PageFile pageFile("testfile.dat", sizeof( BitmapImageData ), 4 );
```

And then we can use can use `BitmapImage` as any other C++ object:

```
BitmapImage* newImage = new BitmapImage(pageFile);
newImage->SetPixel(0, 0);
delete newImage;
```

2. Overview of the Paging Framework

Figure XXX below shows the logical structure of the Paging Framework using UML notation [Fowler and Scott 1997]. The names in normal type are classes in the framework; the others are implemented as follows:

- *Page Table Entry* is a entry in the `pageTable` pointer array.
- *Page in RAM* is a simple (void*) buffer.
- *Page on Disk* is a fixed page in the disk file.
- *Client Implementation* is any client class, such as the BitmapImage class above.

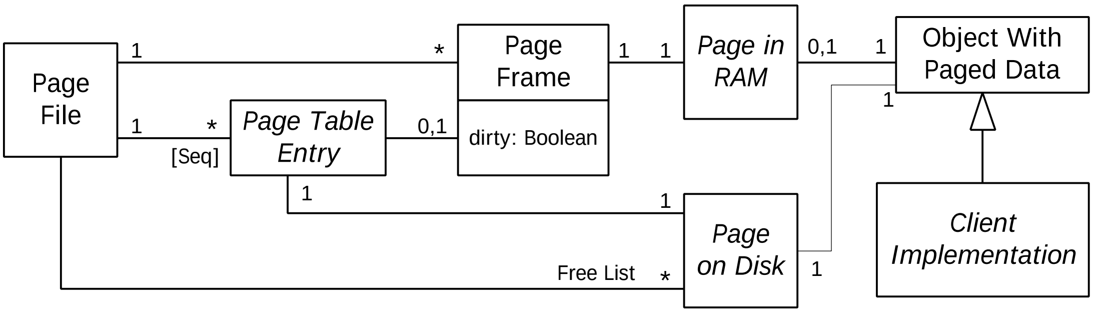


Figure 17: UML Diagram: Logical structure of the object paging system

The page frames and page table are a FIXED DATA STRUCTURE, always occupying the same memory in RAM.

3. Implementation of `ObjectWithPagedData`

ObjectWithPagedData is the base class for the Client implementation classes. It contains only the page number for the data, plus a reference to the PageFile object. This allows us to have several different types of client object being paged independently.

```
class ObjectWithPagedData {
private:
    PageFile& pageFile;
    const int pageNumber;
```

All of its member operations are `protected`, since they're used only by the client implementations. The constructor and destructor use functions in `PageFile` to allocated and free a data page:

```
ObjectWithPagedData::ObjectWithPagedData(PageFile& thePageFile)
    : pageFile( thePageFile ),
      pageNumber( thePageFile.NewPage() )
{}

ObjectWithPagedData::~ObjectWithPagedData() {
    pageFile.DeletePage(pageNumber);
}
```

We need both `const` and non-`const` functions to access the paged data. Each ensures there's a page frame present, then accesses the buffer; the non-`const` version uses the function that sets the dirty flag for the page frame:

```
const void* ObjectWithPagedData::GetPagedData() const {
    PageFrame* frame = pageFile.FindPageFrameForPage( pageNumber );
    return frame->GetConstPage();
}

void* ObjectWithPagedData::GetPagedData() {
    PageFrame* frame = pageFile.FindPageFrameForPage( pageNumber );
    return frame->GetWritablePage();
}
```

3. Implementation of `PageFrame`

A `PageFrame` object represents a single buffer of 'real memory'. If the page frame is 'In Use', a client object has accessed the buffer and it hasn't been saved to disk or discarded. The member `currentPageNumber` is set to the appropriate page if the frame is In Use, or else to `INVALID_PAGE_NUMBER`. `PageFrame` stores the 'dirty' flag for the buffer, and sets it when any client accesses the `GetWritablePage` function.

```
class PageFrame {
    friend class PageFile;

private:
    enum { INVALID_PAGE_NUMBER = -1 };
    bool  dirtyFlag;
    int   currentPageNumber;
    void* bufferContainingCurrentPage;
```

The constructor and destructor simply initialise the members appropriately:

```
PageFrame::PageFrame( int pageSize )
    : bufferContainingCurrentPage( new char[pageSize] ),
      dirtyFlag( false ),
      currentPageNumber(PageFrame::INVALID_PAGE_NUMBER)
{}

PageFrame::~PageFrame() {
    delete [] (char*)(bufferContainingCurrentPage);
}
```

`GetConstPage` and `GetWritablePage` provide access to the buffer:

```
const void* PageFrame::GetConstPage() {
    return bufferContainingCurrentPage;
}

void* PageFrame::GetWritablePage() {
    dirtyFlag = true;
    return bufferContainingCurrentPage;
}
```

And the other two member functions are trivial too:

```
int PageFrame::PageNumber() {
    return currentPageNumber;
}
bool PageFrame::InUse() {
    return currentPageNumber != INVALID_PAGE_NUMBER;
}
```

4. Implementation of `PageFile`

The `PageFile` object manages all of the important behaviour of the paging system. It owns the temporary file, and implements the functions to swap data buffers to and from it.

`PageFile`'s main structures are as follows:

`pageTable` is a vector, with an entry for each page in the page file. These entries are `null` if the page is swapped to secondary storage, or point to a `PageFrame` object is the page is in RAM.

`pageFrameArray` contains all the `PageFrame` objects. It's an array to make it easy to select one at random to discard.

`listOfFreePageNumbers` contains a queue of pages that have been deleted. We cannot remove pages from the page file, so instead we remember the page numbers to reassign when required.

So the resulting private data is as follows:

```
class PageFile {
    friend class ObjectWithPagedData;
private:
    vector<PageFrame*> pageTable;
    vector<PageFrame*> pageFrameArray;
    list<int> listOfFreePageNumbers;
    const int pageSize;
    fstream fileStream;
```

`PageFile`'s constructor must initialise the file and allocate all the FIXED DATA STRUCTURES. It requires a way to abort if the file open fails; this example simply uses a variant of the ASSERT macro to check:

```
PageFile::PageFile( char* fileName, int pageSizeInBytes, int nPagesInCache )
    : fileStream( fileName,
                  ios::in| ios::out | ios::binary | ios::trunc),
      pageSize( pageSizeInBytes ) {
    ASSERT_ALWAYS( fileStream.good() );
    for (int i = 0; i<nPagesInCache; i++) {
        pageFrameArray.push_back( new PageFrame( pageSize ) );
    }
}
```

The destructor tidies up memory and closes the file. A complete implementation would delete the file as well:

```
PageFile::~PageFile() {
    for (vector<PageFrame*>::iterator i = pageFrameArray.begin();
         i != pageFrameArray.end(); i++ )
        delete *i;
    fileStream.close();
}
```

The function `NewPage` allocates a page on disk for a new client object. It uses a free page on disk if there is one, or else allocates a new `pageTable` entry and expands the page file by writing a page of random data to the end.

```
int PageFile::NewPage() {
    int pageNumber;
    if (!listOfFreePageNumbers.empty()) {
        pageNumber = listOfFreePageNumbers.front();
        listOfFreePageNumbers.pop_front();
    } else {
        pageNumber = pageTable.size();
        pageTable.push_back( 0 );
        int newPos = fileStream.rdbuf()->pubseekoff( 0, ios::end );
        fileStream.write(
            (char*)pageFrameArray[0]->bufferContainingCurrentPage,
            PageSize() );
    }
    return pageNumber;
}
```

The corresponding `DeletePage` function is trivial:

```
void PageFile::DeletePage(int pageNumber) {
    listOfFreePageNumbers.push_front(pageNumber);
}
```

The function `FindPageFrameForPage` assigns a `PageFrame` for the given page number and ensures that the page is in RAM. If there's already a `PageFrame` for the page, it just returns the pointer; otherwise it finds a `PageFrame` and fills it with the requested page from disk.

```
PageFrame* PageFile::FindPageFrameForPage( int pageNumber ) {
    PageFrame* frame = pageTable[pageNumber];
    if (frame == 0) {
        frame = MakeFrameAvailable();
        LoadFrame( frame, pageNumber );
        pageTable[pageNumber] = frame;
    }
    return frame;
}
```

The function `MakeFrameAvailable` assigns a frame by paging out or discarding an existing page, chosen at random.

```
PageFrame* PageFile::MakeFrameAvailable() {
    PageFrame* frame = pageFrameArray[ (rand() * pageFrameArray.size()) /
                                       RAND_MAX ];
    if (frame->InUse()) {
        SaveOrDiscardFrame( frame );
    }
    return frame;
}
```

The function that provides the meat of the paging algorithm is `SaveOrDiscardFrame`. This writes out the page to the corresponding location in file – if necessary – and resets the page table entry.

```
void PageFile::SaveOrDiscardFrame( PageFrame* frame ) {
    if (frame->dirtyFlag) {
        int newPos = fileStream.rdbuf()->pubseekoff( frame->PageNumber() *
                                                     PageSize(), ios::beg );
        fileStream.write( (char*)frame->bufferContainingCurrentPage,
                          PageSize() );
        frame->dirtyFlag = false;
    }
    pageTable[frame->PageNumber()] = 0;
    frame->currentPageNumber = PageFrame::INVALID_PAGE_NUMBER;
}
```

And finally, the corresponding function to load a frame is as follows:

```
void PageFile::LoadFrame( PageFrame* frame, int pageNumber ) {
    int newPos = fileStream.rdbuf()->pubseekoff( pageNumber * PageSize(),
                                                 ios::beg );
    fileStream.read( (char*)frame->bufferContainingCurrentPage, PageSize() );
    frame->currentPageNumber = pageNumber;
 }
```

❖ ❖ ❖

Known Uses

Almost every modern disk operating system provides paged virtual memory, including most versions of UNIX including LINUX, Mac OS, and MS Windows [Goodheart 1994; Card, Dumas, Mével 1998, Microsoft 1997]

OO Databases almost all use some form of object paging. ObjectStore uses the UNIX (or NT) paging support directly, but replaces the OS-supplied paging drivers with drivers that suit the needs of OO programs with persistent data [Chaudhri 1997].

Infocom games implemented a paged interpreter on machines like Apple-IIs and early PCs, paging main memory to floppy disks [Blank and Galley 1995]. This enabled games to run on machines with varying sizes of memory — although of course games would run slower if there was less main memory available. The LOOM system implemented paging in Smalltalk for Smalltalk [Kaehler and Krasner 1983].

See Also

The other patterns in this chapter — PROCESSES, DATA FILES, PACKAGES, and RESOURCE FILES— provide alternatives to this pattern. Paging can also use COPY ON WRITE to optimise access to *read-only* storage, and can be extended to support SHARING. System Memory is a *global* resource, so some operating systems implement CAPTAIN OATES, discarding segments from different processes rather than from the process that requests a new page.

An INTERPRETER [Gamma et al 1995] can make PAGING transparent to user programs. VIRTUAL PROXIES and BRIDGES [Gamma et al 1995], and ENVELOPE/LETTER or HANDLE/BODY [Coplien 1994] can provide paging for objects without affecting the objects' client interfaces.

Major Technique: Compression

Version 14/06/00 16:34 - Charles Weir 7

How can you fit a quart of data into a pint pot of memory?

- The *memory requirements* of the code and data appear greater than the memory available, whether primary memory, secondary storage, read-only memory or some combination of these
- You cannot reduce the functionality and omit some of the data or code
- You need to transmit information across a communications link as *quickly* as possible.
- You cannot choose **SMALL DATA STRUCTURES** to reduce the memory requirements further

Sometimes you just don't have enough memory to go around. The most usual problem is that you need to store more data than the space available, but sometimes the executable code can be too large. You can often choose suitable **DATA STRUCTURES** to ensure that the right amount of memory is allocated to store the data; you can also use **SECONDARY STORAGE** and **READ-ONLY STORAGE** move the data out of RAM. These techniques have one important limitation, however: they don't reduce the total amount of storage, of all kinds, needed to support the whole system.

For example, the Strap-It-On wrist-mounted PC needs to store the data for the documents the user is working on. It also needs sound files recorded by the internal microphone, data traces from optional body well-being monitors, and a large amount of executable code downloaded by the user to support "optional applications" (typically Tetris, Doom and Hunt-the-Wumpus, but sometimes work-related programs as well!). This information can certainly exceed the capacity of the Strap-It-On's primary memory and secondary storage combined. How can we improve the Strap-It-On's usability without forcing every user to carry around the optional 2 Gb disk back-pack?

No matter how much memory such a system may have, you will always find users who need more. Extra storage is expensive, so you should use what you have as effectively as possible.

Therefore: *Use a compressed representation to reduce the memory required.*

Store the information in a compressed form and decompress it when you need to access it. There are a wide variety of compression algorithms and approaches you can choose from, each with different space and time trade-offs.

So, for example, the Strap-It-On PC stores its voice sound files using GSM compression; its music uses MP3; its data traces use **DIFFERENCE COMPRESSION**; its databases use **TABLE COMPRESSION**; and its documents are stored using GZIP. The device drivers for Strap-It-On's secondary storage choose the appropriate **ADAPTIVE COMPRESSION** technique based on the file type, ensuring all files are stored in a compressed form.

Consequences

The *memory requirements* of your system decrease because compressed code and data need less space than uncompressed code or data. Some forms of *time performance* may also improve – for example, reading from slow secondary storage devices or over a network.

However: Compressed information is often more difficult to process from within the program. Some compression techniques prevent random access to the compressed information. You may have to decompress an entire data stream to access any part of it – requiring enough *main* *memory* to store all the decompressed information, in addition to the *memory* needed for the decompression itself.

The program has to provide compression and decompression support, making it *more complex to maintain*, requiring a fair amount of *programmer effort* to implement, increasing the *testing cost* of the program and reducing the *realtime responsiveness*.

The compression process also *takes time* and *extra temporary memory* increasing the possibilities for failure; compression can also increase a program's *power consumption*. In some cases – program code, resource file data, and information received via telecommunications – the compression cost may be paid once by large powerful machines better able to handle it. The amount of memory required to store a given amount of data becomes *less predictable*, because it depends upon well the data can be compressed.

❖ ❖ ❖

Implementation

The key idea behind compression is that most data contains a large amount of *redundancy* — information that is not strictly required [Bell, Cleary, Whitten, 1990]. The following sections explore several types of redundancy, and discuss compression techniques to exploit each type.

1. Mechanical Redundancy

Consider the ASCII character set. ASCII defines around 100 printable characters, yet most text formats use eight, sixteen, or even thirty-two bits to store characters for processing on modern processors. You can store ASCII text using just seven bits per character; this would reduce memory used at a cost of increased processing time, because most processors handle eight or thirty-two bit quantities much more easily than seven bit quantities. Thus, 1 bit in a single byte encoding, or 9 bits in a sixteen bit UNICODE encoding are redundant. This kind of redundancy is called mechanical redundancy.

For text compression, the amount of compression is usually expressed by the number of (compressed) bits required per character in a larger text. For example, storing ASCII characters in seven bit-bytes would give a compression of 7 bits per character. For other forms of data we talk about the *compression ratio* – the compressed size divided by the decompressed size. Using 7 bit ASCII to encode a normal 8-bit ASCII file would give a compression ratio of 7/8, or 87.5%.

TABLE COMPRESSION and **SEQUENCE CODING** explore other related forms of mechanical redundancy.

2. Semantic Redundancy

Consider the traditional English song:

Verse 1:
Voice: Whear 'as tha been sin' I saw thee?
Reply: I saw thee
Chorus: On Ilkley Moor Bah t'at
Voice: Whear 'as tha been sin' I saw thee?
Reply: I saw thee
Voice: Whear 'as tha been sin' I saw thee?

Chorus: On Ilkley Moor Bah t'at
Reply: Bah t'at
Chorus: On Ilkley Moor Bah t'at
On Ilkley Moor Bah t'at

Verse 2:
Voice: Tha's been a-coortin' Mary Jane
Reply: Mary Jane
Chorus: On Ilkley Moor Bah t'at

… etc., for 7 more verses.

This song has plenty of redundancy because of all the repeats and choruses; you don't need to store every single word sung to reproduce the song. The songbook '*Rise up singing*' [Blood and Paterson 1992] uses bold type, parentheses and repetition marks to compress the complete song to 15 short lines, occupying a mere 6 square inches on the page without compromising readability:

1. Whear 'ast tha been sin' I saw thee (I saw thee)
On Ilkley Moor Bah T'at
Whear 'ast tha been sin' I saw thee (2x)

On Ilkley Moor bah t'at (bah t'at) on Ilkley Moor bah t'at
On Ilkely Moor bah t'at.
2. Tha's been a-coortin' Mary Jane
3. Tha'll go an' get thee death o' cowld

…etc., for 6 more lines

LZ compression (see **ADAPTIVE COMPRESSION**) and its variants use a similar but mechanical technique to remove redundancy in text or binary data.

3. Lossy Compression

Compression techniques that ensure that the result of decompression is exactly the same data as before compression are known as *lossless*. Many of the more powerful forms of compression are *lossy*. With lossy compression, decompression will produce an approximation to the original information rather than an exact copy. Lossy compression uses knowledge of the specific kind of data being stored, and of the required uses for the data.

The key to understanding lossy compression is the difference between *data* and *information*. Suppose you wish to communicate the information represented by the word "elephant" to an audience. Sent as a text string, 'elephant' occupies 8 7-bit ASCII characters, or 56 bits. Alternatively, as a spoken word encoded in 16 bit samples 8000 times per second, 'elephant' requires 1 second of samples, i.e. 128 KBits. A full-screen colour image of an elephant at 640*480 pixels might require 2.5 Mbits, and a video displayed for one second at 50 frames per second could take 50 times that, or 125 Mbits. None of the more expensive techniques convey much more information than just the text of "elephant", however. If all you are interested in the basic concept of an "elephant", most of the data required by the other techniques is redundant. You can exploit this redundancy in various ways.

Simplest, you can omit irrelevant data. For example you might be receiving uncompressed sound data represented as 16 bit samples. If your sound sampler isn't accurate enough to record 16-bit samples, the least significant 2 bits in each sample will be random, so you could achieve a simple compression by just storing 14 instead of 16 bits for each sample.

You can also exploit the nature of human perception to omit data that's less important. For example, we perceive sound on a 'log scale'; the ear is much less sensitive to differences in intensity when the intensity is high than when intensity is low. You can effectively compress sound samples by converting them to a log scale, and supporting only a small number of logarithmic intensities. This is the principle of some simple sound compression techniques, particularly mu-law and a-law, which compress 14 bit samples to 8 bits in this way [CCITT G.711, Brokish and Lewis 1997].

You can take this idea of omitting data further, and transform the data into a different form to remove data irrelevant to human perception. Many of the most effective techniques do this:

JPEG The most commonly used variants of the JPEG standard represent each 8x8 pixel square as a composite of a standard set of 64 'standard pictures' – a fraction of each picture. The transformation is known as the 'cosine transform'. Then the fractions are represented in more or less detail according to the importance of each to human perception. This gives a format that compresses photographic data very effectively. [ITU T.87, Gonzalez and Woods 1992]

GSM GSM compression represents voice data in terms of a mathematical model of the human voice (Regular Pulse Excited Linear Predictive Coding)[1]. In this way it encodes separate 20mS samples in just 260 bits, allowing voice telephony over a digital link of only 13 Kbps. [Degener 1994]

GIF, PNG The proprietary GIF and standard PNG formats both map all colours in an image to a fixed-size palette before encoding. [CompuServe 1990, Boutell 1996].

MP3 MP3 represents sound data in terms of its composite frequencies – known as the 'Fourier Transformation'. The MP3 standard specifies the granularity of representation of each frequency according to its importance to the human ear and the amount of compression required, allowing FM radio-quality sound in 56 Kb per second [MP3].

MPEG The MPEG standard for video compression uses JPEG coding for initial frames. It then uses a variety of specific techniques – to spot motion in a variety of axes, changes of light, etc. – to encode the differences between successive frames in minimal data forms that fit in with the human perception of a video image [MPEG].

Some of these techniques exploit mechanical redundancy in the resulting data as well, using **TABLE COMPRESSION**, **DIFFERENCE CODING** and **ADAPTIVE** techniques.

❖ ❖ ❖

Specialised Patterns

The rest of this chapter contains specialised patterns describing compression and packing techniques. Each of these patterns removes different kinds of mechanical and semantic redundancy, with different consequences for accessing the compressed data.

[1] GSM was developed using Scandinavian voices; hence all voices tend to sound Scandinavian on a mobile phone.

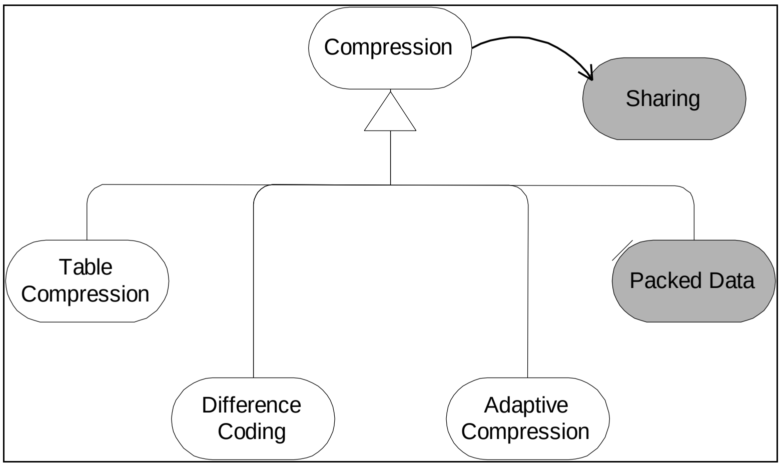


Figure 1: Compression Patterns

The patterns are as follows:

- **TABLE COMPRESSION** reduces the average number of bits to store each character (or value) by mapping it to a variable number of bits, such that the most common characters require the fewest bits.
- **DIFFERENCE COMPRESSION** addresses data series or sequences, by storing only the differences between successive items. Alternatively or additionally, if several successive items are the same it stores simply a count of the number of identical items.
- **ADAPTIVE COMPRESSION** analyses the data before or while compressing it to produce a more efficient encoding, storing the resulting parameters along with the compressed data – or uses the data itself as a table to support the compression.

The **PACKED DATA** pattern, that reduces the amount of memory allocated to store random-access data structures, can also been seen as a special kind of compression.

1. Evaluating Compression Techniques

There are many possible compression techniques. Here are some of the things to consider when choosing an appropriate technique:

1.1 Processing and Memory Required. Different techniques vary significantly in the processing and memory costs they impose. In general, **DIFFERENCE CODING** has the lowest costs, followed by fixed **TABLE COMPRESSION**, and most forms of **ADAPTIVE COMPRESSION** have quite high costs on both counts – but there are many exceptions to this rule. *Managing Gigabytes* [Witten, Moffat, Bell 1999] examines the costs in some detail.

1.2. Encoding vs. Decoding. Some compression algorithms reduce the processing cost of decoding the data by increasing the cost of encoding. This is particularly advantageous if there is one large and powerful encoding system and many decoders with a lower specification. This is a situation common in broadcast systems.

MP3 and MPEG, for example, require much more processing, code and memory to encode than decode, which suits them for broadcast transmission [MP3,MPEG]. Interestingly LZ **ADAPTIVE COMPRESSION** has the same feature, so ZIP archives can be distributed with their relatively simple decoding software built-in [Ziv and Lempel 1977].

Some compressed representations can be used directly without any decompression. For example, Java and Smalltalk use byte coding to reduce the size of executable programs; intermediate codes can be smaller than full machine language [Goldberg and Robson 1983, Lindholm and Yellin 1999]. These byte codes are designed so that they can be interpreted directly by a virtual machine, without a separate decompression step.

1.3. Programming Cost. Some techniques are simple to implement; others have efficient public domain or commercial implementations. Rolling your own complicated compression or decompression algorithm is unlikely to be a sensible option for many projects.

1.4. Random Access and Resynchronisation. Most compression algorithms produce a stream of bits. If this stream is stored in memory or in a file, can you access individual items within that file randomly, without reading the whole stream from the beginning? If you're receiving the stream over a serial line and the some is corrupted or deleted, can you resynchronise that data stream, that is, can you identify the start of a meaningful piece of data and continue decompression? In general, most forms of **TABLE COMPRESSION** can provide both random access and resynchronisation; **DIFFERENCE CODING** can also be tailored to handle both; **ADAPTIVE COMPRESSION**, however, is unlikely to work well for either.

Known Uses

Compression is used very widely. Operating systems use compression to store more information on secondary storage, communications protocols use compression to transmit information more quickly, virtual machines use compression to reduce the memory requirements of programs, and general purpose file compression tools are use ubiquitously for file archives.

See Also

As well as compressing information, you may be able to store it in cheaper **SECONDARY STORAGE** or **READ ONLY MEMORY.** You can also remove redundant data using **SHARING**

The excellent book 'Managing Gigabytes' [Witten, Moffat, Bell, 1999] explains all of this chapter's techniques for compressing text and images in much greater detail. 'Text Compression' [Bell, Cleary, Witten 1990] focuses on text compression.

The online book, 'Information Engineering Across the Professions' [Cyganski, Orr, and Vaz 1998] has explanations of many different kinds of Text, Audio, Graphical and Video compression.

The FAQ of the newsgroup `comp.compression` describes many of the most common compression techniques. Steven Kinnear's web page [1999] provides an introduction to multimedia compression, with an excellent set of links to other sites with more detail.

'Digital Video and Audio Compression' [Solari 1997] has a good description of techniques for multimedia compression.

Table Compression Pattern

Also know as: Nibble Coding, Huffman Coding.

How do you compress many short strings?

- You have lots of small-to-medium sized strings in your program — all different
- You need to reduce your program's *memory requirements.*
- You need random access to individual strings.
- You don't want to expend too much extra *programmer effort*, *memory space,* or *processing time* on managing the strings.

Many programs use a large number of strings — stored in databases, read from **RESOURCE FILES**, received via telecommunications links or hard-coded in the program. All these strings increase the program's *memory requirements* for main memory, read-only memory, and secondary storage.

Programs need to be able to perform common string operations such as determining their length and internal characters, concatenating strings, and substituting parameters into format strings, however strings are represented. Similarly, each string in a collection of strings needs to be individually accessible. If the strings are stored in a file on secondary storage, for example, we need *random access* to each string in the file.

Although storing strings is important, it is seldom the most significant memory use in the system. Typically you don't want to put to much *programmer effort* into the problem. Equally, you may not want to demand too much *temporary memory* to decompress each string.

For example, the Strap-It-On PC needs to store and display a large number of information and error messages to the user. The messages need to be stored in scarce main memory or read-only memory, and there isn't really enough space to store all the strings directly. The Programs must be able to access each string individually, to display them to the user when appropriate. Given that many of the strings describe exceptional situations such a memory shortage, they need to be able to be retrieved and displayed quickly, efficiently, and without requiring extra memory.

Therefore: *Encode each element in a variable number of bits so that the more common elements require fewer bits.*

The key to table compression is that some characters are statistically much more likely to occur than others. You can easily map from a standard fixed size representation to one where each character takes a different number of bits. If you analyse the kind of text you're compressing to find which characters are most probable, and map these characters to the most compact encoding, then on average you'll end up with smaller text.

For example the chart below shows the character frequencies for all the lower-case characters and spaces in a draft of this chapter.

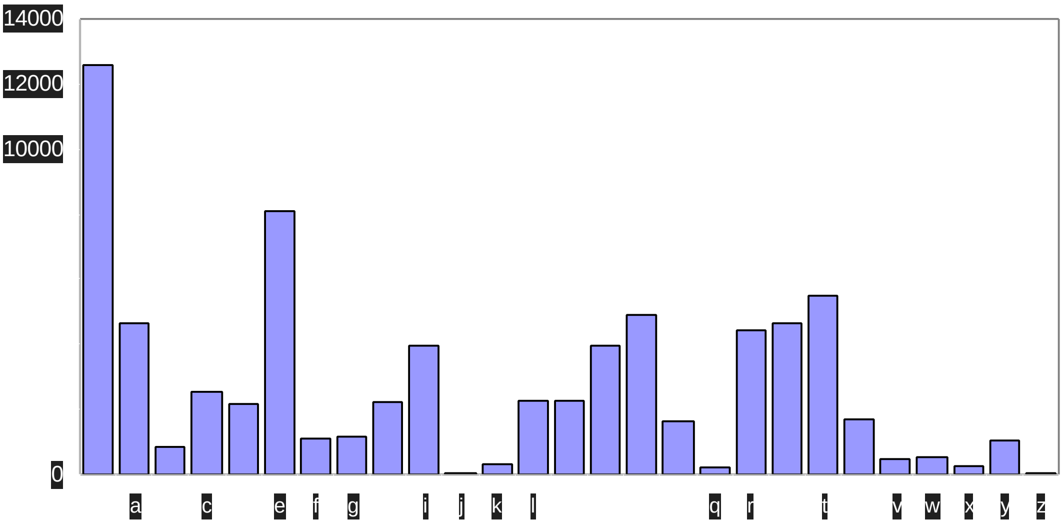


Figure 2 Distribution of Characters in this Chapter

Obviously some characters (space, 'e') appear much more often than others do ('z', 'j'). A surprisingly large proportion of the characters is never used at all[2]. The most common character, space, occupies 15% of the memory required to hold the chapter. The 15 most common characters occupy 75% of the total memory.

Given that the Strap-It-On's information and error messages have a similar distribution of characters, its designers get a significant reduction in the storage space required by encoding the most common characters in fewer bits, and the less common characters in more bits. Using the Huffman Encoding as described below, the designers of the Strap-it-on have achieved compression of 5 bits per character for its error messages, with negligible costs in run-time performance and temporary memory costs.

Consequences

Typically you get a reasonable compression for the strings themselves, reducing the program's *memory requirements*. Sequential operations on compressed strings execute almost as fast as operations on native strings, preserving *time performance*. String compression is quite easy to implement, so it does not take much *programmer effort*. Each string in a collection of compressed strings can be accessed individually, without decompressing all proceeding strings.

However: the total compression of the program data – including non-string data – isn't high, so the program's *memory requirements* may not be greatly reduced**.**

String operations that rely on random access to the characters in the string may execute up to an order of magnitude slower than the same operations on decompressed strings, reducing the program's *time performance*. Because characters may have variable lengths, you can only access a specific character by scanning from the start of the string. If you want operations that change the characters in the string you have to uncompress the string, make the changes, and recompress it.

It requires *programmer discipline* to use compressed strings, especially for string literals within the program code. Compressed strings require either manual encoding or a string pre-processing pass, either of which increases complexity.

[2] Well, hardly ever. [kjxcheck reference:Gilbert&Sullivan]

You have to *test* the compressed string operations, but these tests are quite straightforward.

❖ ❖ ❖

Implementation

There are many techniques used for table compression [Whitten et al 2000]. The following sections explore a few common ones.

1. Simple coding

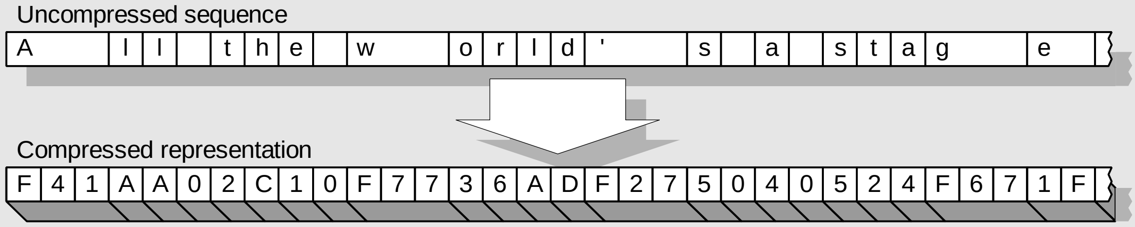
If the underlying character set has only 128 characters, such as ASCII, it certainly makes sense to store each character in seven bits, rather than eight, sixteen, or thirty-two bits. But, as we discussed above, in fact a large proportion of normal text could be encoded in just four bits. Other non-European languages might be better with five or six bits.

If you encode most of the text into, say, small fixed size characters, what do you do with the characters not within the most common set? The answer is to use 'escape codes'. An escape code is a special character that changes the meaning of the following character (or sometimes of the characters up to the next escape code).

For example, a common simple coding technique is to use a *nibble code*, where each character is coded into four bits. A nibble code is a easy to implement, because a nibble is always half a byte, making it easy to write the packed data. In a basic nibble code, we might have only one escape code, which is followed by the eight-bit ASCII code of the next character. So using the data in figure xx above to deduce the most common characters, we can construct an encoding and decoding table as follows:

Plain text	Encoded Nibbles	Encoded Bits
?	0	0000
a	4	0100
b	F 6 1	1111 0110 0001
c	F 6 2	1111 0110 0010
d	D	1101
e	1	0001
f	F 6 5	1111 0110 0101
… etc		

Thus the phrase "All the world's a stage" would encode as follows:


Uncompressed sequence
A l l t h e w o r l d ' s a s t a g e
Compressed representation
F 4 1 A A 0 2 C 1 0 F 7 7 3 6 A D F 2 7 5 0 4 0 5 2 4 F 6 7 1 F

Nibble coding

All values in hexadecimal.

Using this nibble code, 75% of characters in this chapter can be encoded in a 4 bits; the remainder all require 12 bits. On this simple calculation the average number of bits required per character is 6 bits; when we implemented the nibble code and tested it on the file, we achieved 5.4 bits per character in practice.

2. Huffman Coding

Why choose specifically 4 bits for the most common characters and 12 bits for the escaped characters? It would seem more sensible to have a more even spread, so that the most common characters (e.g. space) use less than four bits, fairly common characters ('u', 'w') require between four and eight bits, and only the least common ones ('Q', '&') require more. Huffman Coding takes this reasoning to its extreme. With Huffman coding, the 'Encoded bits' column in table xxx becomes will contain bit values of arbitrary lengths instead of either 4 or 12 [Huffman 1952].

Decoding Huffman data is a little trickier. Since you can't just look up an unknown length bit string in an array, Huffman tables are often represented as trees for decoding; each terminal node in the tree is a decoded character. To decode a bit string, you start at the root, then take the left node for each 1 and the right node for each 0. So, for example, if you had the following simple encoding table for a 4-character alphabet, where A is the most frequent character in your text, then D, then B and C:

Plain Text	Encoded Bits
A	1
B	010
C	011
D	00

This can be represented as a Huffman Tree as:

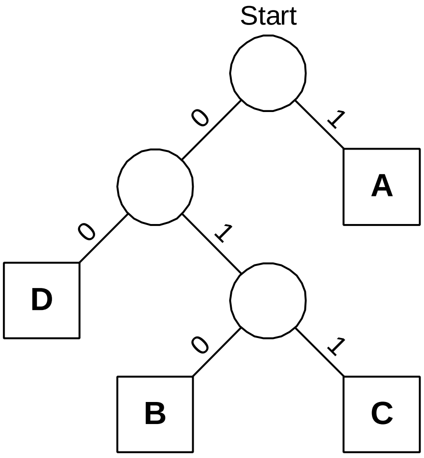


Figure 3: Huffman Tree

For more about Huffman coding – more efficient decoding techniques and a discussion on generating the Huffman encoding tables – see 'Managing Gigabytes' [Witten, et al 1999] or any other standard reference on text compression.

3. Encoding more than just characters

There's no need to limit **TABLE COMPRESSION** to strings; anything that contains data items of fixed numbers of bytes can be compressed in this way. Other compression techniques, for example, often apply Huffman coding to their intermediate representations to increase their compression ratios (see, for example, the **ADAPTIVE COMPRESSION** technique GZIP).

TABLE COMPRESSION does not have to be restricted to compressing fixed-length items, as long as each item has a clearly defined end. For example, Huffman Word Compression achieves very high compression ratios (3 bits per character or so) by encoding each word separately [Witten, et al 1999]. To achieve this ratio, Huffman Word Compression requires a very large compression table – the size of the dictionary used.

4. Compressing String Literals

Compressed strings are more difficult to handle in program code. While programming languages provide string literals for normal strings, they do not generally support compressed strings. Most languages support escape codes (such as `"\x34"`) that allow any numeric characters to be stored into the string. Escape codes can be used store compressed strings in standard string literals. For example, here's a C++ string that stores the nibble codes for the "All the world's a stage" encoding in figure XX.

```
const char* AllTheWorldsAStage =
          "\xf4\x1a\xa0\x2c\x10\xf7\x73\x6a\xdf\x27\x50\x40\x52\x4f\x67\x1f";
```

You can also write a pre-processor to work through program texts, and replace standard encoded strings with compressed strings. This works particularly well when compressed strings can be written as standard string or array literals. Alternatively, in systems that store strings in **RESOURCE FILES**, the resource file compiler can compress the string, and the resource file reader can decompress it.

5. UTF8 Encoding

To support internationalisation, an increasing number of applications do all their internal string handling using two-byte character sets – typically the UNICODE standard [Unicode 1996]. Given that the character sets for most European languages require less than 128 characters, the extra byte is clearly redundant. For storage and transmission, many environments encode UNICODE strings using the UTF8 encoding.

In UTF8, each UNICODE double byte is encoded into one, two or three bytes (though the standard supports further extensions). The coding encodes the bits as follows:

UNICODE value	**1st Byte**	**2nd Byte**	**3rd Byte**
000000000xxxxxxx	0xxxxxxx		
00000yyyyyxxxxxx	110yyyyy	10xxxxxx	
Zzzzyyyyyyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx

So in UTF8, the standard 7-bit ASCII characters are encoded in a single byte; in fact, the UNICODE encoding is exactly the same as the one byte ASCII encoding for these characters. Common extended characters are encoded as two bytes, with only the least common characters requiring three bytes. Every UTF8 character starts and ends on a byte boundary, so you can identify a substring within a larger buffer of UTF8 characters using just a byte offset and a length.

UTF8 wasdesigned to be especially suitable for transmission along serial connections. A terminal receiving UTF8 characters can always determine which byte represents the start of a UNICODE character, because either their top bit is 0, or the top two bits are ‘11’. Any UTF8 bytes with the top bits equal to “10” are always 2nd or 3rd in sequence and should be ignored unless the terminal has received the initial byte.

Example

This example implements a nibble code to compress the text in this chapter. Figure 3 below shows the distribution of characters in the text, sorted by frequency, with the 15 most common characters to the left of the vertical line.

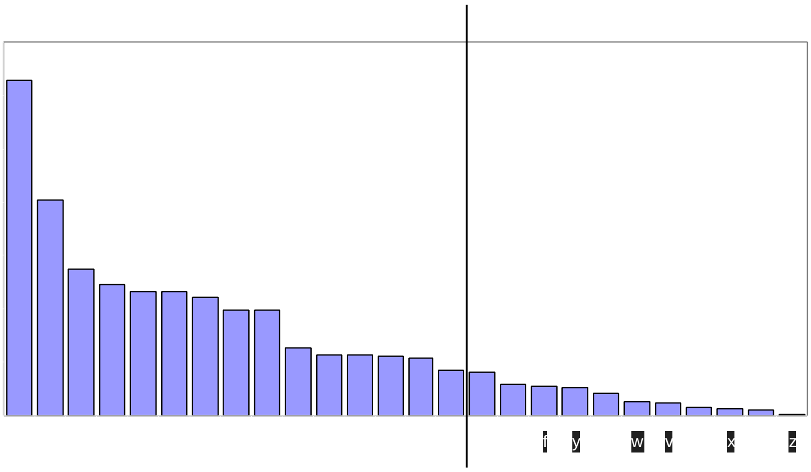


Figure 4 Choosing the 15 most common characters.

Here is a Java example that implements this nibble code. The `StringCompression` class uses a byte array output stream to simplify creating the compressed string — the equivalent in C++ would use an `ostrstream` instance. The most common characters are represented in the string `NibbleChars`:

```
protected final String NibbleChars = " etoasrinclmhdu";
protected final int NibbleEscape = 0xf;
protected int lastNibble;
protected ByteArrayOutputStream outStream;
```

The `encodeString` method takes each fixed-size character and encodes it, character by character. This function has to deal with end effects, ensuring the last nibble gets written to the output file by padding it with an escape character.

```
protected byte[] encodeString(String string) {
        outStream = new ByteArrayOutputStream();
        lastNibble = -1;
        for (int i = 0; i < string.length(); i++) {
            encodeChar(string.charAt(i));
        }
        if (lastNibble != -1) {
            putNibble(NibbleEscape);
        }
        byte[] result = outStream.toByteArray();
        outStream = null;
        return result;
}
```

The most important routine encodes a specific character. The `encodeChar` method searches the `NibbleChars` string directly; if the character to be encoded is in the string it is output as a nibble, otherwise we output an escape code and a high and low nibble. A more efficient implementation could use a 256-entry table lookup.

```
protected void encodeChar(int charCode) {
    int possibleNibble = NibbleChars.indexOf(charCode);
    if (possibleNibble != -1) {
      putNibble(possibleNibble);
    } else {
      putNibble(NibbleEscape);
      putNibble(charCode >>> 4);
      putNibble(charCode & 0xf);
    }
  }
```

The `putNibble` method simply adds one nibble to the output stream. We can only write whole bytes, rather than nibbles, so the `lastNibble` variable stores a nibble than has not been output. When another nibble is received, both `lastNibble` and the current nibble `n` can be written as a single byte:

```
protected void putNibble(int nibble) {
    if (lastNibble == -1) {
      lastNibble = nibble;
    } else {
      outStream.write((lastNibble << 4) + nibble);
      lastNibble = -1;
    }
  }
}
```

Decoding is similar to encoding. For convenience, the decoding methods belong to the same class; they use a `ByteArrayInputStream` to retrieve data. The `decodeString` method reads a character at a time and appends it to the output:

```
protected ByteArrayInputStream inStream;

protected String decodeString(byte [] inBuffer) {
  inStream = new ByteArrayInputStream(inBuffer);
  StringBuffer outString = new StringBuffer();
  lastNibble = -1;
  int charRead;
  while ((charRead = decodeChar()) != -1) {
    outString.append( (char)charRead );
  }
  return outString.toString();
}
```

The `decodeChar` method reads as many input nibbles as are required to compose a single character.

```
protected int decodeChar() {
  int s = getNibble();
  if (s == -1) return -1;
  if (s != NibbleEscape) {
    return NibbleChars.charAt(s);
  } else {
    s = getNibble();
    if (s == -1) return -1;
    return (s << 4) + getNibble(); }
}
```

Method `getNibble` actually returns one nibble from the input stream, again keeping the extra nibble in the `lastNibble` field when a full byte is read by only one nibble returned.

```
protected int decodeChar() {
    int nibble = getNibble();
    if (nibble == -1) {
        return -1;
    }
    if (nibble != NibbleEscape) {
        return NibbleChars.charAt(nibble);
    } else {
        nibble = getNibble();
        if (nibble == -1) {
            return -1;
        }
        return (nibble << 4) + getNibble();
    }
}
```

Nibble encoding can be surprisingly effective. For example a text-only version of this chapter compresses to just 5.4 bits per char (67%) using this technique. Similarly, the complete set of text resources for a release of the EPOC32 operating system would compress to 5.7 bits per character (though as the total space occupied by the strings is only 44 Kb, the effort and extra code required have so far not been worthwhile).

❖ ❖ ❖

Known Uses

Reuters worldwide IDN network uses Huffman encoding to reduce the bandwidth required to transmit all the world's financial market prices worldwide. The IDN Huffman code table is reproduced and re-implemented in many different systems.

GZIP uses Huffman encoding as part of the compression process, though their main compression gains are from **Adaptive Compression**. [Deutsch 1996]

The MNP5 and V42.bis modem compression protocols uses Huffman Encoding to get compression ratios of 75% to 50% on typical transmitted data [Held 1994].

Nibble codes were widely used in versions of text adventure games for small machines [Blank and Galley 1980]. Philip Gage used a similar technique to compress an entire string table [Gage 97]. Symbian's EPOC16 operating system for the Series 3 used table compression for its **RESOURCE FILES** [Edwards 1997].

Variants of UTF8 encoding are used in Java, Plan/9, and Windows NT to store Unicode characters [Unicode 1996, Lindholm and Yellin 1999, Pike and Thompson 1993].

See Also

Many variants of table compression are also **Adaptive**, calculating the optimal table for each large section of data and including the table with the compressed data.

Compressed strings can be stored in **Resource Files** in **Secondary Storage** or **Read-Only Memory**, as well as primary storage. Information stored in **Data Files** can also be compressed.

Witten, Moffat, and Bell [1999] and Cyganski, Orr, and Vaz [1998] discuss Huffman Encoding and other forms of table compression in much more detail than we can give here. Witten, Moffat and Bell also includes discussions of memory and time costs for each technique.

Business Data Communications and Networking [Fitzgerald and Dennis 1995] provides a good overview of modem communications. Sharon Tabor's course materials for '*Data Transmission*' [2000] provide a good terse summary. The '*Ultimate Modem Handbook*' includes an outline of various modem compression standards [Lewart 1999].

Difference Coding Pattern

Also know as: Delta Coding, Run Length Encoding

How can you reduce the memory used by sequences of data?

- You need to reduce your program's *memory requirements*
- You have *large* streams of data in your program
- The data streams which will be accessed sequentially
- There are significant time or financial costs of storing or transferring data.

Many programs use sequences or series of data — for example, sequential data such as audio files or animations, time series such as stock market prices, values read by a sensor, or simply the sequence of bytes making up program code. All these sequences increase the program's *memory requirements*, or worsen the *transmission time* using a telecommunications link.

Typically this sort of streamed data is accessed sequentially, beginning at the first item and then processing each item in turn. Programs rarely or never require random access into the middle of the data. Although storing the data is important, it often isn't the largest problem you have to face — gathering the data is often much more work than simply storing it. Typically, you don't want to devote too much *programmer effort*, *processing time*, or *temporary memory* to the compression operations.

For example, the Strap-It-On PC needs to store results collected from the Snoop-Tronic series of body wellbeing monitors. These monitors are attached onto strategic points on the wearer's body, and regularly measure and record various physiological, psychological, psychiatric and psychotronic metrics (heartbeats, blood-sugar levels, alpha-waves, influences from the planet Gorkon, etc). This information needs to be stored in the background while the Strap-It-On is doing other work, so the recording process cannot require much processor time or memory space. The recording is continuous, gathering data whenever the Strap-It-On PC and Snoop-Tronic sensors are worn and the wearer is alive, so large amounts of data are recorded. Somehow, we must reduce the memory requirements of this data.

Therefore: *represent sequences according to the differences between each item.*

Continuous data sequences are rarely truly random — the recent past is often an excellent guide to the near future. So in many sequences:

- The values don't change very much between adjacent items, and
- There are 'runs', where is no change for several elements.

These features result in two complementary techniques to reduce the number of bits stored per item:

- Delta Coding stores just differences between each successive item.
- Run-length Encoding (RLE) replaces a run of identical elements with a repeat count.

For example, in the data stored by the Snoop-Tronic monitors, the values read are very close or the same for long periods of time. The Strap-It-On PCs driver for the Snoop-Tronic sensors uses sequence coding on the data streams as they arrive from each sensor, buffers the data, and stores it to secondary storage — without imposing a noticeable overhead on the performance of the system.

Consequences

Difference compression can achieve an excellent compression ratio for many kinds of data (particularly cartoon-style picture data and some forms of sound data) reducing the program's *memory requirements*. Sequential operations on the compressed data can execute almost as fast as operations on native values preserving *time performance* and *real-time responsiveness*, and considerably improving *time performance* if there are slow disk or telecommunication links involved.

Difference compression is quite easy to implement, so it does not take much *programmer effort*, or extra *temporary memory*.

However: The compressed sequences are more to difficult to manage than sequences of absolute data values. In particular, it is difficult to provide random access into the middle of compressed sequences without first uncompressing them, requiring *temporary memory* and *processing time*.

Some kinds of data – such as hi-fi sound or photographic images – don't reduce significantly with **Difference Compression**.

You have to *test* the compressed sequence operations, but these tests are quite straightforward.

❖ ❖ ❖

Implementation

Here are several difference coding techniques that you can consider:

1. Delta Coding

Delta coding (or difference coding) stores differences between adjacent items, rather than the absolute values of the items themselves [Bell et all 1990]. Delta coding saves memory space because deltas can often be stored in smaller amounts of memory than absolute values. For example, you may be able to encode a slowly varying stream of sixteen bit values using only eight-bit delta codes.

Of course, the range of values stored in the delta code is less than the range of the absolute item values (16-bit items range from 0 to 65536, while 8 bit deltas give you +- 127). If the difference to be stored is larger than the range of delta codes, typically the encoder uses an escape code (a special delta value, say –128 for an 8-bit code) followed by the full absolute value of the data item.

Figure xx below shows such a sequence represented using delta coding. All values are in decimal, and the escape code is represented as '? ':

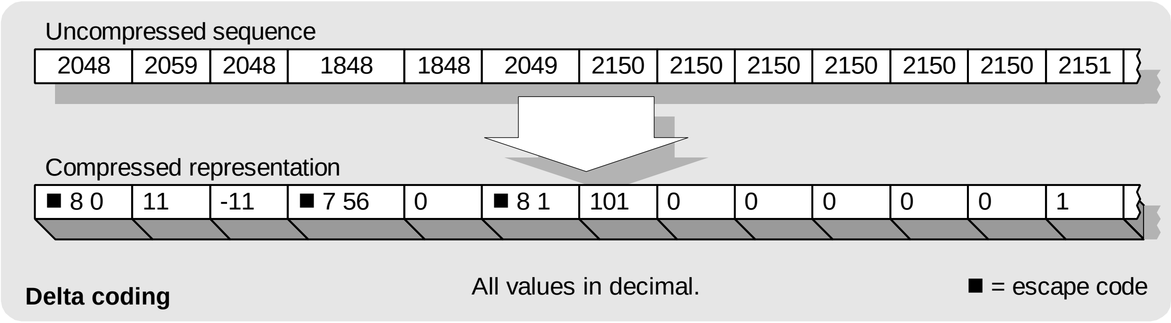


Figure 5: Delta Coding

2. Run Length Encoding

Run-length encoding compresses a run of duplicated items by storing the value of the duplicated items once, followed by the length of the run [Bell et all 1990]. For example, we can extend the delta code above to compress runs by always following the escape code by a count byte as well as the absolute value. Runs of between 4 and 256 items can be compressed as the escape code, the absolute value of the repeated item, and the count. Runs of longer than 256 items can be stored as repeated runs of 256 characters, plus one more run of the remainder. Figure XX shows RLE added to the previous example:

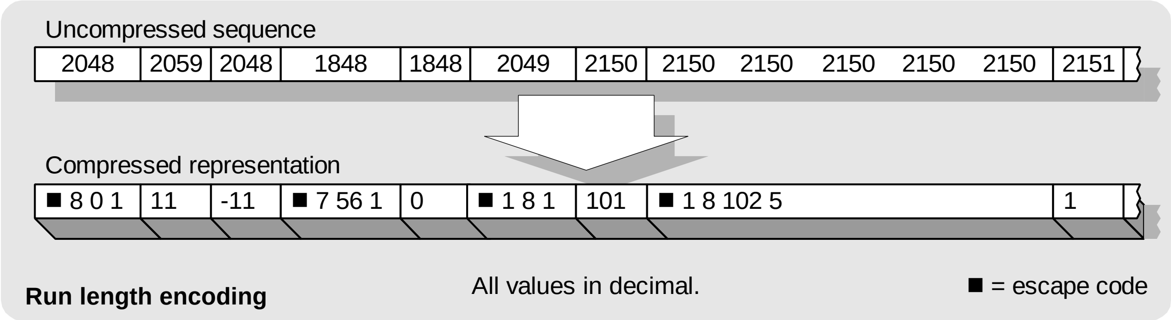


Figure 6: Run Length Encoding and Delta Coding

3. Lossy Difference Compression

Here are some common techniques that increase the compression ratio of sequence compression by losing some of the information compressed:

1. You can treat a sequence with only negligible differences in values as if they were a run of items with identical values. For example, in the data series above, differences within a quarter of a percent of the absolute value of the data items may not be significant in the analysis. Quite possibly they could be due to noise in the recording sensor or the ambient temperature when the data item was recorded. A quarter of one percent of 2000 is 20 — so we can code the first three items as a run.
2. You can handle large jumps in delta values by allowing a lag in catching up. Thus, for example, the difference of 200 between 2048 to 1848 can be represented as two deltas, rather than an escape code.

Using these two techniques, we can code the example sequence as shown in figure XX:

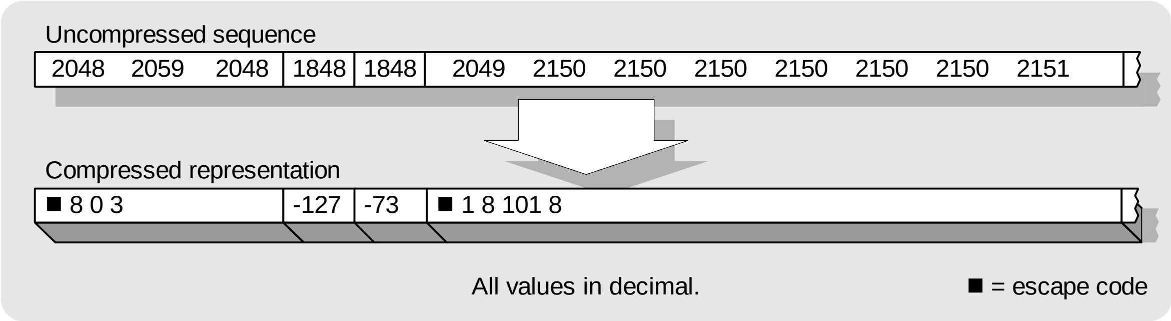


Figure 7: Lossy Sequence Compression

3. You can increase the granularity of the delta values, so that each delta value is scaled by the magnitude of the items they are representing. So, for example, each delta step could be the nearest integer below 0.25% of the previous item's value, allowing much larger deltas.

4. Resynchronisation

Sequence compression algorithms are often used for broadcast communications and serial or network connections. In many cases, particularly with multimedia data streams, it doesn't matter very much if part of the sequence is lost or corrupted, so long as later data can be read correctly. Because difference codes assume the receiver knows the correct value for the last item (so that the next item can be computed by adding the difference), one wrong delta means that every subsequent delta will produce the wrong value. To avoid this problem, you can include resynchronisation information; every now and again you can send a complete value as escape code, instead of a delta. The escape code resets the value of the current item, correcting any accumulated error due to corruption or lost data.

5. Non-numeric data

Difference Coding can also be very effective at compressing non-numeric data structures. In Delta Coding, the deltas will be structures themselves; for RLE represents events where the structures haven't changed. For example, you can think some forms of the Observer pattern [Gamma et al 1995] as examples of delta compression: the observer is told only the changes that have happened.

Similarly you can do run-length encoding using a count of a number of identical structures. For example the X Window System can return a single compressed mouse movement event that represents a number of smaller movements — the compressed event contains a count of the number of uncompressed movements it represents [Scheifler and Gettys 1986].

Examples

The following Java example compresses a sequence of two-byte values into a sequence of bytes using both difference compression and run length encoding. The compression is lossless, and the only escape sequence contains both the complete value and the sequence length. As above, the bytes of the escape sequence are:

```
<escape> <high byte of repeated value> <low byte> <sequence count>
```

The `encodeSequence` method takes a sequence of shorts, and passes each one to the `encodeShort` method, which will actually encode them:

```
protected final int SequenceEscape =  0xff;
protected final int MaxSequenceLength =  0xFE;
protected short lastShort;
protected short runLength;

protected void encodeSequence(short[] inputSequence) {
    lastShort = 0;
    runLength = 0;

    for (int i = 0; i < inputSequence.length; i++) {
        encodeShort(inputSequence[i]);
    }
    flushSequence();

}
```

The `encodeShort` method does most of the work. It first checks if its argument is part of a sequence of identical values, and if so, simply increases the run length count for the sequence — if the sequence is now the maximum length that can be represented, an escape code is written. If its argument is within the range of the delta coding (± 128 from the last value) an escape code is written if necessary, and a delta code is written. Finally, if the argument is outside the range, an escape code is written to terminate the current run length encoded sequence if necessary. In any event, the current argument is remembered in the `lastShort` variable.

```
protected void encodeShort(short s) {
        if (s == lastShort) {
            runLength++;
            if (runLength >= MaxSequenceLength) {
                flushSequence();
            }
        } else if (Math.abs(s - lastShort) < 128 ) {
            flushSequence();
            writeEncodedByte(s - lastShort + 128);
        } else {
            flushSequence();
            runLength++;
        }
        lastShort = s;
}
```

The `flushSequence` method simply writes out the escape codes, if required, and resets the run length. It is called whenever a sequence may need to be written out — whenever `encodeShort` detects the end of the current sequence, or that the current sequence the longest that can be represented by the run length escape code.

```
    protected void flushSequence() {
        if (runLength == 0) return;
        writeEncodedByte(SequenceEscape);
        writeEncodedByte(lastShort >>> 8);
        writeEncodedByte(lastShort & 0xff);
        writeEncodedByte(runLength);
        runLength = 0;
}
```

The corresponding decoding functions are straightforward. If an escape code is read, a run of output values is written, and if a delta code is read, a single output is written which differs from the last output value by the delta.

```
protected void decodeSequence(byte[] inBuffer) {
        ByteArrayInputStream inStream =
            new ByteArrayInputStream(inBuffer);
        lastShort = 0;
        int byteRead;

        while ((byteRead = inStream.read()) != -1) {
            byteRead = byteRead & 0xff;

            if (byteRead == SequenceEscape) {
                lastShort = (short) (((inStream.read() &0xff ) << 8) +
                                     (inStream.read() & 0xff));
                for (int c = inStream.read(); c > 0; c--) {
                    writeDecodedShort(lastShort);
                }
            } else {
                writeDecodedShort(lastShort += byteRead -128);
            }
        }
    }
```

❖ ❖ ❖

Known Uses

Many image compression techniques use Different Compression. The TIFF image file format uses RLE to encode runs of identical pixels [Adobe 1992. The GIF and PNG formats do the same after (lossy) colour mapping [CompuServe 1987, Boutell 1996]. The Group 3 and 4 Fax transmission protocols uses RLE to encode the pixels on a line [Gonzalez and Woods 1992]; the next line (in fine mode) or three lines (in standard mode) are encoded as differences from the first line.

MPEG video compression uses a variety of techniques to express each picture as a set of differences from the previous one [MPEG, Kinnear 1999]. The V.42bis modem compression standard includes RLE and **TABLE COMPRESSION** (Huffman Coding), achieving a total compression ratio of up to 33% [Held 1994].

Many window systems in addition to X use Run Length encoding to compress events. For example MS Windows represents multiple mouse movements and key auto-repeats in this way, and EPOC's Window Server does the same [Petzold 1998, Symbian 1999].

Reuters IDN system broadcasts the financial prices from virtually every financial exchange and bank in the world, aiming – and almost always succeeding – in transmitting every update to every interested subscriber in under a second. To make this possible, IDN represents each 'instrument' as a logical data structure identified by a unique name (Reuters Identification Code); when the contents of the instrument (prices, trading volume etc.) change, IDN transmits only the changes. To save expensive satellite bandwidth further, these changes are transmitted in binary form using Huffman Coding (see **TABLE COMPRESSION**), and to ensure synchronisation of all the Reuters systems worldwide, the system also transmits a background 'refresh' stream of the complete state of every instrument.

See Also

You may want to use **TABLE COMPRESSION** in addition to, or instead of **DIFFERENCE CODING**. If you have a large amount of data, you may be able to tailor your compression parameters (**ADAPTIVE COMPRESSION**), or to use a more powerful **ADAPTIVE** algorithm.

The references discussed in the previous patterns are equally helpful on the subject of **DIFFERENCE COMPRESSION**. Witten, Moffat and Bell [1999] explain image compression techniques and tradeoffs; Cyganski, Orr, and Vaz [1998] and Solari [1997] explain audio, graphical and video compression techniques, and Held [1994] discusses modem compression.

Adaptive Compression Pattern

How can you reduce the memory needed to store a large amount of bulk data?

- You have a large amount of data to store, transmit or receive.
- You don't have enough persistent memory space to store the information long term, or you need to communicate the data across a slow telecommunications link
- You have transient memory space for processing the data.
- You don't need random access to the data

A high proportion of the memory requirements of many programs is devoted to bulk data. For example, the latest application planned for the Strap-It-On PC is ThemePark:UK, a tourist guide being produce in conjunction with the Unfriendly Asteroid travel consultancy. ThemePark:UK is based on existing ThemePark products, which guide users around theme parks in Southern California. ThemePark:UK will treat the whole of the UK as a single theme park; the Strap-It-On will use its Global Positioning System together an internal database to present interactive travel guides containing videos, music, voice-overs, and genuine people personalities for cute interactive cartoon characters. Unfortunately the UK is a little larger than most theme parks, and the designers have found that using **TABLE COMPRESSION** and **DIFFERENCE COMPRESSION** together cannot cram enough information into the Strap-It-On's memory.

This kind of problem is common in applications requiring very large amounts of data, whether collections of documents and emails or representations of books and multimedia Even if systems have sufficient main memory to be able to process or display the parts of the data they need at any given time, they may not have enough memory to store all the information they will ever need, either in main memory or secondary storage.

Therefore: *Use an adaptive compression algorithm.*

Adaptive compression algorithms can analyse the data they are compressing and modify their behaviour accordingly. These adaptive compression algorithms can provide high compression ratios, and work in several ways:

- Many compression mechanisms require parameters, such as the table required for table compression or the parameters to decide what data to discard with lossy forms of compression. An adaptive algorithm can analyse the data it's about to compress, choose parameters accordingly, and store the parameters at the start of the compressed data.
- Other adaptive techniques adjust their parameters on the fly, according to the data compressed so far. For example Move-to-front (or MTF) transformations change the table used in, say Nibble Compression, so that the table of codes translating to the minimum (4-bit) representation is always the set of most recently seen characters.
- Further techniques, predominantly the Lempel-Ziv family of algorithms, use the stream of data already encoded as a string table to provide a compact encoding for each string newly received.

Implementations of many adaptive compression algorithms are available publicly, either as free or open source software, or from commercial providers.

For example, ThemePark:UK uses the *gzip* adaptive file compression algorithm for its text pages, which achieves typical compressions of 2.5 bits per character for English text, and requires fairly small amounts of RAM memory for decoding. ThemePark:UK also uses JPEG compression for its images, PNG compression for its maps and cartoons, and MP3 compression for sounds.

Consequences

Modern adaptive compression algorithms provide excellent compression ratios, reducing your *memory requirements*. They are widely used and are incorporated into *popular industry standards* for bulk data.

Adaptive compression can also reduce *data transmission times* for telecommunication. File compression can also reduce the *secondary storage* requirements or *data transmission times* for program code.

However: File compression can require a significant *processing time* to compress and decompress large bulk data sets, and so they are generally unsuitable for *real-time work*. Some *temporary memory* (primary and secondary storage) will be necessary to store the decompressed results and to hold intermediate structures.

The performance of compression algorithms can vary depending on the type of data being compressed, so you have to select your algorithm carefully, requiring *programmer effort*. If you cannot reuse an existing implementation you will need significant further *programmer effort* to code up one of these algorithms, because they can be quite complex. Some of the most important algorithms are *patented*, although you may able to use non-patented alternatives.

❖ ❖ ❖

Implementation

Designing efficient and effective adaptive compression algorithms is a very specialised task, especially as the compression algorithms must be tailored to the type of data being compressed. For most practical uses, however, you do not need to design you own text compression algorithms, as libraries of compression algorithms are available both commercially and under various open source licences. Sun's Java, for example, now officially includes a version of the *zlib* compression library, implementing the same compression algorithm as the *pkzip* and *gzip* compression utilities. In most programs, compressing and decompressing files or blocks of data is as simple as calling the appropriate routine from one of these libraries.

1. LZ Compression

Many of the most effective compression schemes are variants of a technique devised by Zip and Lempel [1977]. Lempel-Ziv (LZ77) compression uses the data already encoded as a table to allow a compact representation of following data. LZ compression is easy to implement; and decoding is fast and requires little extra temporary memory.

LZ77 works by encoding sequences of tuples. In each tuple, the first two items reference a string previously coded – as an offset from the current position, and a length. The third item is a single character. If there's no suitable string previously coded, the first two items are zero. For example, the following shows the LZ77 encoding of the song chorus "do do ron ron ron do do ron ron".

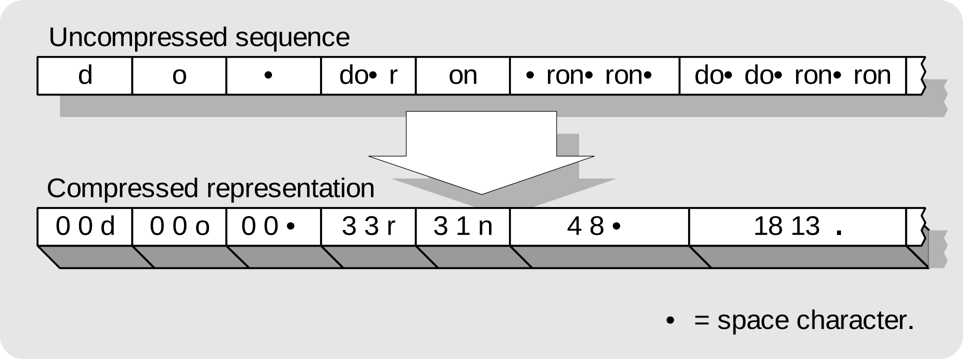


Figure 8: LZ77 Compression

Note how the repeating sequence “ ron ron” is encoded as a single tuple; this works fine for decompression and requires only a small amount of extra effort in the compression code.

There are many variants of LZ compression, adding other forms of compression to the output, or tailored for fast or low-memory compression or decompression. For example GZIP encodes blocks of 64Kb at a time, and uses Huffman Coding to compress the offset and length fields of each tuple still further.

Examples

We examine two examples of adaptive compression. The first, MTF compression, is a simple adaptive extension of Table Compression. The second, more typical of real-world applications, simply uses a library to do compression and decompression for us.

1. MTF Compression

Move-To-Front (MTF) compression can adapt Nibble Compression to the data being encoded, by changing the compression table dynamically so that it always contains the 15 most recently used characters [Bell et al 1990]. The following code shows only the significant changes from the Nibble Coding example in **TABLE COMPRESSION**.

First, we need a modifiable version of the table. As with the fixed version, it can be a simple string.

```
protected String NibbleChars = " etoasrinclmhdu";
```

To start off, we set the table to be a best guess, so both the `encodeString` and `decodeString` methods start by resetting `currentChars` to the value `NibbleChars` (not shown here). Then we simply need to modify the table after encoding each character, by calling the new method `updateCurrent` in `encodeChar`:

```
protected void encodeChar(int charCode) {
        int possibleNibble = NibbleChars.indexOf(charCode);
        if (possibleNibble != -1) {
            putNibble(possibleNibble);
        } else {
            putNibble(NibbleEscape);
            putNibble(charCode >>> 4);
            putNibble(charCode & 0xf);
        }
        updateCurrent((char) charCode);
    }
```

The `updateCurrent` method updates the current table, either by moving the current character to the front of the table. If that character is already in the table, it gets pushed to the front; if not, then the last (least recently used) character is discarded:

```
    protected void updateCurrent(int c)
        {
            int position = NibbleChars.indexOf(c);
            if (position != -1) {
                NibbleChars = "" + c + NibbleChars.substring(0, position) +
                    NibbleChars.substring(position+1);
            } else {
                position = NibbleChars.length() - 1;
                NibbleChars = "" + c + NibbleChars.substring(0, position);
            }
        }
```

The `decodeChar` needs to do the same update for each character decoded:

```
        protected int decodeChar() {
            int result;
            int nibble = getNibble();
            if (nibble == -1) {
                return -1;
            }
            if (nibble != NibbleEscape) {
                result = NibbleChars.charAt(nibble);
            } else {
                nibble = getNibble();
                if (nibble == -1) {
                    return -1;
                }
                result = (nibble << 4) + getNibble();
            }
            updateCurrent(result);
            return result;
        }
```

This example doesn't achieve as much compression as the fixed table for typical English text; for the text of this chapter it achieves only 6.2 bits per character. The MTF version does achieve some degree of compression on almost any non-random form of text, however, including executable binaries.

2. ZLIB Compression

This example uses an existing compression library, and so is more typical of real-world applications of adaptive compression. The Java Zlib libraries provide compressing streams that are **DECORATORS** of existing streams [Gamma et al 1995, Chan, Lee and Kramer 1998]. This makes it easy to compress any data that can be implemented as a stream. To compress some data, we open a stream on that data, and pass it through a compressing stream and then to an output stream.

```
    protected static byte[] encodeSequence(byte[] inputSequence)
                throws IOException {
            InputStream inputStream = new ByteArrayInputStream(inputSequence);
            ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
            GZIPOutputStream out = new GZIPOutputStream(outputStream);

            byte[] buf = new byte[1024];
            int len;
            while ((len = inputStream.read(buf)) > 0) {
                out.write(buf, 0, len);
            }
            out.close();
            return outputStream.toByteArray();
        }
```

In this model, decompressing is much like compressing. This time, the compressing stream is on the reading side; but in all other respects the code is virtually the same.

```
protected static byte [] decodeSequence(byte [] s) throws IOException {
        GZIPInputStream inputStream =
            new GZIPInputStream(new ByteArrayInputStream(s));
        ByteArrayOutputStream outputStream =
            new ByteArrayOutputStream();

        byte[] buf = new byte[1024];
        int len;
        while ((len = inputStream.read(buf)) > 0) {
            outputStream.write(buf, 0, len);
        }
        outputStream.close();
        return outputStream.toByteArray();
}
```

❖ ❖ ❖

Known Uses

Lempel-Ziv and variant compression algorithms are an industry standard, evidenced by the many `PKZip` and `gzip` file compression utilities used to reduce the size of email attachments, or to archive little-used or old versions of files and directories [Ziv and Lempel 1977, Deutsch 1996].

The PDF format for device-independent images uses LZ compression to reduce its file sizes [Adobe 1999] . Each PDF file contains one or more streams, each of which may be compressed with LZ.

File compression is also used architecturally in many systems. Linux kernels can be stored compressed and are decompressed when the system boots, and Windows NT supports optional file compression for each disk volume [Ward 1999, Microsoft NT 1996]. Java's JAR format uses gzip compression [Chan et al 1998] although designing alternative class file formats specially to be compressed can give two to five times better compression than gzip applied to the standard class file format [Horspool and Corless 1998, Pugh 1999]. Some backup tape formats use compression, notably the Zip and Jaz drives, and the HTTP protocol allows any web server to compress data, though as far as we are aware this feature is little used [Fielding 1999].

The current state-of-the-art library for adaptive file compression, Bzip2, achieves typical compressions of 2.3 bits per character on English text by transforming the text data before using LZ compression [Burroughs Wheeler 1994]. BZip2 requires a couple of Mbytes of RAM to compress effectively. See Witten et al [1999] and BZip2's home page [BZip2] for more detail.

See Also

TABLE COMPRESSION and **DIFFERENCE CODING** are often used with, or as part of, adaptive compression algorithms. You may also need to read a file a bit at a time (**DATA FILES)** to compress it.

Text Compression [Bell et al 1990] and *Managing Gigabytes* [Witten et al 1999] describe and analyse many forms of adaptive compression, including LZ compression, arithmetic coding and many others.

Small Data Structures

Version 13/06/00 02:15 - 33 by Charles Weir

How can you reduce the memory needed for your data?

- The *memory requirements* of the data exceed the memory available to the system.
- You want to increase *usability* by allowing users to store as much of their data as possible.
- You need to be able to *predict* the program's use of memory.
- You cannot delete some of the data from the program.

The fundamental difference between code and data is that programmers care about code while users care about data. Programmers have some direct control over the size of their code (after all, they write it!), but the data size is often completely out of the programmers' control. Indeed, given that a system is supposed to store users' data, any memory allocated to code, buffers, or other housekeeping is really overhead as far as the user is concerned. Often the amount of memory available to users can make or break a systems *usability* — a word processor which can store a hundred thousand word document is much more useful than one which are only store a hundred words.

Data structures that are appropriate where memory is unrestricted may be far too prodigal where memory is limited. For example, a typical implementation of an address database might store copies of information in indexes as well as the actual data, effectively storing everything in the database twice. Porting such an approach to the Strap-It-On wrist-top PC would halve the number of addresses that could be stored in the database.

Techniques like COMPRESSION and using SECONDARY STORAGE can reduce a program's main memory requirements, but both have significant liabilities when used to manage the data a program needs to work on. Many kinds of compressed data cannot be accessed randomly; if random access is required the data must be uncompressed first, costing time, and requiring a large amount of buffer memory for the uncompressed data. Data stored on secondary storage is similarly inaccessible, and needs to be copied into main memory buffers before it can be accessed.

Therefore: *Choose the smallest structure that supports the operations you need.*

For any given data set there are many different possible data structures that might support it. Suppose, for example, you need an unordered collection of object references with no duplicates – in mathematical terms, a set. You could implement it using a linear array of pointers, using a hash table, or using a variety of tree structures. Most class libraries will provide several different implementations of such collections; the best one to choose depends on your requirements. Where *memory is limited*, therefore, you must be particularly careful to choose a structure to minimise the program's *memory requirements*.

You can think of data structure design as a three-stage process. First analyse the program's requirements to determine the information the program needs to store; unnecessary information requires no memory!

Second, analyse the characteristics of the data; what's its total volume; how does it vary over a single program run and across different runs; and what's its granularity – does it consist of a few large objects or many small objects? You can also analyse the way you'll access the data: whether it is read and written, or only ever read; whether it is accessed sequentially or randomly; whether elements are inserted into the middle of the data or only added at the end.

Third, choose the data structures. Consider as many different possibilities as you can – your standard class libraries will provide some basic building blocks, but consider also options like embedding objects (FIXED ALLOCATION), EMBEDDING POINTERS, or PACKING the DATA. For each candidate data structure design, work out the amount of memory it will require to store the data you need, and check that it can support all the operations you need to perform. Then consider the benefits and disadvantages of each design: for example a smaller data structure may require more processing time to access, provide insufficient flexibility or give insufficient real-time performance. You'll need also to evaluate the resulting memory requirements for each possibility against the total amount of memory available – in some cases you may need to do simple trials using scratch code. If none of the solutions are satisfactory you may need to go back and reconsider your earlier analysis, or even the requirements of the system as a whole. On the other hand there's no need to optimise memory use beyond your given requirements (see the THRESHOLD SWITCH pattern [Auer and Beck 1996]).

For example, the Strap-It-On^{TM} address program has enough memory to store the address records but not indexes. So its version of the address program uses a sorted data structure that does not need extra space for an index but that is slower to access than the indexed version.

Consequences

Choosing suitable data structures can reduce a program's *memory requirements*, and the time spent can increase the *quality of the program's design.*

By increasing the amount of users' information the program can store, careful data structure design can increase a program's *usability*.

However: analysing a program's requirements and optimising data structure design takes *programmer discipline* to do, and *programmer effort* and time to do well.

Optimising data structure designs to suit limited memory situations can restrict a program's *scalability* should more memory become available.

The *predictability* of the program's memory use, the *testing costs*, and the program's *time performance* may or may not be affected, depending upon the chosen structures.

❖ ❖ ❖

Implementation

Every data structure design for any program must trade off several fundamental forces: *memory requirements*, *time performance*, and *programmer effort* being the most important. Designing data structures for a system with tight memory constraints is no different in theory from designing data structures in other environments, but the practical tradeoffs can result in different solutions. Typically you are prepared to sacrifice *time performance* and put in more *programmer effort* than in an unconstrained system, in order to reduce the *memory requirements* of the data structure.

There are several particular issues to consider when designing data structures to minimise memory use:

1. Predictability versus Exhaustion

The *predictability* of a data structure's memory use, and ultimately of the whole program can be as important as the structure's overall memory requirements, because making memory use more predictable makes it easier to manage. Predictability is closely related to the need to deal with *memory exhaustion:* if you can predict the program's maximum memory use in advance then you can use FIXED ALLOCATION to ensure the program will never run out of memory.

2. Space versus Flexibility

Simple, static, inflexible data structures usually require less memory than more complex, dynamic, and flexibly data structures. For example, you can implement a one-to-one relationship with a single pointer or even an inline object (see FIXED ALLOCATION), while a one-to-many relationship will require collection classes, arrays, or EMBEDDED POINTERS. Similarly, flexible collection classes require more memory than simple fixed sized arrays, and objects with methods or virtual functions require more memory than simple records (C++ `structs`) without them. If you don't need flexibility, don't pay for it; use simple data structures that need less memory.

3. Calculate versus Store

Often you can reduce the amount of main memory you need by calculating information rather than storing it. Calculating information reduces a program's time performance and can increase its power consumption, but can also reduce its memory requirements. For example, rather than keeping an index into a data structure, you can traverse the whole data structure using a linear search. Similarly, the PRESENTER pattern [Vlissides 1998] describes how graphical displays can be redrawn from scratch rather than being updated incrementally using a complex object structure.

❖ ❖ ❖

Specialised Patterns

This chapter contains five specialised patterns that describe a range of techniques for designing data structures to minimise memory requirements. The following figure shows the relationships between the patterns.

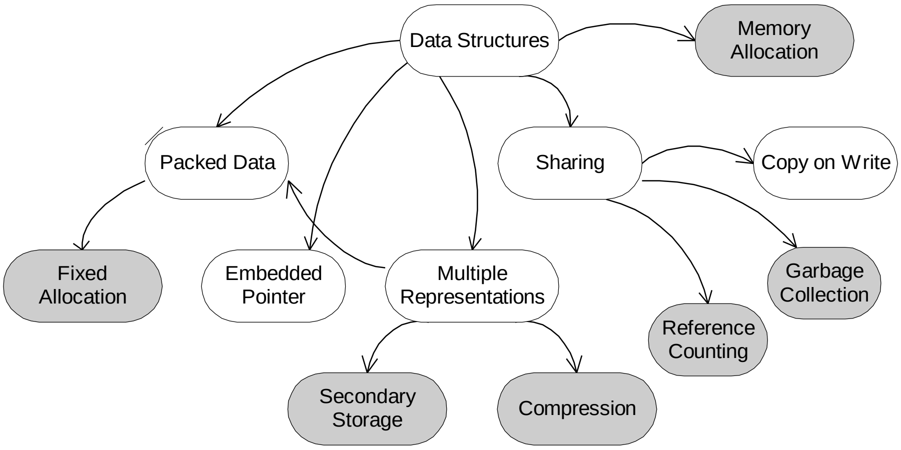


Figure 1: Data Structure Pattern Relationships

PACKED DATA selects suitable internal representations of the data elements in an object, to reduce its memory footprint.

SHARING removes redundant duplicated data. Rather than using multiple copies of functions, resources or data, the programmer can arrange to store only one copy, and use that copy wherever it is needed.

COPY-ON-WRITE extends SHARING so that shared objects can be modified without affecting other client objects that use the shared objects.

EMBEDDED POINTERS reduce the memory requirements for collection data structures, by eliminating auxiliary link objects and moving pointers into the data objects stored in the structures.

MULTIPLE REPRESENTATIONS are effective when no single representation is simultaneously compact enough for storage yet efficient enough for actual use.

Known Uses

Like Elvis, data structures are everywhere.

The classic example of designing data structures to save memory is the technique of allocating only two BCD digits to record the year when storing dates [Yourdon 2000]. This had unfortunate consequences, although not the disasters predicted in the lead-up to the millennium [Berry, Buck, Mills, Stipe 1987]. Of course these data structure designs were often made for the best of motives: in the 1960s disk and memory was much more expensive than it is today; and allocating two extra characters per record could cost millions.

An object-oriented database built using Smalltalk needed to be scaled up to cope with millions of objects, rather than several thousand. Unfortunately, a back-of-the envelope calculation showed that the existing design would require a ridiculous amount of disk space and thus buffer memory. Examination of the database design showed that Smalltalk Dictionary (hash table) objects occupied a large proportion of its memory; further investigation showed and that these Dictionaries contained only two elements: a date and a time. Redesigning the database to use Smalltalk Timestamp objects that stored a date and time directly, rather than the dictionary, reduced the number of objects needed to store each timestamp from at least eight to three, and made the scaled-up database project feasible.

See also

Once you have designed your data structures, you then have to allocate the memory to store them. The MEMORY ALLOCATION chapter presents a series of patterns describing how you can allocate memory in your programs.

In many cases, good data structure design alone is insufficient to manage your program's memory. The COMPRESSION chapter describes how memory requirements can be reduced by explicitly spending processor time to build very compact representations of data that generally cannot be used directly in computations. Moving less important data into SECONDARY STORAGE and constant data into READ-ONLY MEMORY can reduce the demand for writable primary storage further.

There are many good books describing data structure design in depth. Knuth [1997] remains a classic, though its examples are, effectively, in assembly language. Hoare [1972] is another seminal work, though nowadays difficult to find. Aho, Hopcroft and Ullman [1983] is a standard text for university courses, with examples in a Pascal-style pseudo-code, Cormen et al [1990] is a more in-depth Computer Science text, emphasising the mathematical analysis of algorithms. Finally Segewick's series beginning with *Algorithms* [1988] provide a more approachable treatment, with editions quoting source code in different languages – for example *Algorithms in C++: Fundamentals, Data Structures, Sorting, Searching,* [Segewick 1999]

Packed Data

Also known as: Bit Packing

How can you reduce the memory needed to store a data structure?

- You have a data structure (a collection of objects) that has significant memory requirements.
- You need fast random access to every part of every object in the structure.
- You need to store the data in these objects in main memory.

No matter what else you do in your system, sooner or later you end up having to design the low-level data structures to hold the information your program needs. In an object-oriented language, you have to design some key classes whose objects store the basic data and provide the fundamental operations on that data. In a program of any size, although there may be only a few key data storage classes, there can be a large number of instances of these classes. Storing all these objects can require large amounts of memory, certainly much more than storing the code to implement the classes.

For example, the Strap-It-On's Insanity-Phone application needs to store all of the names and numbers in an entire local telephone directory (200,000 personal subscribers). All these names and numbers should just about fit into the Strap-It-On's memory, but would leave no room for the program than displayed the directory, let alone any other program in the Wrist-Top PC.

Because these objects (or data structures) are the core of your program, they need to be easily accessible as your program runs. In a program of any complexity, the objects will need to be accessed randomly (rather than in any particular order) and then updated. Taken together, random access with updating requires that the objects are stored in main memory.

You might consider using COMPRESSION on each object or on a set of objects, but this would make processing slow and difficult, and makes random access to the objects using references almost impossible. Similarly, moving objects into SECONDARY STORAGE is not feasible if the objects need to be accessed rapidly and frequently. Considering the Insanity-Phone example again, the data cannot be placed in the Strap-It-On's secondary memory because that would be too slow to access; and the data cannot be compressed effectively while maintaining random access because each record is too small to compressed individually using standard adaptive compression algorithms.

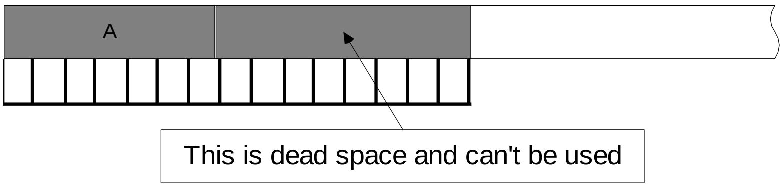
Therefore: *Pack data items within the structure so that they occupy the minimum space.*

There are two ways to reduce the amount of memory occupied by an object:

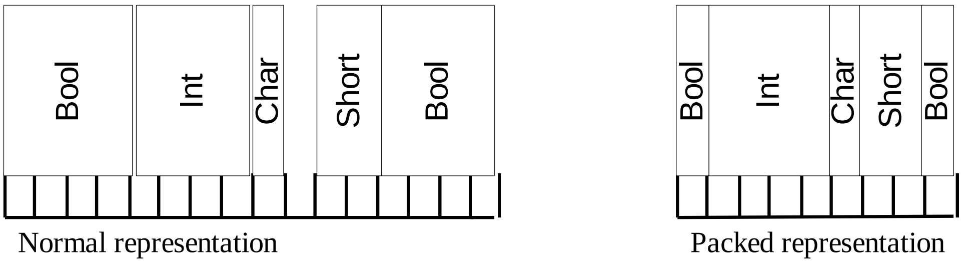
1. Reduce the amount of memory required by each field.
2. Reduce the amount of unused memory allocated between fields.

Consider each individual field in turn, and consider how much information that field really needs to store. Then, chose the smallest possible representation for that information. This may be the smallest suitable language level-data type, or even smaller, using different bits within, say, a machine word to encode different data items.

Once you have analysed each field, analyse the class as a whole to ensure that extra memory is not allocated between fields. Compilers or assemblers often ensure that fields are aligned to take advantage of CPU instructions that make it easier to access aligned data, so, for example, all two-byte fields be stored at even addresses, and all four-byte fields at addresses that are multiples of four. Aligning fields wastes the memory space between the end of one field and the start of the next.



The figure below shows how packing an object can almost halve the amount of memory that it requires. The normal representation on the left allocates four bytes for each Boolean variable (presumably to use faster CPU instructions) and aligns two and four-byte variables to two or four-byte boundaries; the packed representation allocates only one byte for each Boolean variable and dispenses with alignment for longer variables.



Considering the Insanity-Phone again, the designers realised that local phone books never cover more than 32 area codes – so each entry requires only 5 bits to store the area code. A seven-digit decimal number requires 24 bits. Surnames are duplicated many times, so Insanity-Phone stores each surname just once – an example of SHARING – and therefore gets less than 30,000 unique names in each book; this requires 18 bits. Storing up to three initials (5 bits each – see STRING COMPRESSION) costs a further 15 bits. The total is 62 bits, and this can be stored in one 64 bit long integer for each entry.

Consequences

Each instance occupies less memory reducing the total *memory requirements* of the system, even though the same amount of data can be stored, updated, and accessed randomly. Choosing to pack one data structure is usually a *local* decision, with little *global* effects on the program as a whole.

However: The *time performance* of a system suffers, because CPUs are slower at accessing unaligned data. If accessing unaligned data requires many more instructions than aligned data, it can impact the program's *power consumption*. More complicated packing schemes like bit packing can have even higher overheads.

Packing data requires *programmer effort* to implement, produces less intuitive code which is harder to *maintain*, especially if you use non-standard data representations. More complicated techniques can increase *testing costs*.

Packing schemes that rely on particular aspects of a machine's architecture, such as word sizes or pointer formats, will reduce *portability*. If you're using non-standard internal representations, it is harder to exchange objects with other programs that expect standard representations.

Finally, packing can reduce *scalability*, because it can be difficult to unpack data structures throughout a system if more memory becomes available.

❖ ❖ ❖

Implementation

The default implementation of a basic type is usually chosen for time performance rather than speed. For example, Boolean variables are often allocated as much space as integer variables, even though they need only a single bit for their representation. You can pack data by choosing smaller data types for variables; for example, you can represent Booleans using single byte integers or bit flags, and you may be able to replace full-size integers (32 or 64 bits) with 16 or even 8-bit integers (C++'s short and char types).

Compilers tend to align data members on machine-word boundaries which wastes space (see the figure on p.N above). Rearranging the order of the fields can minimise this padding, and can reduce the overhead when accessing non-aligned fields. A simple approach is to allocate fields within words in decreasing order of size.

Because packing has significant overheads in speed and maintainability, it is not worthwhile unless it will materially reduce the program's memory requirements. So, pack only the data structures that consume significant amounts of memory, rather than packing every class indiscriminately.

Here are some other issues to consider when applying the PACKED DATA pattern.

1. Compiler and Language Support

Compilers, assemblers and some programming language definitions support packing directly.

Many compilers provide a compilation flag or directive that ensures all data structures use the tightest alignment for each item, to avoid wasted memory at the cost of slower run-time performance. Microsoft C++, the directive:

```
#pragma pack( n )
```

sets the packing alignment to be based on n-byte boundaries, so pack(1) gives the tightest packing; the default packing 8 [Microsoft 1997]. G++ provides a `pack` attribute for individual fields to ensure they are allocated directly after the preceding field [Stallman 1999].

2. Packing Objects into Basic Types

Objects can impose a large memory overhead, especially when they contain only a small amount of data. Java objects impose an allocation overhead of at least one an additional pointer, and C++ objects with virtual functions require a pointer to a virtual function table. You can save memory by replacing objects by more primitive types (such as integers or pointers), an example of MULTIPLE REPRESENTATIONS.

When you need to process the data, wrap each primitive type in a first-class object, and use the object to process the data; when you've completed processing, discard the object, recover the basic type, and store it once again. To avoid allocating lots of wrapper objects, you can reuse the same wrapper for each primitive data item. The following Java code sketches how a `BigObject` can be repeatedly initialised from an array of integers for processing. The `become` method reinitialises a `BigObject` from its argument, and the `process` method does the work.

```
BigObject obj = new BigObject(0);

    for (int i=1; i<10; i++)
    {
        obj.become(bigarray[i]);
        obj.process();
    }
```

In C++, we can define operators to convert between objects and basic types, so that the two can be used interchangeably:

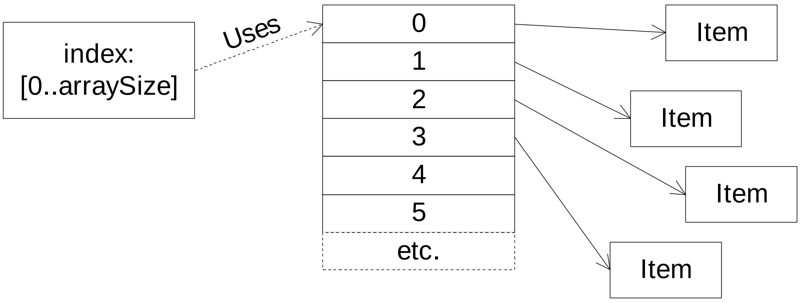
```
class BigIntegerObject
{
public:
   BigIntegerObject( int anInt=0 ) : i( anInt ) {}
   operator int() { return i; }
private:
   int i;
};

int main()
{
   BigIntegerObject i( 2 ), j( 3 );
   BigIntegerObject k = i*j;  // Uses conversion operators
```

3. Packing Pointers using Tables

Packing pointers is more difficult, because they don't obviously contain redundant data. To pack pointers you need to look at what they reference.

If a given pointer may point to only one of a given set of then it may be possible to replace the pointer with an index into an array; often, an array of pointers to the original item. Since the size of the array is usually much less than the size of all the memory in the system, the number of bits needed for an array index can be much less than the number of bits needed for a general pointer.



Take, for example, the Insanity-phone application above. Each entry apparently needs a pointer to the corresponding surname string: 32 bits, say. But if we implement an additional array of pointers into the string table, then each entry need only store an index into this array (16 bits). The additional index costs memory: 120Kb using 4-byte pointers.

If you know that each item is guaranteed to be within a specific area of memory, then you can just store offsets within this memory. This might happen if you've built a string table, or if you're using POOLED ALLOCATION, or VARIABLE ALLOCATION within a heap of known size. For example, if all the Insanity-Phone surname strings are stored in a contiguous table (requiring less than 200K with STRING COMPRESSION), the packed pointer needs only hold the offset from the start of the table: 18 bits rather than 32.

4. Packing Pointers using Bitwise Operations

If you are prepared to sacrifice portability, and have an environment like C++ that allows you to manipulate pointers as integers, then you have several possible ways to pack pointers. In some architectures, pointers contain redundant bits that you do not need to store. Long pointers in the 8086 architecture had at least 8 redundant bits, for example, so could be stored in three bytes rather than four.

You can further reduce the size of a pointer if you can use knowledge about the heap allocation mechanism, especially about alignment. Most memory managers allocate memory blocks aligned on word boundaries; if this is an 8-byte boundary, for example, then you can know that any pointer to a heap object will be a multiple of eight. In this case, the lowest three bits of each pointer are redundant, and can be reused or not stored. Many garbage collectors, for example, pack tag information into the low bits of their pointers [Jones and Lins 1996].

Example

Consider the following simple C++ class:

```
class MessageHeader {
    bool  inUse;
    int   messageLength;
    char  priority;
    unsigned short channelNumber;
    bool  waitingToSend;
};
```

With 8-byte alignment, this occupies 16 bytes, using Microsoft C++ on Windows NT. With the compiler packing option turned on it occupies just 9 bytes. Note that the packed structure does not align the integer `i1` to a four-byte boundary, so on some processors it will take longer to load and store.

Even without compiler packing, we can still improve the memory use just by reordering the data items within the structure to minimise the gaps. If we sort the fields in decreasing order of size

```
class ReorderedMessageHeader {
    int   messageLength;
    unsigned short channelNumber;
    char  priority;
    bool  inUse;
    bool  waitingToSend;
};
```

the class occupies just 12 bytes, a saving of four bytes. If you're using compiler field packing, both `MessageHeader` and `ReorderedMessageHeader` occupy 9 bytes, but there's still a benefit to latter since it puts all the member items on the correct machine boundaries where they can be manipulated fast.

We can optimise the structure even more using bitfields. The following version contains the same data as before:

```
class BitfieldMessageHeader {
    int      messageLength;
    unsigned channelNumber: 16;
    unsigned priority:      8;
    unsigned inUse:         1;
    unsigned waitingToSend: 1;

public:
    bool IsInUse() { return inUse; }
    void SetInUseFlag( bool isInUse ) { inUse = isInUse; }
    char Priority() { return priority; }
    void SetPriority( char newPriority ) { priority = newPriority; }
    // etc.
};
```

but occupies just 8 bytes, a further saving of four bytes – or one byte if you're using compiler packing.

Unfortunately compiler support for booleans in bitfields tends to be inefficient. This problem isn't actually a sad reflection on the quality of C++ compiler writers today; the real reason is that it requires a surprising amount of code to implement the semantics of, say, the `setB1` function above. We can improve performance significantly by using bitwise operations instead of bitfields, and implement the member functions directly to expose these operations:

```
class BitwiseMessageHeader {
    int      messageLength;
    unsigned short channelNumber;
    char priority;
    unsigned char flags;
public:
    enum FlagName { InUse = 0x01, WaitingToSend = 0x02 };
    bool GetFlag( FlagName f )   { return (flags & f) != 0; }
    void SetFlag( FlagName f )   { flags |= f;  }
    void ResetFlag( FlagName f ) { flags &= ~f;  }
};
```

This optimises performance, at the cost of exposing some of the implementation.

❖ ❖ ❖

Known Uses

Packed data is ubiquitous in memory-limited systems. For example virtually all Booleans in the EPOC system are stored as bit flags packed into integers. The Pascal language standard includes a special `PACKED` data type qualifier, used to implement the original Pascal compiler.

To support dynamic binding, a C++ object normally requires a vtbl pointer to support virtual functions [Stroustrup 1995, Stroupstrup 1997]. EPOC requires dynamic binding to support Multiple Representations for its string classes, but a vtbl pointer would impose a four bytes overhead on every string. The EPOC string base class (`TDesC`) uses the top 4 bits of its 'string length' data member to identify the class of each object:

```
class TDesC8 {  private:
   unsigned int iLength:28;
   unsigned int iType:4;
   /* etc... */
```

Dynamically bound functions that depend on the actual string type are called from `TDesC` using a switch statements on the value of the `iType` bitfield.

Bit array classes are available in both the C++ Standard Template Library and the Java Standard Library. Both implement arrays of bits using array of machine words. Good implementations of the C++ STL also provide a template specialisation to optimise the special case of an array of Booleans by using a bitset [Stroustrup 1997, Chan et al 1998].

Java supports object wrapper versions for many primitive types (Integer, Float). Programmers typically use the basic types for storage and the object versions for complicated operations. Unfortunately Java collections store objects, not basic types, so every basic type must be wrapped before it is stored into a collection [Gosling, Joy, Steele 1996]. To avoid storing multiple copies of the same information the Palm Spotless JVM carefully shares whatever objects it can, such as constant strings defined in different class files [Taivalsaari et al 1999].

See Also

EMBEDDED POINTERS provides a way to limit the space overhead of collections and similar data structures. FIXED ALLOCATION and POOLED ALLOCATION provide ways to reduce any additional memory management overhead.

Packing string data often requires STRING COMPRESSION.

The VISITOR and PRESENTER [Vlissides 1996] patterns can provide behaviour for collections of primitive types (bit arrays etc.) without having to make each basic data item into an object. The FLYWEIGHT PATTERN [Gamma et al 1995] allows you to process each item of a collection of packed data within the context of its neighbours.

Sharing

Also Known As: Normalisation.

How can you avoid multiple copies of the same information?

- The same information is repeated multiple times.
- Very similar objects are used in different components.
- The same functions can be included in multiple libraries
- The same literal strings are repeated throughout a program.
- Every copy of the same information uses memory.

Sometimes the same information occurs many times throughout a program, increasing the program's *memory requirements.* For example, the Strap-It-On user interface design includes many icons showing the company's bespectacled founder. Every component displaying the icon needs to have it available, but every copy of that particular gargoyle wastes memory.

Duplication can also enter the program from outside. Data loaded from RESOURCE FILES or from an external database must be recreated inside a program, so loading the same resource or data twice (possibly in different parts of the program) will also result in two copies of the same information.

Copying objects has several benefits. Architecturally, it is important that components take responsibility for the objects they use, so copying objects between components can simplify ownership relationships. Some language constructs (such as C++ value semantics and Smalltalk cloning) assume object copying; and sometimes copying objects to where they are required can avoid indirection, making systems run faster.

Unwanted duplication doesn't just affect data objects. Unless care is taken, every time a separately compiled or built component of the program uses a library routine, a copy of that routine will be incorporated into the program. Similarly every time a component uses a string or a constant a copy of that string may be made and stored somewhere.

Unfortunately, for whatever reason information is duplicated, every copy takes up memory that could otherwise be put to better use.

Therefore: *Store the information once, and share it everywhere it is needed.*

Analyse your program to determine which information is duplicated, and which information can be safely shared. Any kind of information can be duplicated — images, sounds, multimedia resources, fonts, character tables, objects, and functions, as well as application data.

Once you have found common information, check that it can be shared. In particular, ensure that it never needs to be changed, or that all its clients can cope whenever it is changed. Modify the information's clients so that they all refer to a single shared copy of the information, typically by accessing the information through a pointer rather than directly.

If the shared information can be discarded by its clients, you may need to use REFERENCE COUNTING or GARBAGE COLLECTION so that it is only released once it is no longer needed anywhere in the program. If individual clients may want to change the data, you may need to use COPY-ON-WRITE.

For example, the Strap-It-On PC really only needs one copy of Our Founder's bitmap. This bitmap is never modified, so a single in-memory copy of the bitmap is shared everywhere it is needed in the system.

Consequences

Judicious sharing can reduce a program's *memory requirements,* because only one copy is required of a shared object. SHARING also increases the *quality* of the design, since there's less chance of code duplication. SHARING generally does not affect a program's *scalability* when more memory is made available to the system, nor its *portability*. Since there's no need to allocate space for extra duplicate copies, SHARING can reduce *start-up times*, and to a lesser extent *run-time performance*.

However: *programmer effort* and *discipline*, and *team co-ordination* is required to design programs to take advantage of sharing. Designing sharing also increases the complexity of the resulting system, adding to *maintenance* and *testing* costs since shared objects create interactions between otherwise independent components.

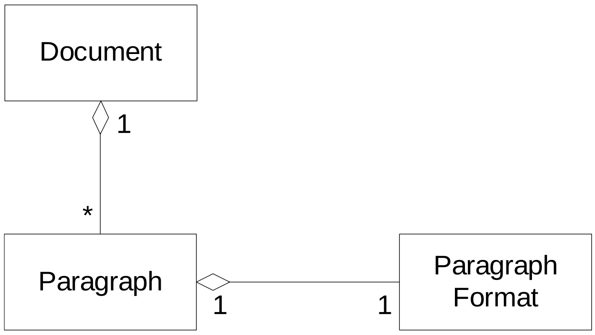
Although it does not affect the *scalability* of centralised systems, sharing can reduce the *scalability* of distributed systems, since it can be more efficient to make one copy of each shared object for each processor.

Sharing can introduce many kinds of aliasing problems, especially when read-only data is changed accidentally [Hogg 1991, Noble et al 1998], and so can increase *testing costs*. In general, sharing imposes a *global* cost on programs to achieve *local* goals, as many components may have to be modified to share a duplicated data structure.

❖ ❖ ❖

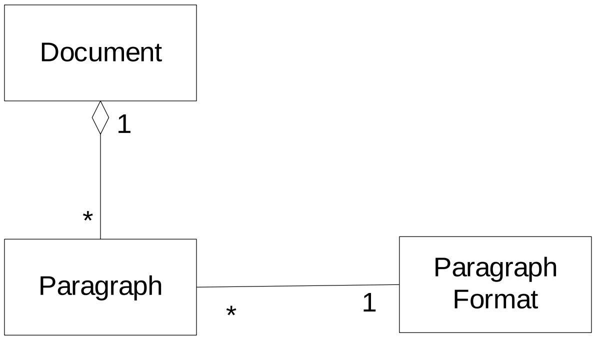
Implementation

Sharing effectively changes one-to-one (or one-to-many) relationships into many-to-one (or many-to-many) relationships. Consider the example below, part of a simple word processor, in UML notation [Fowler 1997]. A `Document` is made up of many `Paragraphs`, and each `Paragraph` has a `ParagraphFormat`.



Considering each class in turn, it is unlikely that `Documents` or `Paragraphs` will be duplicated, unless, for example, many documents have many identical paragraphs. Yet many paragraphs within a document will have the same format. This design, however, gives each `Paragraph` object has its own `ParagraphFormat` object. This means that the program will contain many `ParagraphFormat` objects (one for each `Paragraph`) whilst many of these objects will have exactly the same contents. `ParagraphFormats` are obvious candidates for sharing between different `Paragraphs`.

We can show this sharing as a one-to-many relationship.



In the revised design, there will only be a few `ParagraphFormat` objects, each one different, and many `Paragraphs` will share the same `ParagraphFormat` object.

The design activity of detecting repeated data and splitting it into a separate object is called *normalisation.* Normalisation is an essential part of relational database design theory, and boasts an extensive literature [Connolly and Begg 1999; Date 1999; Elmasri and Navathe 2000].

You should consider the following other issues when applying the SHARING **pattern**.

1. Making Objects Shareable

Aliasing problems make it difficult to share objects in object-oriented programs [Hogg 1991, Noble, Vitek, Potter 1998]. Aliasing problems are the side effects caused by changing a shared object: if a shared object is changed by one of its clients the change will affect any other client of the shared object, and such changes can cause errors in clients that do not expect them. For example, changing the font in a shared Paragraph Format object will change the fonts for all Paragraphs that share that format. If the new font is, say, a printer-only font, and is changed to suit one particular paragraph that will never be displayed on screen, it will break other paragraphs using that format which do need to be displayed on screen, because a printer-only font will not work for them.

The only kinds of objects that can be shared safely without side effects are *immutable* objects, objects that can never be changed. The immutability applies to the object itself – it's not enough just to make some clients read-only, since other, writing, clients may still change the shared object 'behind their back'. To share objects safely you typically have to change clients so that they make new objects rather than changing existing ones (see COPY-ON-WRITE). You should consider removing any public methods or fields that can be used to change shared objects' state: in general, every field should only be initialised in the object's constructor (in Java, all fields should be `final`). The FLYWEIGHT pattern [Gamma 1995] can be used to move dynamic state out of shared objects and into their clients.

2. Establishing Sharing

For two or more components to be able to share some information, each component must be able to find the information that is shared.

In small systems, where only a few distinguished objects are being shared, you can often use a global variable to store each shared object, or store shared instances within the objects' classes using the SINGLETON pattern [Gamma 1995]. The shared global variables can be initialised statically when the program starts, or the first time a shared objects is accessed using LAZY INITIALISATION [Beck 1997].

To provide a more general mechanism you can implement a *shared cache*, an in-memory database mapping from keys to shared objects. To find a shared object, a component checks the cache. If the shared object is already in the cache you use it directly; if not you create the object and store it back into the cache. For this to work you need a unique key for each shared object to identify it in the cache. A shared cache works particularly well when several components are loading the same objects from resource files databases, or networks like the world wide web. Typical keys could be the fully qualified file names, web page URLs, or a combination of database table and database key within that table – the same keys that identify the data being loaded in the first place.

3. Deleting Shared Objects

Once you’ve created shared objects, you may need to be able to delete them when no longer required.

There are three standard approaches to deleting shared objects: Reference Counting, Garbage Collection and object ownership. Reference Counting keeps a count of the number of objects interested in a shared object; when this becomes zero the object can be released (by removing the reference from the cache and, in C++, deleting the object). A Garbage Collector can detect and remove shared objects without requiring reference counts. Note that if a shared object is accessed via a cache, the cache will always have a reference to the shared object, preventing the garbage collector from deleting it, unless you can use some form of weak reference [Jones and Lins 1996].

With object ownership, you can identify one other single object or component that has the responsibility of managing the shared object (see the Small Architecture **pattern)**. The object’s owner accepts the responsibility of deleting the shared object at the appropriate time; generally it needs to be an object with an overview of all the objects that ‘use’ the shared object [Weir 1996, Cargill 1996].

4. Sharing Literals and Strings

In many programs literals occupy more space than variables, so you can assign often used literals to variables and then replace the literals by the variables, effectively Sharing one literal in many places. For example, LaTeX uses this technique, coding common literals such as one, two, and minus one as the macros ‘`\@ne`’, ‘`\tw@`’, and ‘`\m@on'`. Smalltalk shares literals as part of the language environment, by representing strings as ‘symbols’. A symbol represents a single unique string, but can be stored internally as if it were an integer. The Smalltalk system maintains a ‘symbol table’ that maps all known symbols to the strings they represent, and the compilation and runtime system must search this table to encode each string as a symbol, potentially adding a new entry if the string has not been presented before. The Smalltalk environment uses symbols for all method names, which both compresses code and increases the speed of method lookup.

5. Sharing across components and processes

It’s more difficult to implement sharing between several components in different address spaces. Most operating systems provide some kind of shared memory, but this is often difficult to use. In concurrent systems, you need to prevent one thread from modifying shared data while another thread is accessing it. Typically this requires at least one semaphore, and increases code complexity and testing cost.

Alternatively, especially when data is shared between many components, you can consider encapsulating the shared data in a component of its own and use client-server techniques to access it. For example, EPOC accesses its relational databases through a single ‘database server’ process. This server keeps a cache of indexes for its open databases; if two applications use the same database they share the same index.

Example

This Java example outlines part of the simple word processor described above. Documents contain `Paragraphs`, each of which has a `ParagraphFormat`. `ParagraphFormats` are complex objects, so to save memory several `Paragraphs` share a single `ParagraphFormat`. The code shows two mechanisms to ensure this:

- When we duplicate a `Paragraph`, both the original and the new `Paragraph` share a single `ParagraphFormat` instance.
- `ParagraphFormats` are referenced by name, like "bold", "normal" or "heading 2". A Singleton `ParagraphFormatCatalog` contains a map of all the names to `ParagraphFormat` objects, so when we request a `ParagraphFormat` by name, the result is the single, shared, instance with that name.

The most important class in the word processor is Document: basically a sequence of Paragraphs, each of which as a (shared) `ParagraphFormat`.

```
class Document {
    Vector paragraphs = new Vector();
    int currentParagraph = -1;
```

The `Paragraph` class uses a `StringBuffer` to store the text of the paragraph, and also stores a reference to a `ParagraphFormat` object.

```
class Paragraph implements Cloneable {
    ParagraphFormat format;
    StringBuffer text = new StringBuffer();
```

A new `Paragraph` can be constructed either by giving a reference to a format object (which is then stored, without being copied, as the new Paragraph's format) or by giving a format name, which is then looked up in the `ParagraphFormatCatalog`. Note that neither initialising or accessing a paragraph's format copies the `ParagraphFormat` object, rather it is passed by reference.

```
Paragraph(ParagraphFormat format) {
        this.format = format;
    }

    Paragraph(String formatName) {
        this(ParagraphFormatCatalog.catalog().findFormat(formatName));
    }

    ParagraphFormat format() {return format;}
```

`Paragraphs` are copied using the clone method (used by the word-processor to implement its cut-and-paste feature). The clone method only copies one object, so the new clone's fields automatically point to exactly the same objects as the old object's fields. We don't want a `Paragraph` and its clone to share the `StringBuffer`, so we must clone that explicitly and install the cloned `StringBuffer` into the cloned `Paragraph`; however we don't want to clone the `ParagraphFormat` reference, because `ParagraphFormats` can be shared.

```
    public Object clone() {
        try {
            Paragraph myClone = (Paragraph) super.clone();
            myClone.text =  new StringBuffer(text.toString());
            return myClone;
        } catch (CloneNotSupportedException ex) {
            return null;
        }
    }
```

Paragraphs find their formats using the `ParagraphFormatCatalog`, The catalog is a SINGLETON [Gamma et al 1995].

```
class ParagraphFormatCatalog {
    private static ParagraphFormatCatalog systemWideCatalog
        = new ParagraphFormatCatalog();
    public static ParagraphFormatCatalog catalog() {
        return systemWideCatalog;
    }
```

that implements a map from format names to shared `ParagraphFormat` objects:

```
Hashtable theCatalog = new Hashtable();
    public void addNewNamedFormat(String name, ParagraphFormat format) {
        theCatalog.put(name,format);
    }

    public ParagraphFormat findFormat(String name) {
        return (ParagraphFormat) theCatalog.get(name);
    }
}
```

Since the `ParagraphFormat` objects are shared, we want to restrict what the clients can do with them. So `ParagraphFormat` itself is just an interface that does not permit clients to change the underlying object.

```
interface ParagraphFormat {
    ParagraphFormat nextParagraphFormat();
    String defaultFont();
    int fontSize();
    int spacing();
}
```

The class `ParagraphFormatImplementation` actually implements the `ParagraphFormat` objects, and includes a variety of accessor methods and constructors for these variables:

```
class ParagraphFormatImplementation implements ParagraphFormat {
    String defaultFont;
    int fontSize;
    int spacing;
    String nextParagraphFormat;
```

Each `ParagraphFormat` object stores the name of the `ParagraphFormat` to be used for the next paragraph. This makes it easier to initialise the `ParagraphFormat` objects, and will give the correct behaviour if we replace a specific `ParagraphFormat` in the catalogue with another.

To find the corresponding `ParagraphFormat` object, it must also refer to the catalogue

```
public ParagraphFormat nextParagraphFormat() {
        return ParagraphFormatCatalog.catalog().
            findFormat(nextParagraphFormat);
}
```

When the `Document` class creates a new paragraph, it use the shared `ParagraphFormat` returned by the format of the current paragraph: Note that `ParagraphFormat` objects are never copied, so the will be shared between all paragraphs that have the same format.

```
public Paragraph newParagraph() {
        ParagraphFormat nextParagraphFormat =
            currentParagraph().format().nextParagraphFormat();
        Paragraph newParagraph = new Paragraph(nextParagraphFormat);
        insertParagraph(newParagraph);
        return newParagraph;
    }
```

❖ ❖ ❖

Known Uses

Java’s String instances are immutable, so implementations share a single underlying buffer between any number of copies of the same String object [Gosling et al 1996]. And all implementations of Smalltalk use tokens for pre-compiled strings, as discussed above.

C++'s template feature often generate many copies of very similar code, leading to 'code bloat'. Some C++ linkers detect and share such instances of duplicated object code – Microsoft, for example, call this 'COMDAT folding' [Microsoft 1997]. Most modern operating systems provide Dynamic Link Libraries (DLLs) or Shared libraries that allow different processes to use the same code without needing to duplicate it in every executable [Kenah and Bate 1984; Card et al1998].

The EPOC Font and Bitmap server stores font and image data loaded from RESOURCE FILES in shared memory [Symbian 1999]. These are used both by applications and by the Window Server that handles screen output for all applications. Each client requests and releases the font and bitmap data using remote procedure calls to the server process; the server loads the data into shared memory or locates an already-loaded item, and thereafter the application can access it directly (read-only). The server uses reference counting to decide when to delete each item; an application will normally release each item explicitly but the EPOC operating system will also notify the server if the application terminates abnormally, preventing memory leaks.

See Also

SHARING was first described as a pattern in the *Design Patterns Smalltalk Companion* [Alpert, Brown, Woolf 1998].

COPY-ON-WRITE provides a mechanism to change a shared object as seen by one object, without impacting any other objects that rely on it. The READ-ONLY MEMORY pattern describes how you can ensure that objects supposed to be read-only cannot be modified. Shared things are often READ-ONLY, and so often end up stored on SECONDARY STORAGE.

The FLYWEIGHT PATTERN [Gamma et al 1995] describes how to make objects read-only so that they can be shared safely. Objects may also need to be moved into a SINGLE PLACE for modelling reasons [Noble 1997]. Ken Auer and Kent Beck [1996] describe techniques to avoid sharing Smalltalk objects by accident.

Copy-on-Write

How can you change a shared object without affecting its other clients?

- You need the system to behave as if each client has its own mutable copy of some shared data.
- To save memory you want to share data, or
- You need to modify data in Read-Only Memory.

Often you want the system to behave as though there are lots of copies of a piece of shared data, each individually modifiable, even though there is only one shared instance of the data. For example, in the Word-O-Matic word processor, each paragraph has its own format, which users can change independently of any other paragraph. Giving every paragraph its own `ParagraphFormat` object ensures this flexibility, but duplicates data unnecessarily because there are only a few different paragraph formats used in most documents.

We can use the SHARING pattern instead, so that each paragraph format object describes several paragraphs. Unfortunately, a change to one shared paragraph format will change all the other paragraphs that share that format, not just the single paragraph the user is trying to change.

There's a similar problem if you're using READ-ONLY MEMORY (ROM). Many operating systems load program code and read-only data into memory marked as 'read-only', allowing it to be shared between processes; In palmtops and embedded systems the code may be loaded into ROM or flash RAM. Clients may want changeable copies of such data; but making an automatic copy by default for every client will waste memory.

Therefore: *Share the object until you need to change it, then copy it and use the copy in future.*

Maintain a flag or reference count in each sharable object, and ensure it's set as soon as there's more than one client to the object. When a client calls any method that modifies a shared object's externally visible state, create a duplicate of some or all of the object's state in a new object, delegate the operation to that new object, and ensure that the the client uses the new object from then on. The new object will initially not be shared (with flag unset or reference count of one), so further modifications won't cause a copy until the new object in turn then gets multiple clients.

You can implement COPY-ON-WRITE for specific objects in the system, or implement it as part of the operating system infrastructure using PAGING techniques. The latter approach is particularly used with code, which is normally read-only but allows a program to modify its own code on occasion, in which case a paging system can make a copy of part of the code for that specific program instance.

Thus Word-O-Matic keeps a reference count of the number of clients sharing each `ParagraphFormat` object. In normal use many `Document` objects will share the same `ParagraphFormat`, but on the few occasions that a user modifies the format of a paragraph, Word-O-Matic makes a copy of its `ParagraphFormat` and keeps that separate to the modified `Paragraph` and to any other `Paragraphs` with the new format.

Consequences

COPY-ON-WRITE gives programmers the illusion of many copies of a piece of data, without the waste of memory that would imply. So it reduces the *memory requirements* of the system. In some cases it increases a program's *execution speed*, and particularly its *start-up time*, since copying can be a slow operation.

COPY-ON-WRITE also allows you to make it appear that data stored in read-only storage can be changed. So you can move infrequently changed data into read-only storage, reducing the program's *memory requirements.*

Once COPY-ON-WRITE has been implemented it requires little *programmer discipline* to use, since clients don't need to be directly aware of it.

However: COPY-ON-WRITE requires *programmer effort* or *hardware or operating system* support to implement, because the system must intercept writes to the data, make the copy and then continue the write as if nothing had happened.

If there are many write accesses to the data, then COPY-ON-WRITE can decrease *time performance*, since each write access must ensure the data's not shared. COPY-ON-WRITE can also lead to lots of copies of the same thing cluttering up the system, decreasing the *predictability* of the system's performance, making it *harder to test*, and ultimately increasing the system's *memory requirements.*

COPY-ON-WRITE can cause problems for object identity if the identity of the copy and the original storage is supposed to be the same.

❖ ❖ ❖

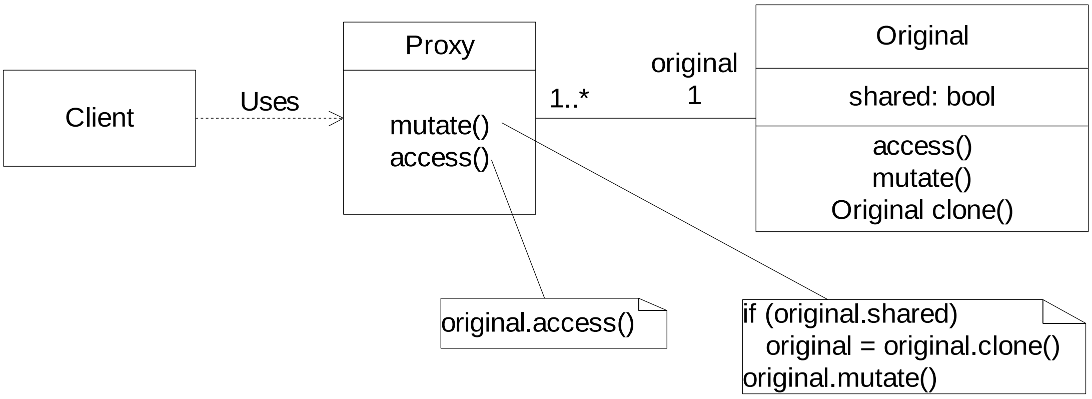
Implementation

Here are some issues to consider when implementing the COPY-ON-WRITE pattern.

1. Copy-On-Write Proxies

The most common approach to implementing COPY-ON-WRITE is to use a variant of the PROXY Pattern [Gamma et al 1995, Buschmann et al 1996, Coplien 1994]. Using the terminology of Bushman et al [1996], a PROXY references an underlying *Original* object and forwards all the messages it receives to that object.

To use PROXY to implement COPY-ON-WRITE, every client uses a different PROXY object, which distinguishes *accessors*, methods that merely read data, from *mutators* that modify it. The Original Object contains a flag that records whether it has more than one Proxy sharing it. Any updator method checks this flag and if the flag is set, makes a (new, unshared) copy of the representation, installs it in the proxy, and forwards the mutator to the that copy instead.



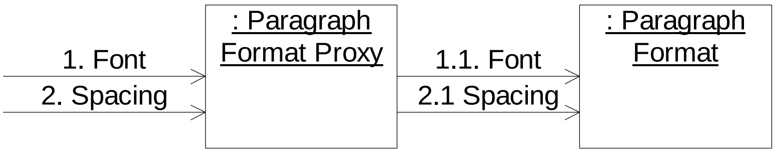
In this design, the `shared` flag is stored in the Original object, and it's the responsibility of the representation's `clone` method to create an object with the flag unset. A valid alternative implementation is to place the flag into the Proxy object; in this case the Proxy must reset the flag after creating and installing a new Original object. As a third option, you can combine the Client and Proxy object, if the Client knows about the use of COPY-ON-WRITE, and if no other objects need to use the Original (other than via the combined Client/Proxy, of course).

You can also combine the function of COPY-ON-WRITE with managing the lifetime of the underlying representation object by replacing the `shared` flag in the Representation object with a reference count. A reference count of exactly one implies the object is not shared and can be modified. See the Reference Counting pattern for a discussion of reference counting in detail.

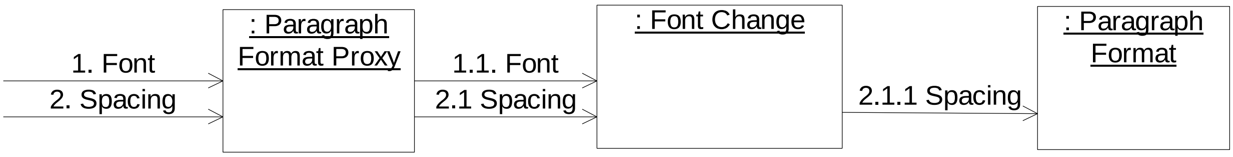
2. Copying Changes to Objects

You do not have to copy all of any object when it is changed. Instead you can create a 'delta' object that stores only the changes to the object, and delegates requests for unchanged data back to the main object. For example, when a user changes the font in a paragraph format, you can create a `FontChange` delta object that returns the new font when it is asked, but forwards all other requests to the underlying, and unchanged, `ParagraphFormat` object. A delta object can be implement as a Decorator on the original object [Gamma et al 1995]. The diagram below uses UML shows a possible implementation as a UML Collaboration Diagram [Fowler 1997].

Before:



After:



3. Writing to Objects in Read-Only Memory

You can use Copy-On-Write so that shared representations in ROM can be updated. Clearly the `shared` flag must be set in the ROM instance and cleared in the copy, but otherwise this is no different from a RAM version of the pattern.

In C++ the rule is that only instances of classes without a constructor may be placed in ROM. So a typical implementation must use static initialisation for the flag, and must therefore have public data members. The restriction on constructors means that you can't implement a copy constructor and assignment operator; instead you'll need to write a function that uses the default copy constructor to copy the data.

Example

This example extends the word processor implementation from the Sharing pattern, to allow the user to change the format of an individual paragraph. In this example the `Paragraph` object combines the role of Proxy and Client, since we've restricted all access to the `ParagraphFormat` object to via the `Paragraph` object. We don't need to separate out the read-only aspects to a separate interface as no clients will ever see `ParagraphFormats` directly.

The `Document` class remains unchanged, being essentially a list of paragraphs. The `ParagraphFormat` class is also straightforward, but now it supports mutator methods and needs to implement the `clone` method. For simplicity we only show one mutator – to set the font.

```
class ParagraphFormat implements Cloneable {
    String defaultFont;
    int fontSize;
    int spacing;
    String nextParagraphFormat;

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    void privateSetFont(String aFont) {defaultFont = aFont;}
}
```

As in the previous example, the Paragraph class must also implement cloning. This implementation keeps the shared flag in the `Paragraph` class (i.e. in the Proxy), as the member `paragraphFormatIsUnique`.

```
class Paragraph implements Cloneable {
    ParagraphFormat format;
    boolean paragraphFormatIsUnique = false;

    StringBuffer text = new StringBuffer();

    Paragraph(ParagraphFormat format) {
        this.format = format;
    }
    Paragraph(String formatName) {
        this(ParagraphFormatCatalog.catalog().findFormat(formatName));
    }
```

The `Paragraph` implementation provides two private utility functions: `aboutToShareParagraphFormat` and `aboutToChangeParagraphFormat`. The method `aboutToShareParagraphFormat` should be invoked whenever we believe it's possible that we may be referencing a `ParagraphFormat` object known to any other object.

```
    protected void aboutToShareParagraphFormat() {
        paragraphFormatIsUnique = false;
    }
```

If any external client obtains a reference to our `ParagraphFormat` object, or passes in one externally, then we must assume that it's shared:

```
    ParagraphFormat format() {
        aboutToShareParagraphFormat();
        return format;
    }

    public void setFormat(ParagraphFormat aParagraphFormat) {
        aboutToShareParagraphFormat();
        format = aParagraphFormat;
    }
```

And similarly, if a client clones this `Paragraph` object, we don't want to clone the format, but instead simply note that we're sharing it:

```
    public Object clone() {
        try {
            aboutToShareParagraphFormat();
            Paragraph myClone = (Paragraph) super.clone();
            myClone.text =  new StringBuffer(text.toString());
            return myClone;
        } catch (CloneNotSupportedException ex) {
            return null;
        }
    }
```

Meanwhile any method that modifies the `ParagraphFormat` object must first call `aboutToChangeParagraphFormat`. This method makes sure the `ParagraphFormat` object is unique to this Paragraph, cloning it if necessary.

```
protected void aboutToChangeParagraphFormat() {
    if (!paragraphFormatIsUnique) {
        try {
            format = (ParagraphFormat) format().clone();
        } catch (CloneNotSupportedException e) {}
        paragraphFormatIsUnique = true;
    }
}
```

Here’s a simple example of a method that modifies a `ParagraphFormat`:

```
void setFont(String fontName) {
    aboutToChangeParagraphFormat();
    format.privateSetFont(fontName);
}
```

❖ ❖ ❖

Known Uses

Many operating systems use COPY-ON-WRITE in their paging systems. Executable code is very rarely modified, so it’s usually SHARED between all processes using it, but this pattern allows modification when processes need it. By default each page out of an executable file is flagged as read-only and shared between all processes that use it. If a client writes to a shared page, the hardware generates an exception, and operating system exception handler then creates a writable copy for that process alone. [Kenah and Bate 1984; Goodheart and Cox 1994]

RogueWave’s Tools.h++ library uses COPY-ON-WRITE for its `CString` class [RogueWave 1994]. A `CString` object represents a dynamically allocated string. C++’s pass-by-value semantics mean that the `CString` objects are copied frequently, but very seldom modified. So each `CString` object is simply a wrapper referring to a shared implementation. `CString`’s copy constructor and related operators manipulate a reference count in the shared implementation. If any client does an operation to change the content of the string; the `CString` object simply makes a copy and does the operation on the copy. One interesting detail is that there is only one instance of the null string, which is always shared. All attempts to create a null string, for example by initialising a zero length string, simply access that shared object.

Because modifiable strings are relatively rare in programs, Sun Java implements them using a separate class, `StringBuffer`. However `StringBuffer` permits it’s clients to retrieve `String` objects with the method `toString`. To save memory and speed up performance the resulting `String` uses the underlying buffer already created by `StringBuffer`. However the `StringBuffer` object has a flag to indicate that the buffer is now shared; if a client attempts to make further changes to the buffer, `StringBuffer` creates a copy and uses that. [Chan et al 1998]

Objects in NewtonScript were defined using inheritance, so that common features could be declared in a parent object and then shared by all child objects that needed them. Default values for objects’ fields were defined using copy-on-write slots. If a child object didn’t define a field it would inherit that field’s value from its parent object, but when a child object wrote to a shared field a local copy of the field was automatically created in the child object. [Smith 1999].

See Also

HOOKS provide an alternative technique for changing the contents of read-only storage.

Embedded Pointer

How can you reduce the space used by a collection of objects?

- Linked data structures are built out of pointers to objects
- Collection objects (and their internal link objects) occupy large amounts of memory to store large collections.
- Traversing through a linked data structure can require temporary memory, especially if the traversal is recursive.

Object-Oriented programs implement relationships between objects by using collection objects that store pointers to other objects. Unfortunately, collection objects and the objects they use internally can require a large amount of memory. For example, the Strap-It-On's 'Mind Reader' brainwave analysis program must receive brainwave data in real-time from an interrupt routine, and store it in a list for later analysis. Because brainwaves have to be sampled many times every second, a large amount of data can accumulate before it can be analysed, even though each brainwave sample is relatively small (just a couple of integers). Simple collection implementations based on linked lists can impose an overhead of at least three pointers for every object they store, so storing a sequence of two-word samples in such a list more than doubles the sequence's intrinsic memory requirements – see figure xx below.

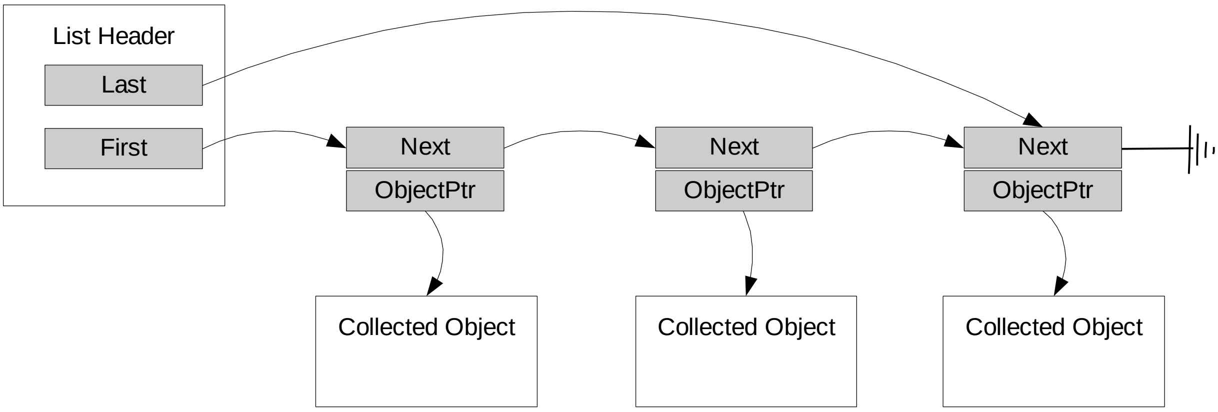


Figure 2: Linked list using external pointers

As well as this memory overhead, linked data structures have other disadvantages. They can use large numbers of small internal objects, increasing the possibility of fragmentation (see Chapter N). Allocating all these objects takes an unpredictable amount of time, making it unsuitable for real-time work. Traversing the structure requires following large numbers of pointer links; this also takes time, but more importantly, traversals of recursive structures like graphs and trees can also require an unbounded amount of temporary memory; in some cases, similar amounts of memory to that required to store the structure itself. Finally, any function that adds an object to such a collection may fail if there is insufficient memory, and so must carry all the costs of PARTIAL FAILURE.

Of course, linked structures have many compensating advantages. They can describe many different kinds of structures, including linked lists, trees, queues, all of a wide variety of different subtypes [Knuth 1997]. These structures can support a wide variety of operations quite efficiently (especially insertions and deletions in the middle of the data). Linked structures support VARIABLE ALLOCATION, so they never need to allocate memory that is subsequently unused. The only real alternative to building linked structures is to use some kind of FIXED ALLOCATION, such as a fixed-size array. But fixed structures place arbitrary limits on the number of objects in the collection, waste memory if they are not fully occupied, and insertion and deletion operations can be very expensive. So, how can you keep the benefits of linked data structures while minimising the disadvantages?

Therefore: *Embed the pointers maintaining the collection into each object.*

Design the collection data structure to store its pointers within the objects that are contained in the structure, rather than in internal link objects. You will need to change the definitions of the objects that are to be stored in the collection to include these pointers (and possibly to include other collection-related information as well).

You will also need to change the implementation of the collection object to use the pointers stored directly in objects. For a collection that is used by only one external client object, you can even dispense completely with the object that represents the collection, and incorporate its data and operations directly into the client. To traverse the data structure, use iteration rather than recursion to avoid allocating stack frames for every recursive call, and use extra (or reuse existing) pointer fields in the objects to store any state related to the traversal.

So, for example, rather than store the Brainwave sample objects in a collection, Strap-it-On's Brainwave Driver uses an embedded linked list. Each Brainwave sample object has an extra pointer field, called `Next`, that is used to link brainwave samples into a linked list. As each sample is received, the interrupt routine adjusts its `Next` field to link it into the list. The main analysis routine adjusts the sample object's pointers to remove each from the list in its own time for processing.

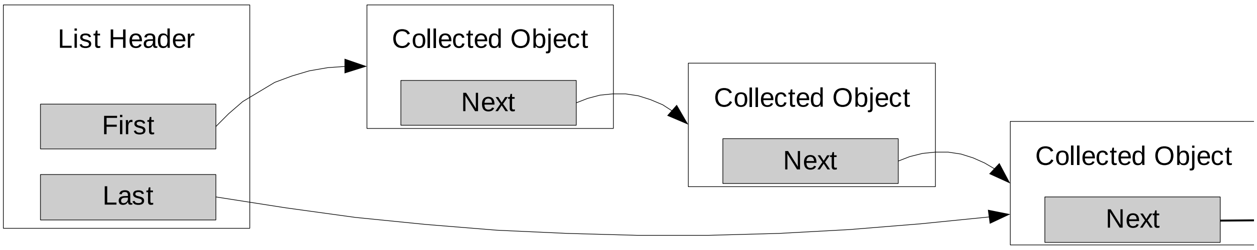


Figure 3: Linked list using embedded pointers

Consequences

Embedded pointers remove the need for internal link objects in collections, reducing the number of objects in the system and thus the system's *memory requirements*, while increasing the *predictability* of the systems memory use (especially if traversals are iterative rather than recursive). The routines to add and remove items from the linked list cannot suffer memory allocation failure.

Using embedded pointers reduces or removes the need for dynamic memory allocation, improving the *real-time performance* of the system. Some operations may have better *run-time performance*; for example with an embedded doubly-linked list you can remove an element in the collection simply by using a pointer to that element directly. With an implementation using external pointers (such as STL's Deque [Austern 1998]) you'd need first to set an iterator to refer to the right element, which requires a linear search.

However: Embedded pointers don't really belong to the objects they are embedded inside. This pattern reduces those objects' encapsulation, gaining a *local* benefit but reducing the *localisation* of the design. The pattern tightly couples objects to the container class that holds them, making it more difficult to reuse either class independently, increasing the *programmer effort* required because specialised collections often have to be written from scratch, reducing the *design quality* of the system and making the program harder to *maintain*.

In many cases a given collected object it will often need to be in several different collections at different times during its lifetime. It requires *programmer discipline* to ensure that the same pointer is never used by two collections simultaneously.

❖ ❖ ❖

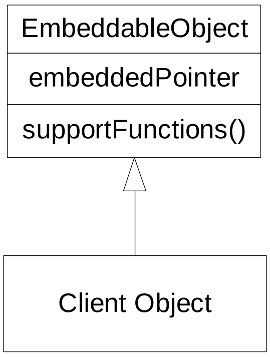
Implementation

Applying the Embedded Pointer pattern is straightforward: place pointer members into objects and build up linked data structures using those pointers, instead of using external collection objects. You can find the details in any decent textbook on data structures from Knuth [1997], which will describe the details, advantages, and disadvantages of the classical linked data structure designs, from simple singly and doubly linked lists to subtle complex balanced trees. See the SMALL DATA STRUCTURES pattern for a list of such textbooks.

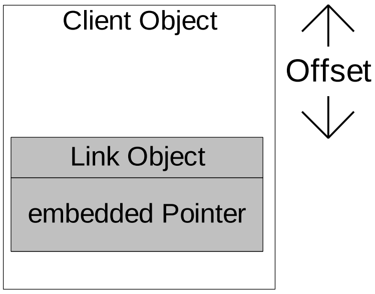
1. Reuse

The main practical issue when using embedded pointers is how to incorporate the pointers into objects in a way that provides some measure of reuse, to avoid re-implementing all the collection operations for every single list. The key idea is for objects to somehow present a consistent interface for accessing the embedded pointers to the collection class (or the functions that implement the collection operations). In this way, the collection can be used with any object that provides a compatible interface. There are three common techniques for establishing interfaces to embedded pointers: inheritance, inline objects, and preprocessor constructs.

1.1. Inheritance. You can put the pointers and accessing functionality into a superclass, and make the objects to be stored in a collection inherit from this class. This is straightforward, and provides a measure of reuse. However: you can't have more than one instance of such a pointer for a given object; if the pointer is implementing a collection, this would limit. In single-inheritance languages like Smalltalk it also prevents any other use of inheritance for the same object, and so limits any object to be in only one collection at a time. In languages with multiple inheritance objects could be in multiple collections provided each collection accesses the embedded pointers through a unique interface, supplied by a unique base class (C++) or interface (Java).



1.2. Inline Objects. In languages with inline objects, like C and C++, you can embedded a separate 'link' object that contains the pointers directly into the client object. This doesn't suffer from the disadvantages of using Inheritance, but you need to be able to find the client object from a given link object and vice versa. In C++ this can be implemented using pointers to members, or (more commonly) as an offset in bytes.



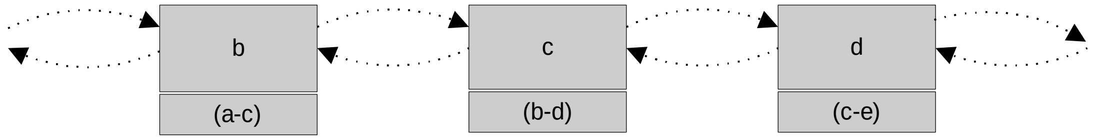
For example, EPOC's collection libraries find embedded pointers using byte offsets. Whenever a new collection is created, it must be initialised with the offset inside its client objects where its pointers are embedded.

1.3. Preprocessors. C++ provides two kinds of preprocessing: the standard preprocessor `cpp`, and the C++ template mechanisms. So in C++ a good approach is to include the embedded pointers as normal (possibly public) data members, and to reuse the management code via preprocessing. You can also preprocess code in almost any other language given a suitable preprocessor , which could be either a special purpose program like `m4`, or a general purpose program like `perl`.

2. Pointer Differences

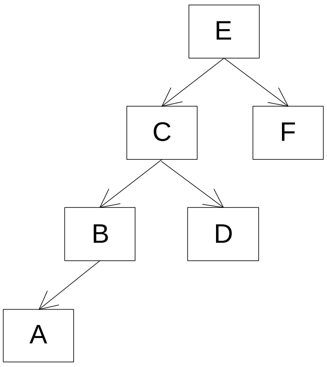
Sometimes an object needs to store two or more pointers; for example a circular doubly-linked list node needs pointers to the previous and next item in the list. You can reduce the amount of memory needed to store by storing the difference (or the bitwise exclusive or) of the two pointers, rather than the pointer itself. When you are traversing the structure forwards, for example, you take the address of the previous node and add the stored difference to find the address of the next node; reverse traversals work similarly.

For example, in Figure XXX, rather than node `c` storing the dotted forward and back pointers (i.e. the addresses of nodes `b` and `d`) node c stores only the difference between these two addresses. Given a pointer to node `b` and the difference stored within `c`, you can calculate the address of node d as (b – (d-c)). Similarly, traversing the list the other way, given the address of node d and (b-c) you can calculate the address of node b as (d+(b-d)). For this to work, you need to store two initial pointers, typically a head and tail pointer for a circular doubly-linked list [Knuth 1997].

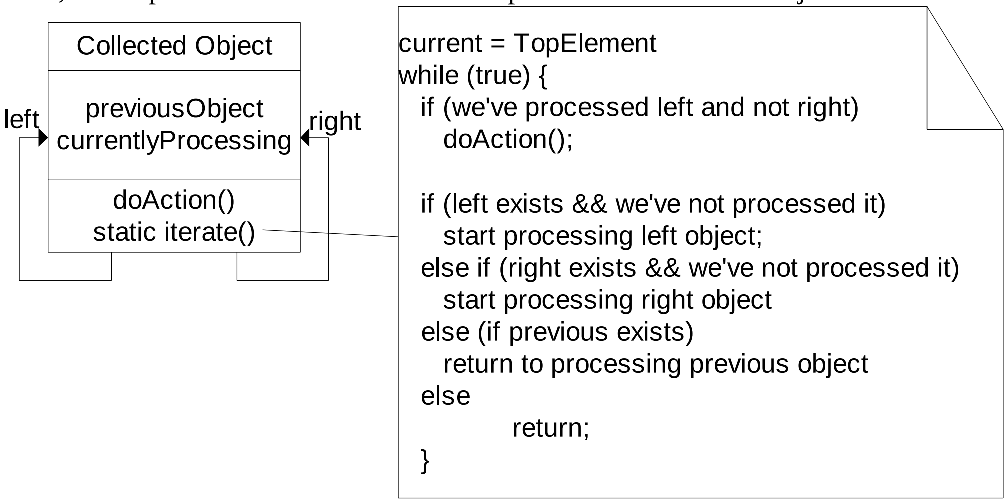


(3) Traversals

A related, but different, problem happens when you need to traverse an arbitrarily deep structure, especially if the traversal has to be recursive. Suppose for example you have an unbalanced binary tree, and you need to traverse through all the elements in order. A traversal beginning at E will recursively visit C, then F; the traversal at C will visit B and D, and so on. Every recursive call requires extra memory to store activation records on the stack, so traversing larger structures can easily exhaust a process's stack space.



3.1. Iterative traversals using extra embedded pointers. Consider the traversal more closely: at each object it needs to store one thing on the stack: the identity of the object its coming from (and possibly any working data or parameters passed through the iteration). So for example, when C invokes the operation on D, it must store that it needs to return to E on completion. You can use Embedded Pointers in each object to store this data (the parent, and the traversal state — two Boolean flags that remember whether the left and right leaves have been processed). This allows you to iterate over the structure using a loop rather than recursion, but imposes the overhead of an extra pointer or two in each object.

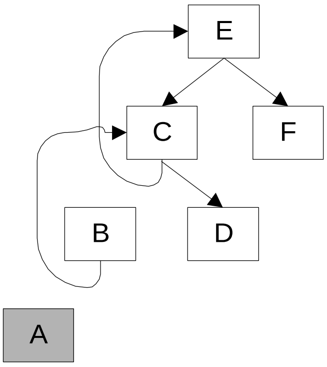


```
current = TopElement
while (true) {
  if (we've processed left and not right)
    doAction();

  if (left exists && we've not processed it)
    start processing left object;
  else if (right exists && we've not processed it)
    start processing right object
  else (if previous exists)
    return to processing previous object
  else
          return;
  }
```

3.2. Iterative traversals using pointer reversal. Consider the iteration process further. At any time one of the three pointers in each element, left leaf, right leaf or parent, is redundant. If there is no iteration, the parent pointer is redundant; if a left or right left is currently being processed that leaf pointer is redundant (because the traversal has already reached that leaf). Pointer reversal allows iterative traversals of linked structures by temporarily using pointers to leaf nodes as parent pointers: as the traversal proceeds around the object, the pointers currently being followed are 'reversed', that is, used to point to parent objects.

In the figure above, for example, when a traversal is at node A, node B's left leaf pointer would be reversed to point to its parent, node C; because B is C's left child, C's left child pointer would also be reversed to point to node E.



Example

Here’s an example using Embedded Pointers to store a data structure and traverse it using pointer reversal. The example program implements a sorting algorithm using a simple binary tree. To start with we’ll give the objects that are stored in the tree (the `BinaryTreeObjects`) a single character of local data. We also need a `greaterThan` operation and a operation called to do the action we need (`doIt`).

```
class BinaryTreeObject {
    char data;

    BinaryTreeObject(char data) {
        this.data = data;
    }

    Object doIt(Object param) {
        return ((String) param + data);
    }

    boolean greaterThan(BinaryTreeObject other) {
        return data > other.data;
    }
```

A binary tree needs a left pointer and a right pointer corresponding to each node. Using the Embedded Pointers pattern, we implement each within the structure itself:

```
BinaryTreeObject left;
BinaryTreeObject right;
```

Adding an element to the binary tree is fairly easy. We can traverse the tree starting at the top, going left when our new element is less than the current item, greater when it’s greater, until we get to a vacant position in the tree.

```
static void insert(BinaryTreeObject top, BinaryTreeObject newItem) {
        BinaryTreeObject current = top;

        for (;;) {
            if (current.greaterThan(newItem)) {
                if (current.left == null) {
                    current.left = newItem;
                    return;
                } else {
                    current = current.left;
                }
            } else {
                if (current.right == null) {
                    current.right = newItem;
                    return;
                } else {
                    current = current.right;
                }
            }
        }
    }
```

Note that this method is not recursive and so should allocate no memory other than one stack frame with one local variable (`current`).

Traversing the tree is more difficult, because the traversal has to visit all the elements in the tree, and this means backtracking up the tree when it reaches a bottom-level node. To traverse the tree without using recursion, we can add two embedded pointers to every tree node: a pointer to the previous (parent) item in the tree, and a marker noting which action, left node or right node, the algorithm is currently processing.

```
BinaryTreeObject previous;
    static final int Inactive = 0, GoingLeft = 1, GoingRight = 2;
    int action = Inactive;
```

The `traversal` method, then, must move through each node in infix order. Each iteration visits one node; however this may mean up to three visits to any given node (from parent going left, from left going right, and from right back to parent); we use the stored action data for the node to see which visit this one is. The `traversal` method must also call the `doIt` method at the correct point – after processing the left node, if any.

```
static Object traversal(BinaryTreeObject start, Object param) {
        BinaryTreeObject current = start;

        for (;;) {

            if (current.action == GoingLeft ||
                (current.action == Inactive && current.left == null)) {
                param = current.doIt(param);
            }

            if (current.action == Inactive && current.left != null) {
                current.action = GoingLeft;
                current.left.previous = current;
                current = current.left;
            } else if (current.action != GoingRight && current.right != null) {
       current.action = GoingRight;
       current.right.previous = current;
       current = current.right;
       }  else {
       current.action = Inactive;
       if (current.previous == null) {
           break;
       }
       current = current.previous;
       }

        }
        return param;
}
```

Of course, a practical implementation would improve this example in two ways. First we can put the `left`, `right`, `action`, `previous` pointers and the `greaterThan` stub into a base class (`SortableObject`, perhaps) or into a separate object. Second we can make the `traversal()` method into a separate ITERATOR object [Gamma et al 1995], avoiding the need to hard-code the `doIt` method.

We can extend this example further, to remove the parent pointer from the data structure using Pointer Reversal. First, we'll need two additional methods, to save the parent pointer in either the left or the right pointer:

```
BinaryTreeObject saveParentReturningLeaf(BinaryTreeObject parent) {
        BinaryTreeObject leaf;

        if (action == GoingLeft) {
            leaf = left;
            left = parent;
        } else {
            leaf = right;
            right = parent;
        }
        return leaf;
    }
```

and then to restore it as required:

```
BinaryTreeObject restoreLeafReturningParent(BinaryTreeObject leafJustDone) {
        BinaryTreeObject parent;

        if (action == GoingLeft) {
            parent = left;
            left = leafJustDone;
        } else {
            parent = right;
            right = leafJustDone;
        }
        return parent;
    }
```

Now we can rewrite the `traversal` method to remember the previous item processed, whether it's the parent of the current item or a leaf node, and to reverse the left and right pointers using the methods above:

```
static Object reversingTraversal(BinaryTreeObject top, Object param) {

        BinaryTreeObject current = top;
        BinaryTreeObject leafJustDone = null;
        BinaryTreeObject parentOfCurrent = null;

        for (;;) {

            if (current.action == GoingLeft ||
                (current.action == Inactive && current.left == null)) {
                param = current.doIt(param);
            }

            if (current.action != Inactive)
                parentOfCurrent = current.restoreLeafReturningParent(leafJustDone);

            if (current.action == Inactive && current.left != null) {
                current.action = GoingLeft;
                BinaryTreeObject p = current;
                current = current.saveParentReturningLeaf(parentOfCurrent);
                parentOfCurrent = p;
            } else if (current.action != GoingRight && current.right != null) {
       current.action = GoingRight;
       BinaryTreeObject p = current;
       current = current.saveParentReturningLeaf(parentOfCurrent);
       parentOfCurrent = p;
            } else {
                current.action = Inactive;
                if (parentOfCurrent == null) {
                    break;
                }
                leafJustDone = current;
                current = parentOfCurrent;
            }

        }
        return param;
    }
```

We're still wasting a word in each object for the 'action' parameter. In Java we could perhaps reduce this to a byte but no further. In a C++ implementation we could use the low bits of, say, the left pointer to store it (see PACKED DATA – packing pointers), thereby reducing the overhead of the traversing algorithm to nothing at all.

❖ ❖ ❖

Known Uses

EPOC provides at least three different 'linked list' collection classes using embedded pointers [Symbian 1999]. The embedded pointers are instances of provided classes (`TSglQueLink`, for example) accessed via offsets; the main collection logic are in separate classes, which use the 'thin template idiom' to provide type safety: `TSglQueue<MyClass>`. EPOC applications, and operating system components, use these classes extensively. The most common reason for preferring them over collections requiring heap memory is that operations using them cannot fail; this is a significant benefit in situations where failure handling is not provided.

The Smalltalk `LinkedList` class uses inheritance to mix in the pointers; the only things you can store into a `LinkedList` are objects that inherit from class Link [Goldberg and Robson 1983]. Class `Link` contains two fields and appropriate accessors (`previous` and `next`) to allow double linking. Compared with other Smalltalk collections, for each element you save one word of memory by using concatenation instead of pointers, plus you save the memory overhead of creating a new object (two words or so) and the overhead of doing the allocation.

See Also

You may be able to use FIXED ALLOCATION to embed objects directly into other objects, rather than just embedding pointers to objects.

Pointer reversal was first described by Peter Deutsch [Knuth 1997] and Schorr and Waite [1967]. Embedded Pointers and pointer reversal are used together in many implementations of GARBAGE COLLECTION [Goldberg and Robson 1983, Jones and Lins 1996]. Jiri Soukup discusses using preprocessors to implement linked data structures in much more detail [1994].

Multiple Representations

How can you support several different implementations of an object?

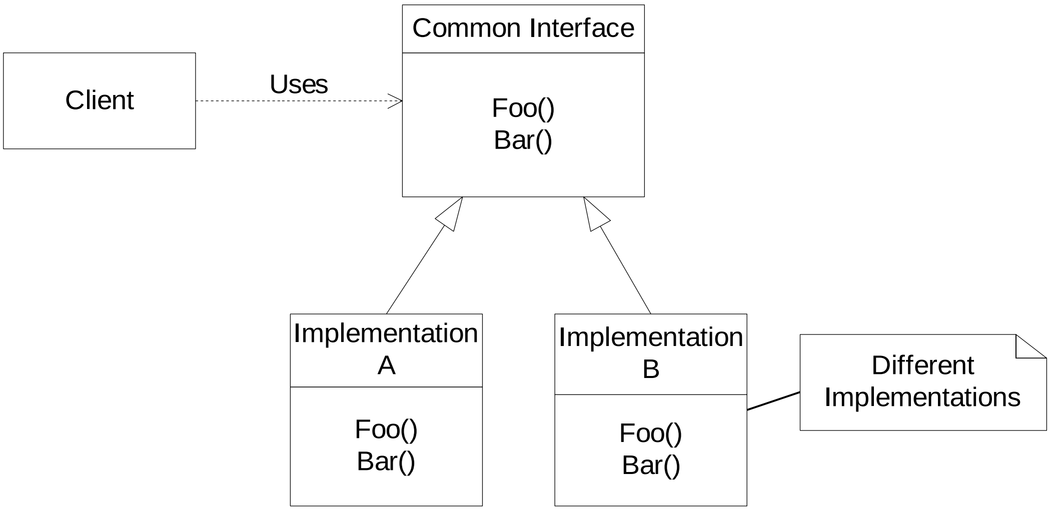
- There are several possible implementations of a class, with different trade-offs between size and behaviour.
- Different parts of your system, or different uses of the class, require different choices of implementation. One size doesn't fit all.
- There are enough instances of the class to justify extra code to reduce RAM usage.

Often when you design a class, you find there can be several suitable representations for its internal data structures. For example, in the Strap-It-On's word-processor (Word-O-Matic) a word may be represented as a series of characters, a bitmap, or a sequence of phonemes. Depending on the current output mechanism (a file, the screen, or the vocaliser) each of these representations might be appropriate.

Having to choose between several possible representations is quite common. Some representations may have small *memory requirements*, but be costly in processing time or other resources; others may be the opposite. In most cases you can examine the demands of the system and decide on a best SMALL DATA STRUCTURE. But what do you do when there's no single 'best' implementation?

Therefore: *Make each implementation satisfy a common interface.*

Design a common abstract interface that suits all the implementations without depending on a particular one, and ensure every implementation meets the interface. Access implementations via an ABSTRACT CLASS [Woolf 2000] or use ADAPTERS to access existing representations [Gamma et al 1995] so clients don't have to be aware of the underlying implementation.



For example, Word-O-Matic defines a single interface '`Word`', which is used by much of the word-processing code. Several concrete classes, `StorableWord`, `ViewableWord`, `SpokenWord`, that implement the `Word` interface. Each implementation has different internal data structures and different implementations of the operations that access those structures. The software creates whichever concrete class is appropriate for the current use of the Word object, but the distinction is only significant when it comes to outputting the object. The multiple implementations are concealed from most of the client code.

Consequences

The system will use the most appropriate implementation for any task, reducing the *memory requirements* and *processing time* overheads that would be imposed by using an inappropriate representation. Code using each instance will use the common interface and need not know the implementation, *reducing programmer effort* on the client side and increasing *design quality* and *reusability*.

Representations can be chosen *locally* for each data structure. More memory-intensive representations can be used when more memory is available, adding to the *scalability* of the system.

However: The pattern can also increase total *memory requirements*, since the code occupies additional memory.

MULTIPLE REPRESENTATIONS increases *programmer effort* in the implementation of the object concerned, because multiple implementations are more *complex* than a single implementation, although this kind of complexity is often seen as a sign of *high-quality* design because subsequent changes to the representation will be easier. For the same reason, it increases *testing costs* and *maintenance costs* overall, because each alternative implementation must be tested and maintained separately.

Changing between representations imposes a *space and time* overhead. It also means *more complexity* in the code, and *more complicated testing* strategies, increasing *programmer effort* and making memory use *harder to predict*.

❖ ❖ ❖

Implementation

There are number of issues to take into account when you are using MULTIPLE REPRESENTATIONS.

1. Implementing the Interface.

In Java the standard implementation of dynamic binding means defining either a Java class or a Java interface. Which is more suitable? From the point of view of the client, it doesn't matter; either can define an abstract interface. Using a Java interface gives you more flexibility, because each implementation may inherit from other existing classes as required; however, extending a common superclass allows several implementations to inherit common functionality. In C++ there's only the one conventional option for implementing the common interface: making all implementations inherit from a base class that defines the interface.

There's a danger that clients may accidentally rely on features on a particular implementation – particularly non-functional ones – rather than of the common interface. D'Souza and Wills [1998] discuss design techniques to avoid such dependencies in components.

2. Binding clients to implementations.

Sometimes you need to support several implementations, though a given client may only ever use one. For example, the C++ Standard Template Library (STL) iterator classes work on several STL collections, but any given STL iterator object works with only one [Stroustrup 1997, Austern 1998]. In this case, you can statically bind the client code to use only the one implementation — in C++ you could store objects directly and use non-virtual functions. If, however, a client needs to use several different object representations interchangeably then you need to use dynamic binding.

3. Creating dynamically bound implementations

The only place where you need to reference the true implementation classes in the code is where the objects are created. In many situations, it's reasonable to hard code the class names in the client, as in the following C++ example:

```
CommonInterface *anObject = new SpecificImplementation( parameters );
```

If there's a good reason to hide even this mention of the classes from the client, then the ABSTRACT FACTORY pattern [Gamma et al 1995] can implement a 'virtual constructor' [Coplien 1994] so that the client can specify which object to create using just a parameter.

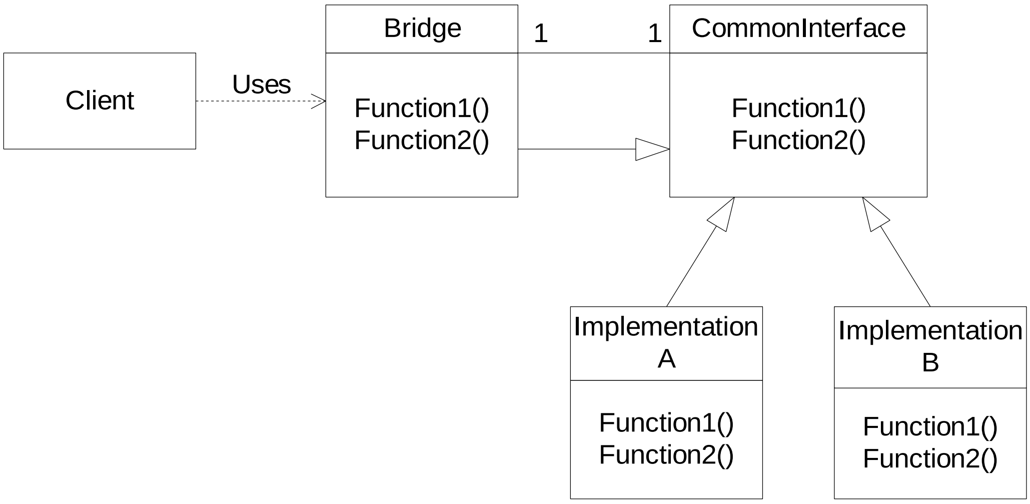
4. Changing between representations

In some cases, an object's representation needs to change during its lifetime, usually because a client needs some behaviour that is not supported well by the object's current representation. Changes to an object's representation can be explicitly requested by its client, or can be triggered automatically within the object itself. Changing representations automatically has several benefits: the client doesn't need knowledge of the internal implementation, improving *encapsulation*, and you can tune the memory use entirely within the implementation of the specific object, improving *localisation*. Changing representations automatically requires dynamic binding, so clients will use the correct representation without being aware of it. In some situations, however, the client can have a better knowledge of optimisation strategies than is available to the object itself, typically because the client is in a better position to know which operations will be required.

4.1. Changing representations explicitly. It is straightforward for an object to let a client change its representation explicitly: the object should implement a conversion function (or a C++ constructor) that takes the common interface as parameter, and returns the new representation.

```
class SpecificImplementation : public CommonInterface
{ public:
    SpecificImplementation( CommonInterface c ) {
    // initialise this from c
     }
};
```

4.2. Changing representations automatically. You can use the BRIDGE pattern to keep the interface and identity of the object constant when its internal structure changes (strictly speaking a 'half bridge', since it varies only the object implementation and not the abstraction it supports) [Gamma et al 1995]. The client sees only the bridge object, which delegates all its operations to an implementation object through a common interface:



The bridge class needs the same methods as the common interface; so it's reasonable (though unnecessary) in C++ and Java to make the Bridge class derive from the common interface. Each implementation object will need a construction function taking the common interface as parameter. Of course, some implementations may store more or less data, so there may be special cases with more specific constructors.

Some languages make it simple to implement the Bridge object itself. In Smalltalk, for example, you can override the `DoesNotUnderstand:` method to pass any unrecognised operation on to the implementation object [Lalonde 1994]. In C++ you can implement `operator->()` to do the same [Coplien 1994], or alternatively you can avoid deriving the Bridge class from the common interface and by make all its functions non-virtual and inline.

Example

This Java example implements a Word object with two representations: as a simple string (the default), and as a string with an additional cached corresponding sound. Both these representations implement the basic Word interface, which can return either a string or sound value, and allows clients to choose the most appropriate representation.

```
interface WordInterface
{
    public byte[] asSound();
    public String asString();
    public void becomeSound();
    public void becomeString();
}
```

The most important concrete class is the Word class, that acts as a bridge between the Word abstraction an its two representations, as a sound and as a text string.

```
class Word implements WordInterface {
    private WordInterface rep;

    public byte[] asSound()     {return rep.asSound();}
    public String asString()    {return rep.asString();}
    public void becomeSound()   {rep.becomeSound();}
    public void becomeString()  {rep.becomeString();}
```

The constructor of the `Word` class must select an implementation. It uses the method `Become`, which simply sets the implementation object.

```
public Word(String word) {
        become(new StringWordImplementation(this, word));
    }
    public void become(WordInterface rep) {
        this.rep = rep;
    }
```

The default implementation stores `Words` as a text string. It also keeps a pointer to its Word BRIDGE object, and uses this pointer to automatically change a word's representation into the other format. It has two constructors: one, taking a string, is used by the constructor for the `Word` object; the other, taking a `WordInterface`, is used to create itself from a different representation.

```
class StringWordImplementation implements WordInterface
{
    private String word;
    private Word bridge;

    public StringWordImplementation(Word bridge, String word) {
        this.bridge = bridge;
        this.word = word;
    }

    public StringWordImplementation(Word bridge, WordInterface rep) {
        this.bridge = bridge;
        this.word = rep.asString();
    }
```

It must also provide implementations of all the `WordInterface` methods. Note how it must change its representation to return itself as a sound; once the `asSound` method returns this object will be garbage:

```
public byte[] asSound()
    {
        becomeSound();
        return bridge.asSound();
    }
    public String asString() {return word;}

    public void becomeSound() {
        bridge.become(new SoundWordImplementation(bridge, this));
    }
    public void becomeString() {}
```

Finally, the sound word class is similar to the text version, but also caches the sound representation. Implementing the sound conversion function is left as an exercise for the reader!

```
class SoundWordImplementation implements WordInterface
{
    private String word;
    private Word bridge;
    private byte[] sound;

    SoundWordImplementation(Word bridge, WordInterface rep) {
        this.bridge = bridge;
        this.word = rep.asString();
        this.sound = privateConvertStringToSound(this.word);
    }

    public String asString() {return word;}
    public byte[] asSound()  {return sound;}
    public void becomeString() {
        bridge.become(new StringWordImplementation(bridge, this));
    }
    public void becomeSound() {}
}
```

❖ ❖ ❖

Known Uses

Symbian's EPOC C++ environment handles strings as *Descriptors* containing a buffer and a length. Descriptors provide many different representations of strings: in ROM, in a fixed-length buffer, in a variable length buffer and as a portion of another string. Each kind of descriptor has its own class, and users of the strings see only two base classes: one for a read-only string, the other for a writable string [Symbian 1999].

The Psion 5's Word Editor has two internal representations of a document. When the document is small the editor keeps formatting information for the entire document; when the document is larger than a certain arbitrary size, the editor switches to storing information for only the part of the document currently on display. The switch is handled internally to the Editor's 'Text View' component; clients of the component (including other applications that need rich text) are unaware of the change in representation.

Smalltalk's collection classes also use this pattern: all satisfy the same protocol, so a user need not be aware of the particular implementation used for a given collection [Goldberg and Robson 1983]. Java's standard collection classes have a similar design [Chan et al 1998]. C++'s STL collections also use this pattern: STL defines the shared interface using template classes; all the collection classes support the same access functions and iterator operations [Stroustrup 1997, Austern 1998].

Rolfe&Nolan's Lighthouse system has a 'Deal' class with two implementations: by default an instance contains only basic data required for simple calculations; on demand, it extends itself reading the entire deal information from its database. Since clients are aware when they are doing more complex calculations, the change is explicit, implemented as a `FattenDeal` method on the object.

MULTIPLE REPRESENTATIONS can also be useful to implement other memory saving patterns. For example the LOOM Virtual Memory system for Smalltalk uses two different representations for objects: one for objects completely in memory, and a second for objects PAGED out to SECONDARY STORAGE [Kaehler and Krasner 1983]. Format Software's PLUS application implements CAPTAIN OATES for images using three representations, which change dynamically: a bitmap ready to `bitblt` to the screen, a compressed bitmap, and a reference to a representation in the database.

See Also

The BRIDGE pattern describes how abstractions and implementations can vary independently [Gamma et al 1994]. The MULTIPLE REPRESENTATIONS pattern typically uses only half of the BRIDGE pattern, because implementations can vary (to give the multiple representations) but the abstraction remains the same.

Various different representations can use explicit PACKED DATA or COMPRESSION, be stored in SECONDARY STORAGE, be READ-ONLY, or be SHARED. They may also use FIXED ALLOCATION or VARIABLE ALLOCATION.

Memory Allocation

Version 06/06/00 09:25 - 2

How do you allocate memory to store your data structures?

- You're developing object-oriented software for a memory-constrained system.
- You've designed suitable data structures for each of your objects.
- You need to store these data structures in main memory
- You need to recycle this memory once the objects are no longer required.
- Different classes – and different instances of a single class – have different allocation requirements.

When a system begins running, it sees only virgin memory space. A running program, particularly an object-oriented one, uses this memory as 'objects' or data structures, each occupying a unique and differently sized area of memory. These objects will change with time: some remain indefinitely; others last varying lengths time; some are extremely transient. Computing environments need *allocation* mechanisms to call these structures into being from the primordial soup of system memory.

For example the Strap-It-On PC uses objects in many different ways. User Interface objects must be available quickly, with no awkward pauses. Transient objects must appear and disappear with minimum overhead. Objects in its major calculation engines must be provided and deallocated with minimum programmer effort. Objects in its real-time device drivers must be available within a fixed maximum time. Yet processing power is limited so, for example, an allocation technique that minimises programmer effort can't possibly satisfy the real-time constraints. No single allocation approach suits all of these requirements.

At first glance, a particular environment may not appear to provide much of a choice, especially as many object-oriented languages, including Smalltalk and Java, allocate all objects dynamically [Goldberg and Robson 1983; Gosling et al 1996; Egremont 1999]. But in practice even these languages support a good deal of variation. Objects can exist for a long or short time (allowing run-time compiler optimisations); you can reuse old objects rather than creating new ones; or you can create all the objects you need at the start of the program. More low-level languages like C and C++ support even more possibilities. So what strategy should you use to store your objects?

Therefore: *Choose the simplest allocation technique that meets your need.*

Analyse each time you allocate an object decide which technique is most suitable for allocating that object. Generally, you should choose the simplest allocation technique that will meet your needs, to avoid unnecessarily complicating the program, and also to avoid unnecessary work. The four main techniques for allocating objects that we discuss in this chapter are (in order from the simplest to the most complex):

FIXED ALLOCATION	Pre-allocating objects as the system starts running
MEMORY DISCARD	Allocating transient objects in groups, often on the stack.
VARIABLE ALLOCATION	Allocating objects dynamically as necessary from a heap.
POOLED ALLOCATION	Allocating objects dynamically from pre-allocated memory space.

The actual complexity of these patterns does depend on the programming language you are using: in particular, in C or C++ **MEMORY DISCARD** is easier to use than **VARIABLE ALLOCATION**, while languages like Smalltalk and Java assume **VARIABLE ALLOCATION** as the default.

What goes up must come down; what is allocated must be deallocated. If you use any of the dynamic patterns (**VARIABLE ALLOCATION**, **MEMORY DISCARD** or **POOLED ALLOCATION**) you'll also need to consider how the memory occupied by objects can be returned to the system when the objects are no longer needed. In this chapter we present three further patterns that deal with deallocation: **COMPACTION** ensures the memory once occupied by deallocated objects can be recycled efficiently, and **REFERENCE COUNTING** and **GARBAGE COLLECTION** determine when shared objects can be deallocated.

Consequences

Choosing an appropriate allocation strategy can ensure that the program meets its *memory requirements*, and that its runtime demands for memory are *predictable*. Fixed allocation strategies can increase a programs *real-time responsiveness* and *time performance*, while variable strategies can ensure the program can *scale up* to take advantage of more memory if it becomes available, and avoid allocating memory that is unused.

However: Supporting more than one allocation strategy requires *programmer effort* to implement. The system developers must consider the allocation strategies carefully, which takes significantly more work than just using the default allocation technique supported by the programming language. This approach also requires *programmer discipline* since developers must ensure that they do use suitable allocation strategies. Allocating large amounts of memory as a system beings executing can increase its *start-up time,* while relying on dynamic allocation can make memory use *hard to predict* in advance.

❖ ❖ ❖

Implementation

As with all patterns, the patterns in this chapter can be applied together, often with one pattern relying on another as part of its implementation. The patterns in this chapter can be applied in a particularly wide variety of permutations. For example, you could have very large object allocated on the heap (**VARIABLE ALLOCATION**), which contains an embedded array of sub-objects (**FIXED ALLOCATION**) that are allocated internally by the large containing object (**POOLED ALLOCATION**). Here, the **FIXED ALLOCATION** and **POOLED ALLOCATION** patterns are implemented within the large object, and each pattern is supported by other patterns in their implementation.

You can use different patterns for different instances of the same class. For example, different instances of the Integer class could be allocated on the heap (**VARIABLE ALLOCATION**), on the stack (**MEMORY DISCARD**), or embedded in another object (**FIXED ALLOCATION**), depending on the requirements of each particular use.

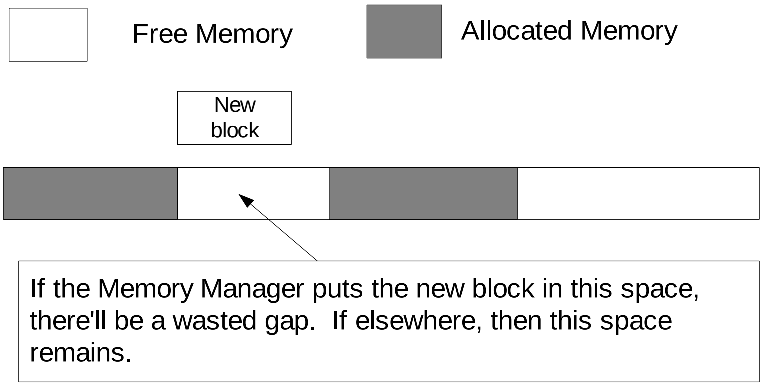
You can also choose between different patterns depending on circumstance. For example, a Smalltalk networking application was required to support a guaranteed minimum throughput, but could improve its performance if it could allocate extra buffer memory. The final design pre-allocated a pool of five buffers (**FIXED ALLOCATION**); if a new work item arrived while all the buffers were in use, and more memory was available, the system dynamically allocated further buffers (**VARIABLE ALLOCATION**).

Here are some further issues to consider when designing memory allocation:

1. Fragmentation

Fragmentation is a significant problem with dynamic memory allocation. There are two kinds of fragmentation: *internal fragmentation*, when a data structure does not use all the memory it has been allocated; and *external fragmentation*, when memory lying between two allocated structures cannot be used, generally because it is too small to store anything else. For example, if you delete an object that occupies the space between two other objects, some of the deleted object's space will be wasted, unless

- the other objects are also deleted, giving a single contiguous memory space;
- you are able to move objects around in memory, to squeeze the unused space out between the objects; or,
- you are lucky enough to allocate another object that fills the unused space exactly.



If the Memory Manager puts the new block in this space, there'll be a wasted gap. If elsewhere, then this space remains.

Fragmentation is difficult to resolve because patterns which reduce internal fragmentation (say by allocating just the right amount of memory) typically increase external fragmentation because space is wasted between all the oddly sized allocated blocks of memory. Similarly, patterns which reduce external fragmentation (by allocating equally-sized blocks of memory) increase internal fragmentation because some memory will be wasted within each block.

2. Memory Exhaustion

No matter what allocation strategy you choose, you can never have enough memory to meet all eventualities: you may not pre-allocate enough objects using **FIXED ALLOCATION**; or a request for a **VARIABLE ALLOCATION** from heap or stack memory can fail; or the memory pools for **POOLED ALLOCATION** can be empty. Sooner or later you will run out of memory. When planning your memory allocation, you also need to consider how you will handle memory exhaustion.

2.1. Fixed Size Client Memories. You can expose a fixed-size memory model directly to your users or client components. For example, many pocket calculators make users choose one of ten memories in which to save a value, with no suggestion that the system could have more memory; many components support up to a fixed number of objects in their interfaces (connections, tasks, operations or whatever) and generate an error if this number is exceeded. This approach is easy to program, but it decreases the usability of the system, because it makes users, or client components, take full responsibility for dealing with memory exhaustion.

2.2. Signal an error. You can signal a memory exhaustion error to the client. This approach also makes clients responsible for handling the failure, but typically leaves them with more options than if you provided a fixed number of user memories. For example, if a graphics editor program does not have enough memory to handle a large image, users may prefer to shut down other applications to release more memory in the system as a whole.

Signalling errors is more problematic internally, when one component sends an error to another. Although it is quite simple to notify client components of memory errors, typically by using exceptions or return codes, programming client components to handle errors correctly is much more difficult (see the **PARTIAL FAILURE** pattern).

2.3. Reduce quality. You can reduce the quantity of memory you need to allocate by reducing the quality of the data you need to store. For example, you can truncate strings and reduce the sampling frequency of sounds and images. Reducing quality can maintain system throughput, but is not applicable if it discards data that is important to users. Using smaller images, for example, may be fine in a network monitoring application, but not in a graphics manipulation program.

2.4. Delete old objects. You can delete old or unimportant objects to release memory for new or important objects. For example telephone exchanges can run out of memory when creating a new connection, but they can regain memory by terminating the connection that's been ringing for longest, because it's least likely to be answered (**FRESH WORK BEFORE STALE, [**Meszaros 1998]). Similarly, many message logs keep from overflowing by storing only a set amount of messages and deleting older messages as new messages arrive.

2.5. Defer new requests. You can delay allocation requests (and the processing that depends on them) until sufficient memory is available. The simplest and most common approach for this is for the system not to accept more input until the current tasks have completed. For example many MS Windows applications change the pointer to a 'please wait' ikon, typically an hourglass, meaning that the user can't do anything else until this operation is complete. And many communications systems have 'flow control' mechanisms to stop further input until the current input has been handled. Even simpler is batch-style processing, reading elements sequentially from a file or database and only reading the next when you've processed the previous one. More complicated approaches require concurrency in the system so that one task can block on or queue requests being processed by another. Many environments support synchronisation primitives like semaphores, or higher-level pipes or shared queues that can block their clients automatically when they cannot fulfil a request. In single-threaded systems component interfaces can support callbacks or polling to notify their clients that they have completed processing a request. Doug Lea's book *Concurrent Programming in Java* [2000] discusses this in more detail, and the techniques and designs he describes are applicable to most object-oriented languages, not just Java.

2.6. Ignore the problem. You can completely ignore the problem, and allow the program to malfunction. This strategy is, unfortunately, the default in many environments, especially where paged virtual memory is taken for granted. For example, the Internet worm propagated through a bug in the UNIX `finger` demon where long messages could overwrite a fixed-sized buffer [Page 1988]. This approach is trivial to implement, but can have extremely serious consequences: the worm that exploited the `finger` bug disabled much of the Internet for several days.

A more predictable version of this approach is to detect the problem and immediately to halt the processing. While this will avoid any the program running amuck through errors in memory use, it does not contribute to system stability or reliability in the long term.

❖ ❖ ❖

Specialised Patterns

The following chapter explores seven patterns of memory allocation:

Fixed Allocation ensures you'll always have enough memory by pre-allocating structures to handle your needs, and by avoiding dynamic memory allocation during normal processing.

Variable Allocation avoids unused empty memory space by using dynamic allocation to take and return memory to a heap.

Memory Discard simplifies de-allocating temporary objects by putting them in a temporary workspace and discarding the whole workspace at once.

Pooled Allocation avoids the overhead of variable allocation given a large number of similar objects, by pre-allocating them as required and maintaining a 'free list' of objects to be reused.

Compaction avoids memory fragmentation by moving allocated objects in memory to remove the fragmentation spaces.

Reference Counting manages shared objects by keeping a count of the references to each shared object, and deleting each object when its reference count is zero.

Garbage Collection manages shared objects by periodically identifying unreferenced objects and deleting them.

The following diagram shows the relationships between the patterns.

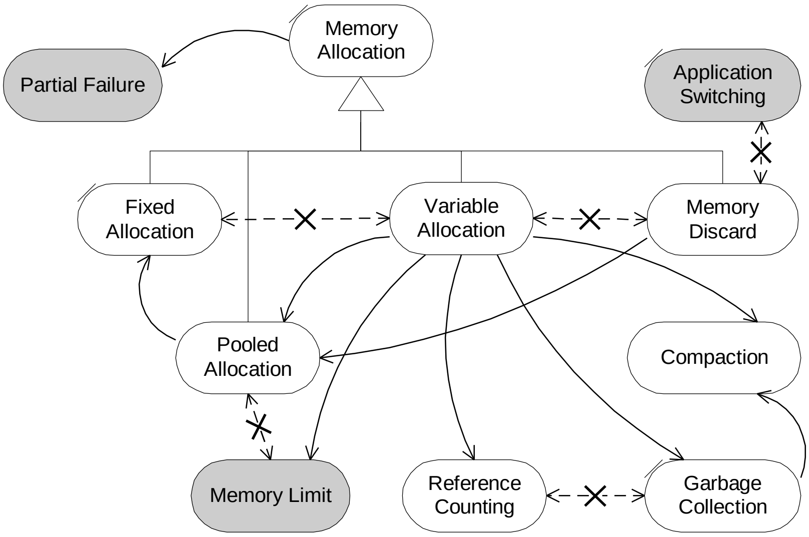


Figure 1: Allocation Pattern Relationships

Possibly the key aspect in choosing a pattern is deciding which is more important: minimising memory size or making memory use predictable. **Fixed Allocation** will make memory use predictable, but generally leaves some memory unused, while **Variable Allocation** can make better use of memory but provides less predictability.

See Also

The **Small Data Structures** chapter explores patterns to determine the structure of the allocated objects. **Compression** provides an alternative means to reduce the space occupied by RAM-resident objects.

The **Small Architecture** patterns describe how to plan memory use for the system as a whole, how memory should be communicated between components, and how to deal with memory exhaustion using **Partial Failure**.

Kent Beck’s *Smalltalk Best Practice Patterns* contain several patterns that describe how variables should be chosen to minimise object’s lifetimes [Beck 1997].

Fixed Allocation

Also known As: Static Allocation, Pre-allocation

How can you ensure you will never run out of memory?

- You can't risk running out of memory.
- You need to predict the amount of memory your system will use exactly.
- You need to allocate memory quickly, or within a given time.
- Allocating memory from the heap has unacceptable overheads

Many applications cannot risk running out of memory. For example the Strap-It-On's Muon-based "ET-Speak" communication system must be prepared to accept short messages from extra-terrestrials at any time; running out of memory while receiving a message could be a major loss to science. Many other systems have absolute limits to the memory available, and have no acceptable means for handling out-of-memory situations. For example, what can an anaesthetist do about a message from a patient monitor that has run out of internal memory? When the users have no effective control over memory use, running out of memory can become a truly fatal program defect.

The usual object-oriented approach is to allocate objects dynamically on the heap whenever there's a need for them [Ingalls 1981]. Indeed many OO languages, including Smalltalk and Java, allocate all objects from the heap. Using dynamic allocation, however, always risks running out of memory. If every component allocates memory at arbitrary times, how can you be certain that memory will never run out?

It's certainly possible to estimate a program's memory use, when designing a **SMALL ARCHITECTURE**, but how can you be sure the estimates accurately reflect the behaviour of the finished system? Similarly, you can test the system with arbitrary combinations of data, but "*testing can be used to show the presence of bugs, but never to show their absence*" [Dijkstra 1972] so how can you be sure you've found the most pathological case?

Dynamic memory allocation has other problems. Some allocation algorithms can take unpredictable amounts of time, making them unsuitable for real-time systems. Virtually all allocation algorithms need a few extra bytes with each item to store the block size and related information. Memory can become fragmented as variable sized memory chunks come and go, wasting further memory, and to avoid memory leaks you must be careful to deallocate every unused object.

Therefore: *Pre-allocate objects during initialisation.*

Allocate fixed amounts of memory to store all the objects and data structures you will need before the start of processing. Forbid dynamic memory allocation during the execution of the program. Implement objects using fixed-sized data structures like arrays or pre-allocated collection classes.

Design your objects so that you can assign them to new uses without having to invoke their constructors – the normal approach is to write separate initialisation functions and dummy constructors. Alternatively (in C++) keep the allocated memory unused until it's needed and construct objects in this memory.

As always you shouldn't 'hard code' the numbers of objects allocated [Plum and Saks 1991] – even though this will be fixed for any given program run. Use named constants in code or system parameters in the runtime system so that the numbers can be adjusted when necessary.

So, for example, the ET-Speak specification team has agreed that it would be reasonable to store only the last few messages received and to set a limit to the total storage it can use. The ET-Speak programmers allocated a fixed buffer for this storage, and made new incoming messages overwrite the oldest messages.

Consequences

FIXED ALLOCATION means you can *predict the system's memory use* exactly: you can tell how much memory your program will need at compile time. The *time required* for any memory allocation operation is constant and small. These two features make this pattern particularly suitable for *real-time* applications.

In addition, fixed allocation minimises *space overhead* for using pointers, and *global overhead* for a garbage collector. Using **FIXED ALLOCATION** *reduces programmer effort* when there's no need to check for allocation failure. It makes programs *easier to test* (they either have enough memory or they don't) and often makes programs more reliable as there is less to go wrong.

Fixed memory tends to be allocated at the start of the process and never deallocated, so there will be little *external fragmentation.*

However: The largest liability of **FIXED ALLOCATION** is that to handle expected worst case loads, you have to allocate more memory than necessary for average loads. This will increase the program's *memory requirements*, as much of the memory is unused due to *internal fragmentation,* particularly in systems with many concurrent applications.

To use **FIXED ALLOCATION,** you have to find ways to limit or defer demands on memory. Often you will have to limit throughput, so that you never begin a new task until previous tasks have been completed; and to limit capacity, imposing a fixed maximum size or number of items that your program will store. Both these reduce the program's *usability.*

Pre-allocating the memory can increase the system's *start-up time* – particularly with programming languages that don't support static data.

In many cases **FIXED ALLOCATION** can increase *programmer effort*; the programmer is forced to write code to deal with the fixed size limit, and should at least think how to handle the problem. It can also make it harder to take advantage of more memory should it become available, reducing the program's *scalability.*

Nowadays programs that use fixed size structures are sometimes seen as lower-quality designs, although this probably says more about fashion than function!

❖ ❖ ❖

Implementation

This pattern is straightforward to implement:

- Design objects that can be pre-allocated.
- Allocate all the objects you need a the start of the program.
- Use only the pre-allocated objects, and ensure you don't ever need (or create) more objects than you've allocated.

1. Designing Fixed Allocation Objects

Objects used for fixed allocation pattern may have to be designed specifically for the purpose. With fixed allocation, memory is allocated (and constructors invoked) only at the very start of the system; any destructor will be invoked only when the system terminates (if then). Rather than using constructors and destructors normally, you'll need to provide extra pseudo-constructors and pseudo-destructors that configure or release the pre-allocated objects to suit their use in the program.

```
class FixedAllocationThing {
public:
    FixedAllocationThing() {}

    Construct() { . . . }
    Destruct()  { . . . }
};
```

By writing appropriate constructors you can also design classes so that only one (or a certain number) of instances can be allocated, as described by the **SINGLETON** pattern [Gamma et al 1995].

2. Pre-allocating Objects

Objects using **FIXED ALLOCATION** need to be pre-allocated at the start of the program. Some languages (C++, COBOL, FORTRAN etc.) support fixed allocation directly, so that you can specify the structure statically at compile time so the memory will be set up correctly as the program loads. For example, the following defines some buffers in C++:

```
struct Buffer { char data[1000]; };
static Buffer AllBuffers[N_BUFFERS_REQUIRED];
```

Smalltalk permits more sophisticated **FIXED ALLOCATION**, allowing you to include arbitrarily complicated allocated object structures into the persistent Smalltalk image loaded whenever an application starts.

In most other OO languages, you can implement **FIXED ALLOCATION** by allocating all the objects you need at the start of the program, calling `new` as normal. Once objects are allocated you should never call `new` again, but use the pre-existing objects instead, and call their pseudo-constructors to initialise them as necessary. To support this pattern, you can design objects so that their constructors (or calls to `new`) signal errors if they are called once the pre-allocation phase has finished.

Pre-allocating objects from the system heap raises the possibility that even this initial allocation could fail due to insufficient memory? In this situation there are two reasonable strategies you can adopt. Either you can regard this as a fatal error and terminate the program; at this point termination is usually a safe option, as the program as not yet started running and it's unlikely to do much damage. Alternatively you can write the code so that the program can continue in the reduced space, as described in the **PARTIAL FAILURE** pattern.

3. Library Classes

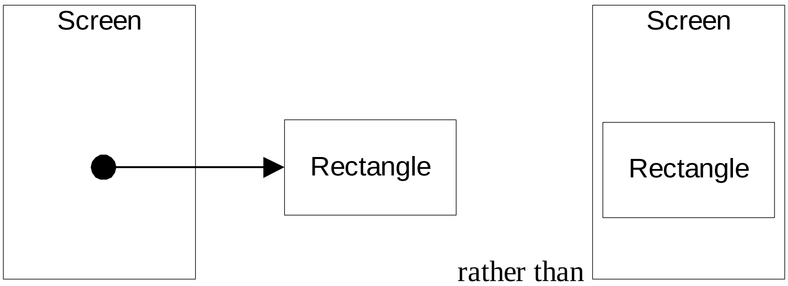
It's straightforward to avoid allocation for classes you've written: just avoid heap operations such as `new` and `delete`. It's more difficult to avoid allocation in library classes, unless the libraries have been designed so that you can override their normal allocation strategies. For example, a Java or C++ dynamic vector class will allocate memory whenever it has insufficient capacity to store an element inserted into it.

Most collection classes separate their size (the number of elements the currently contain) from their capacity (the number of elements they have allocated space for); a collection's size must be less than or equal to its capacity. To avoid extra allocation, you can pre-allocate containers with sufficient capacity to meet their needs (see **HYPOTH-A-SIZED COLLECTION** [Auer and Beck 1996]). For example, the C++ Standard Template Library precisely defines the circumstances when containers will allocate memory, also allows you to customise this allocation [Austern 1998].

Alternatively, you can use **EMBEDDED POINTERS** to implement relationships between objects, thus removing the need for library classes to allocate memory beyond your control.

4. Embedded Objects

A very common form of fixed allocation in object-oriented programs is *object embedding* or *inlining*: one object is allocated directly within another object. For example, a Screen object owning a Rectangle object might embed the Rectangle in its own data structure rather than having a pointer to a separate object.



C++ and Eiffel support inlining directly [Stroupstrup 1997; Meyer 1992]. In other languages you can inline objects manually by refactoring your program, moving the fields and methods from the internal object into the main object and rewriting method bodies as necessary to maintain correctness [Fowler 1999].

Embedding objects removes the time and space overheads required by heap allocation: the main object and the embedded object are just one object as far as the runtime system is concerned. The embedded object no longer exists as a separate entity, however, so you cannot change or replace the embedded object and you cannot use subtype polymorphism (virtual function calls or message sends).

In languages, such as Java and Smalltalk, that do not support embedded intrinsically you won't be able to refer to the embedded object or pass it as an argument to other objects in the system. For example, you might implement a Rectangle using two point objects as follows:

```
class Point {
   private int x;
   private int y;
   // methods omitted
}
```

```
class Rectangle {
   private Point topLeft;
   private Point bottomRight;
   // etc.
}
```

But you could avoid the need to allocate the two Point objects by making the two point objects inline, at the cost of not being able to use the Point objects directly:

```
class InlinedRectangle {
   private int xTopLeft;
   private int yTopLeft;
   private int xBottomRight;
   private int yBottomRight;
   // ...
}
```

4. Deferring Commitment to Fixed Allocation

Sometimes you need to make the decision to use fixed allocation later in the project, either because you are unsure it will be worth the effort, or because you used variable allocation but discovered memory problems during performance testing. In that case you can use **POOLED ALLOCATION** to give a similar interface to variable allocation but from a pre-allocated, fixed-size pool.

Example

This example implements a message store as a fixed-size data structure, similar to the circular buffer used in the ET-speak application. The message store stores a fixed number of fixed size messages, overwriting old messages as new messages arrive.

Figure xxx below shows an example of the message store, handling the text of Hamlet's most famous soliloquy:

```
To be, or not to be, that is the question.
Whether 'tis nobler in the mind to suffer the slings and arrows of outrageous
fortune.
Or to take arms against a sea of troubles and by opposing end them.
To die, to sleep - no more;
And by a sleep to say we end the heart-ache and the thousand natural shocks that
flesh is heir to.
Tis a consummation devoutly to be wished.
```

The figure shows the store just as the most recent message (“`tis a consummation`…” is just about to overwrite the oldest (“`To be, or not to be`…”).

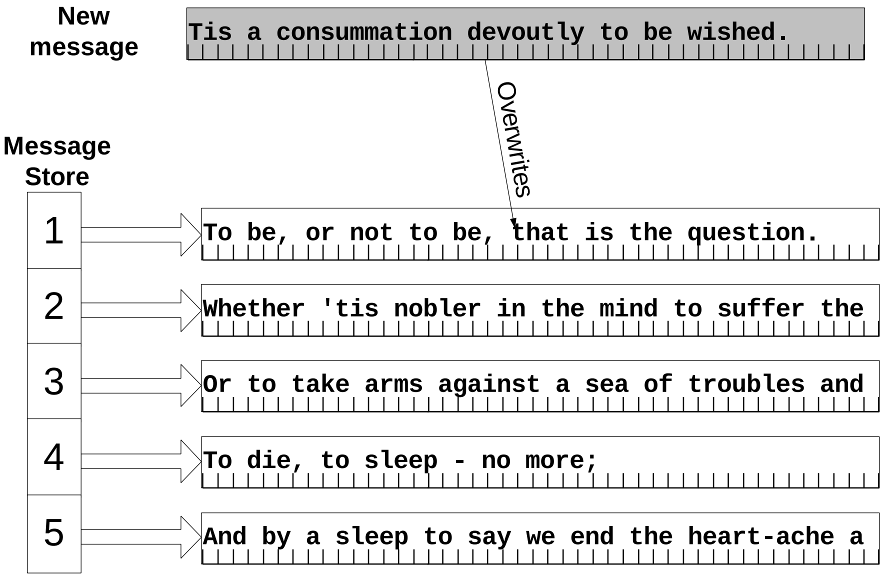

New message
Tis a consummation devoutly to be wished.
Overwrites
Message Store
1
To be, or not to be, that is the question.
2
Whether 'tis nobler in the mind to suffer the
3
Or to take arms against a sea of troubles and
4
To die, to sleep - no more;
5
And by a sleep to say we end the heart-ache a

Figure 2: Circular Buffer Message Store

1. Java Implementation

The basic `MessageStore` class stores a two-dimensional array of messages, and array of message lengths, the index of the oldest message in the store, and a count of the number of messages in the store. It's impossible to avoid memory allocation using the existing versions of the Java String and StringBuffer classes [Chan et al 1998], so we have implemented the store using character arrays. A more robust implementation would create a `FixedMemoryString` class to make it easier to manipulate these character arrays, or else modify the StringBuffer class to allow clients to use it without memory allocation.

```
class MessageStore
{
    protected char[][] messages;
    protected int[] messageLengths;
    protected int messageSize;
    protected int oldestMessage;
    protected int size;
    public int messageSize() {return messages[0].length;}
    public int capacity() {return messages.length;}
```

The `MessageStore constructor` has two initialisation parameters: the number of messages to store (`capacity`) and the maximum size of each message (`maximumMessageLength`). The constructor allocates all the memory ever used; no other method allocates memory either explicitly or through library operations.

```
    public MessageStore(int capacity, int messageSize) {
        this.messageSize = messageSize;
        messages = new char[capacity][messageSize];
        messageLengths =  new int[capacity];
        oldestMessage = size = 0;
    }
```

The most important method adds a new message into the store. This method silently overwrites earlier messages received and truncates messages that are longer than the `chunkSize.` Note that this message accepts a character array and length as parameters, to avoid the allocation implicit in using Java Strings.

```
    public void acceptMessage(char[] msg, int msgLength) {
        int nextMessage = (oldestMessage + size) % capacity();

        messageLengths[nextMessage] = Math.min(msgLength, messageSize);
        System.arraycopy(msg, 0, messages[nextMessage], 0,
                         messageLengths[nextMessage]);
        if (size == capacity()) {
            oldestMessage = (oldestMessage + 1) % capacity();
        } else {
            size++;
        }
    }
```

The `getMessage` method retrieve a messages from a message store. Again, to avoid using the Java `String` classes the client must pass in a pre-allocated buffer and the method returns the length of the string copied into the buffer (see **LENDING** in the **SMALL INTERFACES** pattern).

```
    public int getMessage(int i, char[] destination) {
        int msgIndex = (oldestMessage + i) % capacity();
        System.arraycopy( messages[msgIndex], 0, destination, 0,
                          messageLengths[msgIndex]);
        return messageLengths[msgIndex];
    }
```

2. C++ Implementation

C++ does less memory allocation than Java, so the same example in C++ looks more conventional than the Java example. The main difference is that all the messages are stored inside one large buffer (`messageBuffer`) rather than as a two-dimensional array.

```
class MessageStore {
private:
    char* messageBuffer;
    int oldestMessageNumber;
    int nMessagesInStore;
    int maxMessageLength;
    int maxMessagesInStore;
```

The constructor is straightforward:

```
public:
    MessageStore(int capacity, int maxMsgLength)
        : maxMessagesInStore( capacity ),
          maxMessageLength( maxMsgLength ),
          oldestMessageNumber( 0 ),
          nMessagesInStore( 0 ) {
            messageBuffer = new char[Capacity() * MessageStructureSize()];
    }
```

Note that `MessageStructureSize()` is one byte larger than `maxMessageLength` to cope with the null character `'\0'` on the end of every C++ string:

```
    int NMessagesInStore()      { return nMessagesInStore; }
    int Capacity()              { return maxMessagesInStore; }
    int MessageStructureSize()  { return maxMessageLength+1; }
```

The `AcceptMessage` function copies a new message from a C++ string:

```
        void AcceptMessage(const char* newMessageText) {
            int nextMessage = (oldestMessageNumber + NMessagesInStore()) % Capacity();
            int newMessageLength = strlen( newMessageText );
            int nBytesToCopy = min(newMessageLength,maxMessageLength)+1;
            strncpy(MessageAt(nextMessage), newMessageText, nBytesToCopy);
            MessageAt(nextMessage)[maxMessageLength] = '\0';
            if (NMessagesInStore() == Capacity()) {
                oldestMessageNumber = (oldestMessageNumber + 1) % Capacity();
            } else {
                nMessagesInStore++;
            }
        }
```

Accessing a message is easy: we simply return a pointer directly into the message buffer ('Borrowing' – see **SMALL INTERFACES**). Calling `GetMessage` with index 0 returns the oldest message, 1 the next oldest, etc.

```
    const char* GetMessage(int i) {
            int messageIndex = (oldestMessageNumber + i) % Capacity();
            return MessageAt(messageIndex);
        }
```

`AcceptMessage` uses an auxiliary function to locate each message buffer:

```
    private:
        char * MessageAt( int i ) {
            return messageBuffer + i*MessageStructureSize();
        }
    };
```

❖ ❖ ❖

Known Uses

Many procedural languages support only fixed allocation, so most FORTRAN and COBOL programs use only this pattern, allocating large arrays and never calling `new` or `malloc`. Nowadays, most popular languages support **VARIABLE ALLOCATION** by default, so it can be hard to revert to **FIXED ALLOCATION**. Many real-time systems use this pattern too: dynamic memory allocation and compaction can take an unpredictable amount of time. The *Real-Time Specification for Java* supports Fixed Allocation directly, by allowing objects to be allocated from an `ImmutableMemory` area [Bollella et al 2000].

Safety-critical systems frequently use this pattern, since dynamic memory allocation systems can fail if they run out of memory. Indeed the UK's Department of Defence regulations for safety-critical systems permitted only Fixed Allocation, although this has been relaxed recently in some cases [Matthews 1989, DEF-STAN 00-55 1997]. For example, in a smart mobile phone the telephone application must always be able to dial the emergency number (112, 999 or 911). A smart phone's telephone application typically pre-allocates all the objects it needs to make such a call – even though all its other memory allocation is dynamic.

Strings based on fixed-size buffers use this pattern. EPOC, for example, provides a template class `TBuf<int s>` representing a string up to *s* characters in length [Symbian 99]. Programmers must either ensure than no strings can ever be allowed to overflow the buffer, or else truncate strings where necessary.

See Also

VARIABLE ALLOCATION saves memory by allocating only enough memory to meet immediate requirements, but requires more effort and overhead to manage and make memory requirements harder to predict.

POOLED ALLOCATION can provide memory for a larger number of small objects, by allocating space for a fixed-size number of items from a fixed-sized pool. Pooled allocation can also provide the same interface as variable allocation while allocating objects from a fixed-size memory space.

If you cannot inline whole objects into other objects, you may be able to use **EMBEDDED POINTERS** as an alternative.

MEMORY LIMIT can offer a more flexible approach to the same problem by permitting dynamic allocation while limiting the total memory size allocated to a particular component.

READ-ONLY MEMORY is static by nature and always uses **FIXED ALLOCATION**.

Variable Allocation

Also known as: Dynamic Allocation

How can you avoid unused empty space?

- You have varying or unpredictable demands on memory.
- You need to minimise your program's memory requirements, or
- You need to maximise the amount of data your can store in a fixed amount of memory.
- You can accept the overhead of heap allocation.
- You can handle the situations where memory allocation may fail at any arbitrary point during the processing.

You have a *variable amount of data* to store. Perhaps you don't know how much data to store, or how it will be distributed between the classes and objects in your system.

For example, the Strap-It-On's famous Word-O-Matic word-processor stores part of its current document in main memory. The amount of memory this will require is unpredictable, because it depends upon the size of the document, the screen size resolution, and the fonts and paragraph formats selected by the user. To support a voice output feature beloved by the marketing department, Word-O-Matic also saves the vocal emotions for each paragraph; some documents use no emotions, but others require the complete emotional pantechnicon: joy, anger, passion, despair, apathy. It would be very difficult indeed to pre-allocate suitable data structures to handle Word-O-Matic's requirements, because this would require balancing memory between text, formats, fonts, emotions, and everything else. Whatever choices you made, there would still be a large amount of memory wasted most of the time.

Writing a general-purpose library is even more complex than writing an application, because you can't make assumptions about the nature of your clients. Some clients may have fixed and limited demands; others might legitimately require much more, or have needs that vary enormously from moment to moment.

So how can you support flexible systems while minimising their use of memory?

Therefore*: Allocate and deallocate variable-sized objects as and when you need them.*

Store the data in different kinds of objects, as appropriate, and allocate and free them dynamically as required. Implement the objects using dynamic structures such as linked lists, variable-length collections and trees.

Ensure objects that are no longer needed are returned for reuse, either by making explicit calls to release the memory, or by clearing references so that objects will be recovered by **REFERENCE COUNTING** or a **GARBAGE COLLECTOR.**

For example, Word-O-Matic dynamically requests memory from the system to store the user's documents. To produce voice output Word-O-Matic just requests more memory. When it no longer needs the memory (say because the user closes a document) the program releases the memory back to the Strap-It-On system for other applications to use. If Word-O-Matic runs out of memory, it suggests the user free up memory by closing another application.

Consequences

Variable-size structures avoid unused empty space, thus *reducing memory requirements* overall and generally *increasing design quality.* Because the program does not have assumptions about the amount of memory built into it directly, it is more likely to be *scaleable*, able to take advantage of more memory should it become available.

A heap makes it easier to define interfaces between components; one component may create allocate objects and pass them to another, leaving the responsibility for freeing memory to the second. This makes the system easier to program, improving the *design quality* and making it *easier to maintain*, because you can easily create new objects or new fields in objects without affecting the rest of the allocation in the system.

Allocating memory throughout a program's execution (rather than all at the beginning) can decrease a program's *start-up time*.

However: There will be a *memory overhead* to manage the dynamically allocated memory; typically a two word header for every allocated block. Memory allocation and deallocation require *processor time*, and this cost can be *global* to the language runtime system, the operating system or even, in the case of the ill-fated Intel 432, in hardware. The memory required for typical and worse case scenarios can become *hard to predict*. Because the objects supplied by the heap are of varying size, heaps tend to get *fragmented*, adding to the *memory overhead* and *unpredictability*,

Furthermore, you must be prepared to handle the situation where memory runs out. This may happen *unpredictably* at any point where memory is allocated; handling it *adds additional complexity* to the code and requires additional *programmer effort* to manage, and time and energy to *test properly*. Finally, of couse, it's impossible to use variable allocation in *read-only memory*.

❖ ❖ ❖

Implementation

Using this pattern is trivial in most object-oriented languages; it's what you do by default. Every OO language provides a mechanism to create new objects in heap memory, and another mechanism (explicit or implicit) to return the memory to the heap, usually invoking an object cleanup mechanism at the same time [Ingalls 1981].

1. Deleting Objects

It's not just enough to create new objects, you also have to recycle the memory they occupy when they are no longer required. Failing to dispose of unused objects causes *memory leaks*, one of the most common kinds of bugs in object-oriented programming. While a workstation or desktop PC may be able to tolerate a certainly amount of memory leakage, systems with less memory must conserve memory more carefully.

There are two main kinds of techniques for managing memory: *manual* and *automatic*. The manual technique is usually called *object deletion*; example mechanisms are C++'s `delete` keyword and C's `free` library call. The object is returned to free memory immediately during the operation. The main problem with manual memory management is it is easy to omit to delete objects. Forgetting to delete an object results in a memory leak. A more dramatic problem is to delete an object that is still in use; this can cause your system to crash, especially if the memory is then reallocated to some other object.

In contrast, automatic memory management techniques like **REFERENCE COUNTING** and **GARBAGE COLLECTION** do not require programs to delete objects directly; rather they work out 'automatically' which objects can be deleted, by determining which objects are no longer used in the program. This may happen some time after the object has been discarded. Automatic management prevents the bugs caused by deleting objects that are still in use, since an object still referenced will never be deleted. They also simplify the code required to deal with memory management. Unfortunately, however, it's quite common to have collections, or static variables, still containing references to objects that are no longer actually required. Automatic techniques can't delete these objects, so they remain – they are memory leaks. It remains the programmer's responsibility to ensure this doesn't happen.

2. Signalling Allocation Failure

Variable allocation is flexible and dynamic, so it can tune a systems memory allocation to suit the instantaneous demands of the program. Unfortunately because it is dynamic the program's memory use is unpredictable, and it can fail if the system has insufficient memory to meet a program's request. Allocation failures need to be communicated to the program making the request.

2.1. Error Codes. Allocation functions (such as `new`) to return an error code if allocation fails. This is easy to implement: C's `malloc` for example, returns a null pointer on failure. This approach requires programmer discipline and leads to clumsy application code, since you must check for allocation failure every time you allocate some memory.. Furthermore, every component interface must specify mechanisms to signal that allocation has failed. In practice, this approach, although simple, should be used as a last resort.

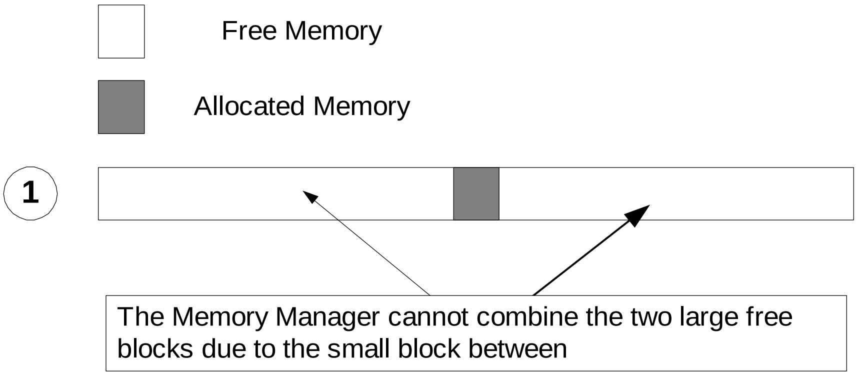
2.2. Exceptions. Far easier for the programmer allocating memory is to signal failure using an exception.. The main benefit of exceptions is that the special case of allocation failure can be handled separately from the main body of the application, while still ensuring the allocation failure does not go unnoticed. Support for exceptions can increase code size, however, and make it more difficult to implement code in intermediate functions that must release resources as a result of the exception.

2.3. Terminating the program. Ignoring allocation failure altogether is the simplest possible approach. If failure does happen, therefore, the system can try to notify the user as appropriately, then abort. For example many MS Windows put up a dialog box on heap failure, then terminate. This approach is clearly only suitable when there is significantly more memory available than the application is likely to need — in other words, when you are not in a memory limited environment. Aborting the program is acceptable in the one case where you are using the system heap to implement **FIXED ALLOCATION** by providing a fixed amount of memory before your program has begun executing, because you cannot tolerate failure once the program has actually started running.

3. Avoiding Fragmentation

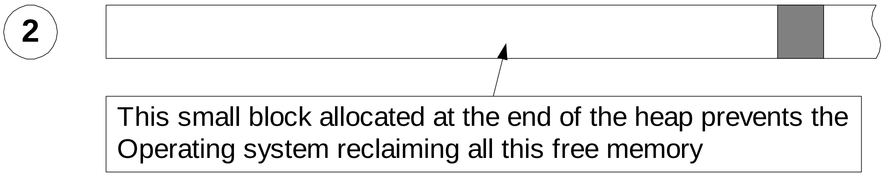
Fragmentation can be a significant problem with variable allocation from a heap for some applications. Often, the best way to address fragmentation is not to worry about it until you suspect that it is affecting your program's performance. You can detect a problem by comparing the amount of memory allocated to useful objects with the total memory available to the system (see the **MEMORY LIMIT** pattern for a way of counting the total memory allocated to objects). Memory that is not in use but cannot be allocated may be due to fragmentation.

3.1. Variable Sized Allocation. Avoid interspersing allocations of very large and small objects. For example, a small object allocated between two large objects that are then freed will prevent the heap manager combining the two large empty spaces, making it difficult to allocate further larger blocks. You can address this using having two or more heaps for different sized objects: Microsoft C++, for example, uses one separate heap for allocations of less than about 200 bytes, and a second heap for all other allocations [Microsoft 97].



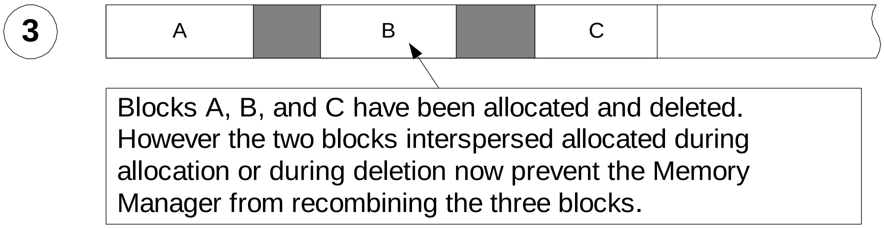
The Memory Manager cannot combine the two large free blocks due to the small block between

3.2. Transient Objects. Operating systems can often reclaim unused memory from the end of the heap, but not from the middle. If you have a transient need for a very large object, avoid allocating further objects until you've de-allocated it. Otherwise, you may leave a very large 'hole' in the heap, which never gets filled and cannot be returned to the wider system until your application terminates. For the same reason, be careful of reallocating buffers to increase their size; this leave the memory allocated to the old buffer unused in the middle of the heap. Use the **MEMORY DISCARD** pattern to provide specialised allocation for transient objects,



This small block allocated at the end of the heap prevents the Operating system reclaiming all this free memory

3.3. Grouping Allocations. Try to keep related allocations and de-allocations together, in preference to interspersing them with unrelated heap operations. This way, the allocations are likely to be contiguous, and the de-allocations will free up all of the contiguous space, creating a large contiguous area for reallocation.



Blocks A, B, and C have been allocated and deleted. However the two blocks interspersed allocated during allocation or during deletion now prevent the Memory Manager from recombining the three blocks.

4. Standard Allocation Sizes

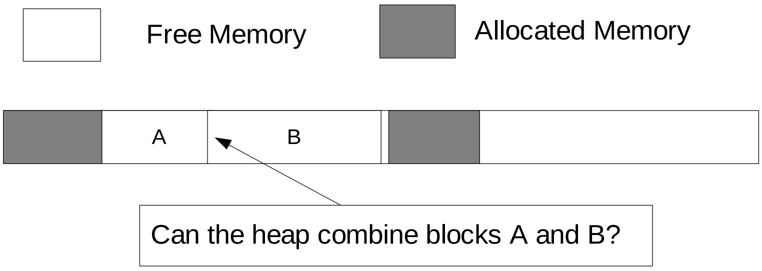
Normally applications tell the heap the sizes of the objects they need to allocate, and the heap allocates memory to store those objects. Another way around fragmentation is for the heap to specify the sizes of memory blocks available to the application. This works if you can afford to waste unused memory inside small blocks (internal fragmentation), if all the objects you allocated have the same size (**POOLED ALLOCATION**) or the application can support non-contiguous allocation. For example, if you can use 10 1K memory blocks rather then one 10K block, the heap will be much more likely to be able to meet your requirements when memory is low.

5. Implementing a Heap

A good implementation of a heap has to solve a number of difficult problems:

- How to represent an allocated cell.
- How to manage the 'free list' of blocks of deallocated memory

- How to ensure requests for new blocks reuse a suitable free block.
- How to prevent the heap becoming fragmented into smaller and smaller blocks of free memory combining adjacent free blocks.



Knuth [1997] and Jones and Lins [1996] describe many implementations of memory heaps and garbage collectors that address these problems; Goldberg and Robson [1983] present a detailed example in Smalltalk. In practice, however, you'll almost always be better off buying or reusing an existing heap implementation. Libraries for low-level memory management, such as Doug Lea's `malloc` [Lea 2000], are readily available.

Example

We can implement a Message Store using **Variable Allocation** from the Java heap. The implementation of this version is much simpler, even though we've kept the strange character-array interface for compatibility with the **Fixed Allocation** version. Because we're able to allocate objects on the heap at any time, we can use library routines that allocate heap memory, and we can rely on the Java built-in memory failure exceptions and garbage collection to deal with resource limitations and object deletion.

The `HeapMessageStore` class simply uses a Java vector to store messages:

```
class HeapMessageStore  {
    protected Vector messages = new Vector();
```

To accept a message, we simply add a string into the vector.

```
    public void acceptMessage(char[] msg, int msgLength) {
        messages.addElement(new String(msg, 0, msgLength));
    }
```

Of course, these allocations could fail if there is not enough memory, propagating exceptions into the client code.

To return a message, we can copy the string into the array provided by the client, keeping the same interface as the **Fixed Allocation** version

```
    public int getMessage(int i, char[] destination) {
        String result = (String) messages.elementAt(i);
        result.getChars(0, result.length(), destination, 0);
        return result.length();
    }
```

Or more simply we could just return the string — if the rest of the system permitted, we could add messages by storing a string directly as well:

```
    public String getMessage(int i) {
        return (String) messages.elementAt(i);
    }
```

Finally, we now need to provide a way for clients to delete messages from the store, since they are no longer overwritten automatically:

```
public void deleteMessage(int i) {
        messages.removeElementAt(i);
    };
```

This relies on Java's **GARBAGE COLLECTION** to clean up the String object and any objects in the internal implementation of the Vector.

❖ ❖ ❖

Known Uses

Virtually all object-oriented programming languages support this pattern by encouraging the use of dynamically allocated memory and by providing libraries based on variable allocation. The vast majority of C++, Smalltalk, and Java applications use this pattern by default. Other languages that encourage dynamic memory allocation also encourage this pattern; hence most C, Pascal, and Lisp programs use this pattern too. Most environments provide dynamically-allocated strings, which use variable-length data structures, and dynamic languages like Smalltalk and Java provide built in garbage collectors to manage dynamically varying storage requirements.

See Also

COMPACTION can reduce the memory overhead from fragmentation, usually at a cost in time performance. If the memory runs out, the program should normally suffer only a **PARTIAL FAILURE**. Using **FIXED ALLOCATION** avoids the overhead, unpredictability and complexity of a variable sized structure at the cost of often allocating more memory than is actually required. **MULTIPLE REPRESENTATIONS** can switch between different variable-sized structures for particular cases. You can limit the memory allocated to specific components by imposing a **MEMORY LIMIT**.

The **HYPOTH-A-SIZE** collection pattern optimises allocation of variable-sized structures [Auer and Beck 1996].

Exceptional C++ [Sutter 2000], *Advanced C++* [Coplien 1994], and *More Effective C++* [Meyers 1996] describe various programming techniques to ensure objects are deleted correctly in C++.

Doug Lea's describes the design of his memory allocator, `malloc,` in *A Memory Allocator* [Lea 2000]. Many versions of the Unix system use this allocator, including Linux. Paul Wilson and Mark Johnston have conducted several surveys of the performance of memory that demonstrate standard allocation algorithms (such as Doug Lea's) are suitable most programs [Johnstone and Wilson 1998].

Lycklama [1999] describes several situations where unused Java objects will not be deleted, and techniques for avoiding them.

Memory Discard

Also known as: Stack Allocation, Scratchpad.

How can you allocate temporary objects?

- You are doing OO programming with limited memory
- You need transient objects that all last only for a well-defined time.
- You don't want temporary allocations to interfere with the allocation of permanent objects.
- You don't want the complexity of implementing garbage collection or the overhead of heap allocation.
- These objects don't own non-memory resources; or have simple mechanisms to free them

Dynamic allocation techniques can impose significant overheads. For example, the designers of the Strap-It-On's 'Back Seat Jet Pilot' application fondly hoped to connect the Strap-It-On to the main control system of a commercial jet plane, allowing passengers to take over in an emergency! The method that calculates the control parameters uses a large number of temporary objects and must execute about fifty times a second. If the objects were allocated from the Strap-It-On's system heap, the cycle time would be too slow, and the jet would crash immediately.

Similar (but less farfetched, perhaps) situations are common in programming. You often need a set of transient objects with lifetimes that are closely linked to the execution of the code; the most common example being objects that last for the duration of a method invocation.

FIXED ALLOCATION is unsuitable for temporary objects because, by definition, it allocates space permanently and requires you to know exactly which objects will be required in advance. **VARIABLE ALLOCATION** isn't suitable for allocating such transient objects either, as it can be relatively slow and lots of temporary objects can fragment the heap.

Therefore*: Allocate objects from a temporary workspace and discard it on completion.*

Use a program stack frame, a temporary heap, or a pre-allocated area of memory as a temporary workspace to store transient objects. Allocate objects from this memory area by incrementing an appropriate pointer. Deallocate all the objects simultaneously by discarding or resetting the memory area. If necessary keep a list of other resources owned by the transient objects and release these explicitly.

For example, the Back Seat Jet Pilot application pre-allocates a buffer (**FIXED ALLOCATION**), and allocates each temporary object by incrementing a pointer within this buffer. On return from the calculation method, the pointer is reset to the start of the buffer, effectively discarding all of the temporary objects. This made the calculation of the jet plane controls quite fast enough, so that when the Aviation Authorities banned Back Seat Jet Pilot on safety grounds the designers were triumphantly able to convert it to create the best-selling Strap-Jet Flight Simulator.

Consequences

Both memory allocation and de-allocation are very fast, improving the system's *time performance*. The time required is fixed, making it suitable for *real-time* systems. Initialising the memory area is fast, so *start-up time* costs are minimal. The temporary workspace doesn't last long, which avoids *fragmentation*.

The basic pattern is easy to program, requiring little *programmer effort*.

By quarantining transient objects from other memory allocations, this pattern can make the memory consumption of the whole system *more predictable,* ensuring transient objects remain a strictly *local* affair.

However: *Programmer Discipline* is required to allocate transient objects from the temporary workspace, and to manage any external resources owned by the objects, and to ensure the transient objects are not used after the workspace has been discarded or recycled. In particular, if the temporary objects use objects from external libraries, these may allocate normal heap memory, or operating system handles.

Because this pattern increases the program's complexity, it also increases its *testing cost.*

Languages that rely on automatic memory management generally do not support the **MEMORY DISCARD** pattern directly: the point of automatic memory management is that it discards the objects for you.

❖ ❖ ❖

Implementation

Here are some issues to consider when implementing **MEMORY DISCARD**.

1. Stack Allocation

In languages that support it, such as C++ and Pascal, stack allocation is so common that we take it for granted; generally, Stack Allocation is the most common form of Memory Discard. Objects are allocated on the program stack for a method or function call, and deallocated when the call returns. This very easy to program, but supports only objects with the exact lifetime of the method.

Some C and C++ environments even allow variable-sized allocation on the stack. Microsoft C++, for example, supports `_alloca` to allocate memory that lasts only till the end of the function [Microsoft 97]

```
   void* someMemory = _alloca( 100 );  // Allocates 100 bytes on the stack
```

GNU G++ has a similar facility [Stallman 1999]. These functions are not standard, however, and no form of stack allocation is possible in standard Java or Smalltalk.

2. Temporary Heaps

You can allocate some memory permanently (**FIXED ALLOCATION**) or temporarily (**VARIABLE ALLOCATION**), and create your objects in this area. If you will delete the transient objects on mass when you delete the whole workspace, you can allocate objects simply by increasing a pointer into the temporary workspace: this should be almost as efficient as stack allocation. You can then recycle all the objects in the heap by resetting the pointer back to the start of the workspace, or just discard the whole heap when you are done.

A temporary heap is more difficult to implement than stack allocation, but has the advantage that you can control the lifetime and size of the allocated area directly.

2.1. Using operating system heaps in C++. Although there are no standard C++ functions that support more than one heap, many environments provide vendor-specific APIs to multiple heaps. EPOC, for example, provides the following functions to support temporary heaps [Symbian99]:

`UserHeap::ChunkHeap`	Creates a heap from the system memory pool
`Rheap::SwitchHeap`	Switches heaps, so that all future allocations for this thread comes from the heap passed as a parameter.

`Rheap::FreeAll`	Efficiently deletes all objects allocated in a heap.
`Rheap::Close`	Destroys a heap

MS Windows CE and other Windows variants provide the functions [Microsoft 97]:

`HeapCreate`	Creates a heap from the system memory pool.
`HeapAlloc, HeapFree`	Allocate and releases memory from a heap passed as a parameter
`HeapDestroy`	Destroys a heap

PalmOs is designed for much smaller heaps than either EPOC or CE, and doesn't encourage multiple dynamic heaps. Of course, Palm applications do **APPLICATION SWITCHING**, so discard all their dynamic program data regularly.

2.2. Using C++ placement new. If you implement your own C++ heap or use the Windows CE heap functions, you cannot use the standard version of operator `new`, because it allocates memory from the default heap. C++ includes the placement `new` operator that constructs an object within some memory that you supply [Stroustrup 1997]. You can use the placement new operator with any public constructor:

```
void* allocatedMemory = HeapAlloc( temporaryHeap, sizeof( MyClass ) );
MyClass* pMyClass = new( allocatedMemory ) MyClass;
```

Placement new is usually provided as part of the C++ standard library, but if not it's trivial to implement:

```
void* operator new( size_t /*heapSizeInBytes*/, void* memorySpace ) {
    return memorySpace;
}
```

3. Releasing resources held by transient objects

Transient objects can own resources such as heap memory or external system objects (e.g. file or window handles). You need to ensure that these resources are released when the temporary objects are destroyed. C++ guarantees to invoke destructors for all stack-allocated objects whenever a C++ function exits, either normally or via an exception. In C++ '*resource de-allocation is finalisation*' [Stroustrup 1995] so you should release resources in the destructor. The C++ standard l library includes the `auto_ptr` class that mimics a pointer, but deletes the object it points to when it is itself destructed, unless the object has been released first. (See **PARTIAL FAILURE** for more discussion of `auto_ptr`).

It's much more complex to releasing resources held by objects in a temporary heap, because the heap generally does not know the classes of the objects that are stored within it. Efficient heap designs do not even store the number of objects they contains, but simply the size of the heap and a pointer to the next free location.

If you do keep a list of every object in a temporary heap, and can arrange that they all share a common base class, you can invoke the destructor of each object explicitly:

```
object->~BaseClass();
```

But it's usually simpler to ensure that objects in temporary heaps do not hold external resources.

When resources can be **SHARED**, so that there may be other references to the resources in addition to the transient ones, simple deallocation from the destructor may not be enough, and you may need to use **REFERENCE COUNTING** or even **GARBAGE COLLECTION** to manage the resources.

4. Dangling Pointers

You have to be very careful about returning references to discardable objects. These references will be invalid once the workspace has been discarded. Accessing objects via such 'dangling pointers' can have unpredictable results, especially if the memory that was used for the temporary workspace is now being used for some other purpose, and it takes care and *programmer discipline* to avoid this problem.

Example

This C++ example implements a temporary heap. The heap memory itself uses **FIXED ALLOCATION;** it's allocated when during the heap object initialisation and lasts as long as the heap object. It supports a `Reset()` function that discards all the objects within it. The heap takes its memory from the system-wide heap so that its size can be configured during initialisation.

Using such a heap is straightforward, with the help of another overloaded operator new. For example the following creates a 1000-byte heap, allocates an object on it, then discards the object. The heap will also be discarded when `theHeap` goes out of scope. Note that the class `IntermediateCalculationResult` may not have a destructor.

```
TemporaryHeap theHeap( 1000 );
IntermediateCalculationResult* p =
                new( theHeap ) IntermediateCalculationResult;
theHeap.Reset();
```

The overloaded operator new is, again, simple:

```
void * operator new ( size_t heapSizeInBytes, TemporaryHeap& theHeap ) {
    return theHeap.Allocate( heapSizeInBytes );
}
```

1. TemporaryHeap Implementation

The `TemporaryHeap` class records the size of the heap, the amount of memory currently allocated, and keeps a pointer (`heapMemory`) to that memory.

```
class TemporaryHeap {
private:
    size_t nBytesAllocated;
    size_t heapSizeInBytes;
    char* heapMemory;
```

The constructor and destructor for the heap class are straightforward; any allocation exceptions will percolate up to the client:

```
TemporaryHeap::TemporaryHeap( size_t heapSize)
    : heapSizeInBytes( heapSize )  {
    heapMemory = new char[heapSizeInBytes];
    Reset();
}

TemporaryHeap::~TemporaryHeap() {
    delete[] heapMemory;
}
```

The function to allocate memory from the `TemporaryHeap` increases a count and throws the `bad_alloc` exception if the heap is full.

```
void * TemporaryHeap::Allocate(size_t sizeOfObject) {
    if (nBytesAllocated + sizeOfObject >= heapSizeInBytes)
        throw bad_alloc();

    void *allocatedCell = heapMemory + nBytesAllocated;
    nBytesAllocated += sizeOfObject;
    return allocatedCell;
}
```

The `Reset` function simply resets the allocation count.

```
void TemporaryHeap::Reset() {
    nBytesAllocated = 0;
}
```

❖ ❖ ❖

Known Uses

All object-oriented languages use stack allocation for function return addresses and for passing parameters. C++ and Eiffel also allow programmers to allocate temporary objects on the stack [Stroustrup 1997, Meyer 1992]. The Real-time Specification for Java will support Memory Discard by allowing programmers to create `ScopedMemory` areas that are discarded when they are no longer accessible by real-time threads [Bollella et al 2000].

In Microsoft Windows CE, you can create a separate heap, allocate objects within that heap, and then delete the separate heap, discarding every object inside it [Boling 1998]. PalmOS discards all memory chunks owned by an application when it application exits [Palm 2000].

Recently, some Symbian developers were porting an existing handwriting recognition package to EPOC. For performance reasons it had to run in the Window Server, a process that must never terminate. Unfortunately the implementation, though otherwise good, contained small memory leaks – particularly following memory exhaustion. Their solution was to run the recognition software using a separate heap, and to discard the heap when it got too large.

'Regions' are a compiler technique for allocating transient objects that last rather longer than stack frames. Dataflow analysis identifies objects to place in transient regions; the regions are allocated from a stack that is independent of the control stack [Tofte 1998].

The 'Generational Copying' **GARBAGE COLLECTORS** [Jones and Lins 1996, Ungar 1984] used in modern Smalltalk and Java systems provide a form of memory discard. These collectors allocate new objects in a separate memory area (the "Eden space"). When this space becomes full, an 'angel with a flaming sword' copies any objects inside it that are still in use out of Eden and into a more permanent memory area. The Eden space is then reset to be empty, discarding all the unused objects. Successively larger and longer-lived memory spaces can be collected using the same technique, each time promoting objects up through a cycle of reincarnation, until permanent objects reach the promised land that is never garbage collected, where objects live for ever.

See Also

APPLICATION SWITCHING is a more coarse-grained alternative, using process termination to discard both heap and executable code. **DATA FILES** often uses stack allocation or a temporary workspace to process each item in turn from secondary storage. The discarded memory area may use either **FIXED ALLOCATION**, or **VARIABLE ALLOCATION**.

POOLED ALLOCATION is similar to **MEMORY DISCARD**, in that both patterns allocate a large block of memory and then apportion it between smaller objects; **POOLED ALLOCATION**, however, supports de-allocation.

Pooled Allocation

Also Known As: Memory Pool

How can you allocate a large number of similar objects?

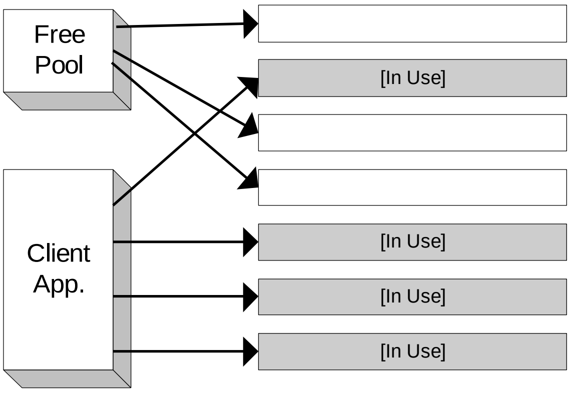
- Your system needs a large number of small objects
- The objects are all roughly the same size.
- The objects need to be allocated and released dynamically.
- You don’t, can’t, aren’t allowed to, or won’t used **VARIABLE ALLOCATION**
- Allocating each object individually imposes a large overhead for object headers and risks fragmentation.

Some applications use a large number of similar objects, and allocate and deallocate them often. For example, Strap-It-On’s ‘Alien Invasion’ game needs to record the positions and states of lots of graphical sprites that represent invading aliens, asteroids, and strategic missiles fired by the players. You could use **VARIABLE ALLOCATION** to allocate these objects, but typical memory managers store object headers with every object; for small objects these headers can double the program’s *memory requirements*. In addition, allocating and deallocating small objects from a shared heap risks fragmentation and increases the time overhead of managing large numbers of dynamic objects.

You could consider using the **FLYWEIGHT** pattern [Gamma+ 1995], but this does not help with managing data that is intrinsic to objects themselves. **MEMORY COMPACTION** can reduce fragmentation but imposes extra overheads in memory and time performance. So how can you manage large numbers of small objects?

Therefore: *Pre-allocate a pool of objects, and recycle unused objects.*

Pre-allocate enough memory to hold a large number of objects at the start of your program, typically by using **FIXED ALLOCATION** to create an array of the objects. This array becomes a ‘pool’ of unused, uninitialised, objects. When the application needs a new object, choose an unused object from the pool and pass it to the program to initialise and use it. When the application is finished with the object, return the object to the pool.



In practice objects do not have to be physically removed and reinserted into the pool (this will be difficult in languages like C++ when the objects are stored directly within the pool array using **FIXED ALLOCATION**). Instead you’ll need to track which pool objects are currently in use and which are free. A linked list (using **EMBEDDED POINTERS**) of the free objects will often suffice.

For example Alien Invasion uses a pool of sprite objects to support the Alien Invasion game. This pool is allocated at the start of the program and holds enough objects to represent all the sprites that can be displayed on the Strap-It-On's high-resolution 2-inch screen. No extra memory is required for each object by the memory manager or runtime system. All unused sprites are kept on a free list, so a new sprite can be allocated or deallocated using just two assignments.

Consequence

By reducing memory used for object headers and lots to fragmentation, pooled allocation lets you store more objects in less memory, reducing the *memory requirements* of the system as a whole. Simultaneously, by allocating a fixed-sized pool to hold all these objects, you can *predict* the amount of memory required exactly. Objects allocated from a pool will be close together in memory, reducing need for **PAGING** overhead in a paged system. Memory allocation and deallocation is fast, increasing *time performance* and *real-time responsiveness*.

However: The objects allocated to the pool are never returned to the heap, so the memory isn't available to other parts of the application, potentially increasing overall *memory requirements*. It takes *programmer effort* to implement the pool, *programmer discipline* to use it correctly, and further effort to *test* that it all works. A fixed-size pool can decrease your program's *scalability*, making it harder to take advantage of more memory should it become available, and also reduce your *maintainability*, by making it harder to subclass pooled objects. Preallocating a pool can increase your system's *startup time*.

❖ ❖ ❖

Implementation

POOLED ALLOCATION combines features of **FIXED ALLOCATION** and **VARIABLE ALLOCATION.** The pool itself is typically statically allocated (so the overall memory consumption is predictable) but objects within the pool are allocated dynamically. Like **FIXED ALLOCATION**, the pooled objects are actually preallocated and have to be initialised before they are used (independently of their constructors). Like **VARIABLE ALLOCATION**, requests to allocate new objects may be denied if the pool is empty, so you have to handle memory exhaustion; and you have to take care to release unused objects back to the pool.

Here are some issues to consider when using **POOLED ALLOCATION**:

1. Reducing Memory Overheads

One reason to use Pooled Allocation is to reduce the amount of memory required for booking in a variable allocation memory manager: pooled allocation needs to keep less information about every individual object allocated, because each objects typically the same size and often the same type. By comparing memory manager overheads with the objects' size, you can evaluate if pooled allocation makes sense in your application. Removing a two-word header from a three-word object is probably worthwhile, but removing a two-word header from a two kilobyte objects is not (unless there are millions of these objects and memory is very scarce).

2. Variable Sized Objects

Pooled allocation works best when every object in the pool has the same size. In practice, this can mean that every object in the pool should be the same class, but this greatly reduces the flexibility of a program's design.

If the sizes of objects you need to allocate are similar, although not exactly the same, you can build a pool capable of storing the largest size of object. This will suffer from internal fragmentation, wasting memory when smaller objects are allocated in a larger space, but this fragmentation is bounded by the size of each pool entry (unlike external fragmentation, which can eventually consume a large proportion of a heap).

Alternatively, you can use a separate pool for each class of objects you need to allocate. Per-class pools can further reduce memory overheads because you can determine the pool to which an object belongs (and thus its class) by inspecting the object's memory address. This means that you do not need to store a class pointer [Goldberg and Robson 1983] or vtbl pointer [Ellis and Stroustrup 1980] with every object, rather one pointer can be shared by every object in the pool. A per-class pool is called "a Big Bag of Pages" or "BiBoP" because it is typically allocated contiguous memory pages so that an object's pool (and thus class) can be determined by fast bit manipulations on its address. [Steele 1977, Dybvig 1994].

3. Variable Size Pools

If your pool is a **FIXED ALLOCATION** you have determine how big the pool should be before your program starts running, and you have no option but to fail a request for a new object when the pool is empty. Alternatively, when the pool is empty you could use **VARIABLE ALLOCATION** and request more memory from the system. This has the advantage that pooled object allocation will not fail when there is abundant memory in the system, but of course it makes the program's memory use more difficult to predict. Flexible pools can provide guaranteed performance when memory is low (from the **FIXED ALLOCATION** portion) while offering extra performance when resources are available.

4. Making Pools Transparent

Sometimes it can be useful for objects that use **POOLED ALLOCATION** to present the same interface as **VARIABLE ALLOCATION**. For example, you could need to introduce pooled allocation into a program that uses variable allocation by default (because the programming language uses variable allocation by default).

In C++ this is easy to implement; you can overload operator `new` to allocate an object from a pool, and operator `delete` to return it. In Smalltalk and Java this approach doesn't work so seamlessly: in Smalltalk you can override object creation, but in Java you cannot reliably override either creation or deletion. In these languages you will need to modify clients to allocate and release objects explicitly.

C++ has a further advantage over more strongly typed languages such a Java. Because we can address each instance as an area of raw memory, we can reuse the objects differently when the client does not need them. In particular, as the picture below shows, we can reuse the first few bytes of each element as an embedded pointer to keep a free list of unallocated objects.

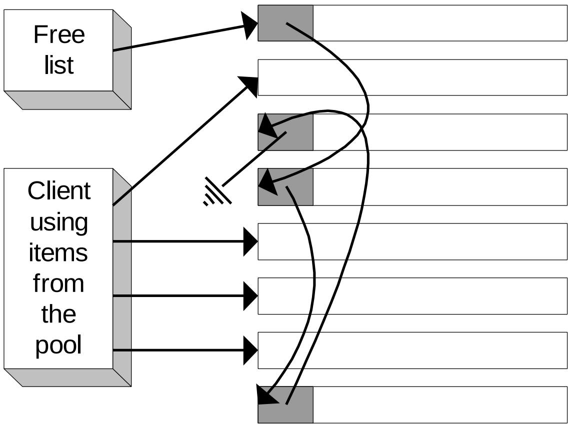


Figure 3: Reusing object memory to implement a free list

The following code uses this technique to implement a C++ class: `TreeNode`, representing part of a tree data structure. Clients use instances of the class as though allocated from the system heap:

```
TreeNode* node = new TreeNode;
delete node;
```

4.1 Implementation. Internally the class uses a static pointer, `freeList`, to the start of the free list. Each TreeNode object is small so we don't want the overhead of heap allocation for each one separately. Instead we allocate them in blocks, of size `BLOCK_SIZE`:

```
class TreeNode {
private:
    enum { BLOCK_SIZE = 10 };
    static void* freeList;

    TreeNode* leftNode;
    TreeNode* rightNode;
    void* data;
// etc.
};
```

(The implementation of a `TreeNode` to give a tree structure using these data members is left as an exercise for the reader!)

`TreeNode` has one static function to allocate new objects when required and add them all to the free list:

```
/* static */
void* TreeNode::freeList=0;

/* static */
void TreeNode::IncreasePool() {
    char* node = new char[BLOCK_SIZE * sizeof(TreeNode)];
    for( int i=0; i<BLOCK_SIZE; i++)
        AddToFreeList( node + (i * sizeof(TreeNode)) );
}
```

To make the **Pooled Allocation** look like **Variable Allocation**, `TreeNode` must implement the operators `new` and `delete`. There's one caveat for these implementations: any derived class will inherit the same implementation. So, in operator `new`, we must check the size of the object being allocated to ensure that we only use this implementation for objects of the correct size, otherwise we allocate the object from the system heap.

```
void* TreeNode::operator new(size_t bytesToAllocate) {
    if( bytesToAllocate != sizeof(TreeNode) )
        return ::operator new( bytesToAllocate );
    if( freeList == 0)
        IncreasePool();
    void *node = freeList;
    freeList = *((void**)node);
    return node;
}
```

Operator `delete` is straightforward (or as straightforward as these operators can ever be). We check that the object is a suitable size to be allocated from the pool, and if so, return it to the free list; otherwise we return it to the heap.

```
void TreeNode::operator delete( void* node, size_t bytesToFree ) {
    if( bytesToFree != sizeof(TreeNode) )
        ::operator delete( node );
    else
        AddToFreeList( node );
}
```

`AddToFreeList` uses the first few bytes of the object as the list pointer:

```
void TreeNode::AddToFreeList( void* node ) {
    *((void**)node) = freeList;
    freeList = node;
}
```

Example

As a contrast to the C++ and Java syntax, here we present a simple Smalltalk implementation of pooled allocation. The class allocates a fixed number of pooled objects when the system starts up and stores them in the class-side array `Pool`. Objects are allocated from the `Pool` as if it were a stack; the class variable `PoolTopObject` keeps track of the top of the stack.

```
Object subclass: #PooledObject
  instanceVariableNames: ''
  classVariableNames:
    'Pool PoolTopObject'
  poolDictionaries: ''
```

The class method `buildPool` initialises all the objects in the pool, and need be called only once on initialisation of the system. Unlike other Smalltalk class initialisation functions, this isn't really a 'compile-time-only' function; a previous execution of the system could have left objects in the pool, so we'll need to call this function to restore the pool to its initial state.

```
PooledObject class

buildPool: poolSize
        "Puts poolSize elements in the pool"
    | newObject |
    Pool := Array new: poolSize.
    (1 to: poolSize) do:
          [ :i | Pool at: i put: PooledObject create. ].
    PoolTopObject = 1.
    ^ Pool
```

We need a `create` class method that `buildPool` can call to create new instances.

```
create
        "Allocates an uninitialised instance of this object"
    ^ super new
```

We can then define the `PooledObject` class `new` method to remove and return the object at the top of the pool.

```
new
    "Allocate a new object from the Pool"
   | newObject |
   newObject := Pool at: PoolTopObject.
   Pool at: PoolTopObject put: nil.
   PoolTopObject := PoolTopObject + 1.
   ^ newObject
```

Clients of the pooled object must send the `free` message to a pooled object when they no longer need it. This requires more discipline than standard Smalltalk programming, which use garbage collection or reference counting to recycle unused objects automatically.

```
free
   "Restores myself to the pool"
   self class free: self
```

The real work of recycling an object is done by the class method `free:` that pushes an object back into the pool.

```
free: aPooledObject
   "Return a pooled object to the pool"
   PoolTopObject := PoolTopObject – 1.
   Pool at: PoolTopObject put: aPooledObject.
```

❖ ❖ ❖

Known Uses

Many operating systems use pooled allocation, and provide parameters administrators can set to control the size of the pools for various operating system resources, such as IO buffers, processes, and file handles. VMS, for example, pre-allocates these into fixed size pools, and allocates each type of objects from the corresponding pool. [Kenah and Bate 1984]. UNIX uses a fixed size pool of process objects (the process table) [Goodheart 1994] and even MS-DOS provides a configurable pool of file IO buffers, specified in the configuration file CONFIG.SYS.

EPOC's Database Server uses a variable sized pool to store blocks of data read from a database, and EPOC's Socket service uses a fixed size pool of buffers [Symbian 1999].

NorTel's Smalltalk implementation of telephone exchange software used pools of Call objects to avoid the real-time limitations of heap allocation in critical parts of the system.

See Also

Memory Pools are often allocated using **FIXED ALLOCATION**, although they can also use **VARIABLE ALLOCATION. MEMORY DISCARD** also allocates many smaller objects from a large buffer; it can handle variable sized objects, though they must all be deleted simultaneously.

MEMORY LIMIT has a similar effect to **POOLED ALLOCATION** as both can cap the total memory used by a component.

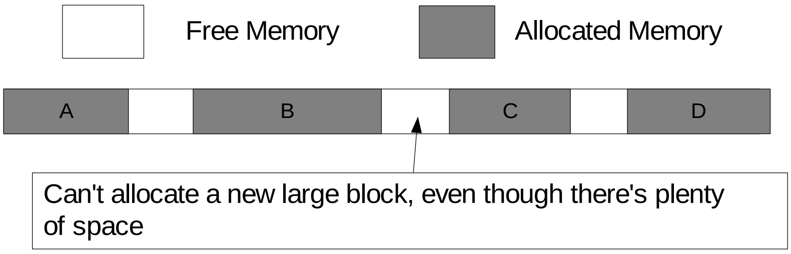
Compaction

Also known as: Managed Tables, Memory Handles, Defragmentation

How do you recover memory lost to fragmentation?

- You have a large number of variable sized objects.
- Objects are allocated and deallocated randomly.
- Objects can change in size during their lifetimes.
- Fragmentation wastes significant amounts of memory space
- You can accept a small overhead in accessing each object.

External fragmentation is a major problem with VARIABLE ALLOCATION. For example, the Strap-It-On™'s voice input decoder has up to a dozen large buffers active at any time, each containing a logical word to decode. They account for most of the memory used by the voice decoder component and can vary in size as decoding progresses. If they were allocated directly from a normal heap, there'd be a lot of memory lost to fragmentation. If objects shrink, more space is wasted between them; one object cannot grow past the memory allocated to another object, but allocating more free memory between objects to leave them room to grow just wastes more memory.



Can't allocate a new large block, even though there's plenty of space

Figure 4: The effect of allocating large buffers

Fragmentation occurs because computer memory is arranged linearly and accessed through pointers. Virtual memory (see PAGING) implements this same linear address space. When you allocate objects in memory you record this allocation and ownership using a pointer: that is, an index into this linear address space.

Therefore*: Move objects in memory to remove unused space between them.*

Space lost by external fragmentation can be recovered by moving allocated objects in memory so that objects are allocated contiguously, one after another: all the previously wasted space is collected at one end of the address space, so moving the objects effectively moves the unused spaces between them.

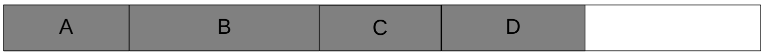


Figure 5: Result of compaction

The main problem with moving objects is ensuring that any references to them are updated correctly: once an object has been moved to a new location, all pointers to its old location are invalid. While it is possible to find and update every pointer to every moved object, it is generally simpler to introduce an extra level of indirection. Rather than having lots of pointers containing the memory address of each object, pointers to refer to a "handle for this object". A handle is a unique pointer to the actual memory address of an object: when you move the allocated object in memory you update the handle to refer to the object's new location, and all other pointers can still access the object correctly.

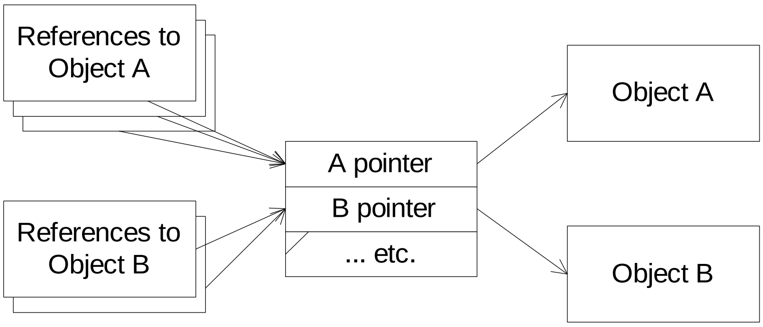


Figure 6: Handles

In the figure above, if you copy object A to a different location in memory and update the 'A Pointer' handle, all external references to object A will remain the same but accesses through them will find the new location.

Thus the Strap-It-On™'s voice input decoder maintains a large contiguous memory area for buffers, and uses a simple algorithm to allocate each buffer from it. Each buffer is accessed through handle. When there's insufficient contiguous space for a new buffer or when an existing buffer needs to grow, even though there's sufficient total memory available, the software moves buffers in memory to free up the space and adjusts the handles to refer to the new buffer locations.

Consequences

You have little or no memory wastage due to external *fragmentation*, reducing the program's *memory requirements* or increasing its capacity within a fixed amount of memory. Compacting data structures can *scale up* easily should more memory become available.

However: You'll need additional code to manage the handles; if the compiler doesn't do this for you, this will require *programmer effort* to implement. Indirect access to objects requires *programmer discipline* to use correctly, and indirection and moving objects increases the program's *testing cost*.

There will be a small additional *time overhead* for each access of each object. Compacting many objects can take a long time, *reducing time performance*. The amount of time required can be *unpredictable*, so standard compaction is often unsuitable for *real-time* applications, although there are more complex incremental compaction algorithms that may satisfy real-time constraints, though such algorithms impose a further *run time* overhead.

❖ ❖ ❖

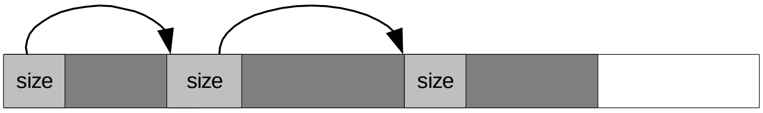
Implementation

Here are some further issues to consider when implementing the **COMPACTION** pattern:

1. Compaction without Handles

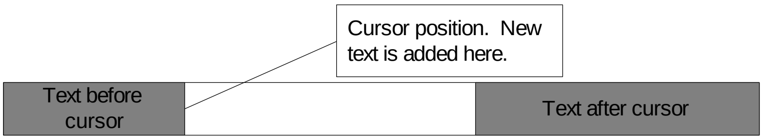
You can compact memory without using explicit handles, provided that you can move objects in memory and ensure they are referenced at their new location.

EPOC's Packed Array template class, `CArrayPakFlat`, is one example of this approach [Symbian 1999]. A Packed Array is a sequential array of items; each element in the packed array contains its size, allowing you to locate the position of the next one. Inserting or deleting an element involves moving all the subsequent elements; the entire array is reallocating and copied if there is insufficient space. The template specialisation `CArrayPak<TAny>` even allows variable-sized elements.



Locating an element by index is slow, though the implementation optimises for some situations by caching the last item found.

Text editors that use an insertion cursor can also use compaction. Text only changes at the cursor position; text before and after the cursor is static. So you can allocate a large buffer, and store the text before the cursor at the start of the buffer and the text after the cursor at the end. Text is inserted directly at the cursor position, without needing to reallocate any memory, however, when the cursor is moved, each character it moves past must be copied from one end of the buffer to the other.



In this case, the indirection is simply the location of the text following the cursor. You store a static pointer to this, so any part of the application can locate it easily.

2. Object tables

If many objects are to be compacted together an *object table* gives better random access performance at a cost of a little more memory. An object table contains all the handles allocated together. Object tables make it easy to find every handle (and thus every object) in the system, making compaction easier. You can store additional information, along with the handle in each *object table entry*, such as a class pointer for objects, count fields for **REFERENCE COUNTING**, mark bits for **GARBAGE COLLECTION**, and status bits or disk addresses for **PAGING**.

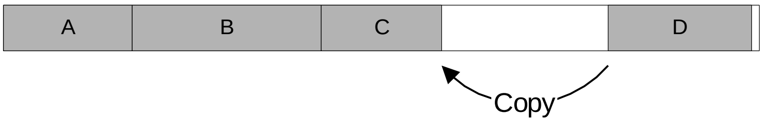
3. Optimising Access to Objects. Using a direct pointer to an object temporarily can increase execution speed compare with indirection through a handle or object table for every access. But, consider the following:

```
SomeClass* p = handle.GetPointer();          // 1
p->FunctionWhichTriggersCompaction();        // 2
p->AnotherFunction();                        // 3. p is now invalid!
```

If the function in line 2 triggers compaction, the object referred to by `handle` may have moved, making the pointer `p` invalid. You can address this problem explicitly by allowing handles to *lock* objects in memory while they're being accessed; objects may not be moved while they are locked. Locking does allow direct access to objects, but requires *programmer discipline* to unlock objects that are not needed immediately, space in each handle to store a lock bit or lock count, and a more complex compaction algorithm that avoids moving locked objects. The PalmOS and MacOs operating systems support lockable handles to most memory objects, so that their system heaps can be compacted [Apple 1985, Palm 2000].

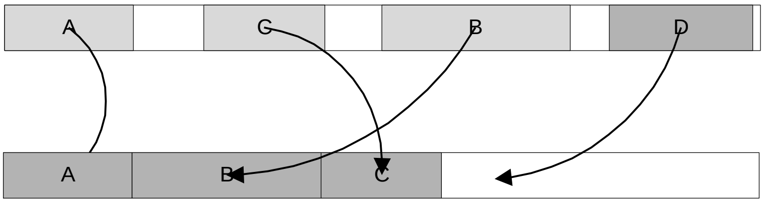
4. Compacting Objects

In the simplest approach, objects are stored in increasing order of memory location. You can compact the objects by copying each one in turn to the high water mark of the ones already compacted.



This approach works well when there is already a logical order on the objects, such as the elements of a sequence. If the sequence is compacted whenever an object is deleted, half the objects will be copied on average.

This does not work so well when objects are not stored in the correct order. In that case a better approach is simultaneously to sort and compact objects by copying them into a different memory space. Copying **GARBAGE COLLECTION** algorithms, for example, copy old objects into a new memory space. Unused objects are not copied, but are discarded when the old space is reused.



5. Compacting on Demand

Persistent or long-lived objects can be compacted occasionally, often on user command. One way to implement this is to store all the persistent objects on to **SECONDARY STORAGE**, reordering them (as described above) as they are stored, then to read them back in. If the objects are compacted rarely, then you can use direct pointers for normal processing, since the time cost of finding and changing all the pointers is paid rarely and under user control.

6. C++ Handle classes

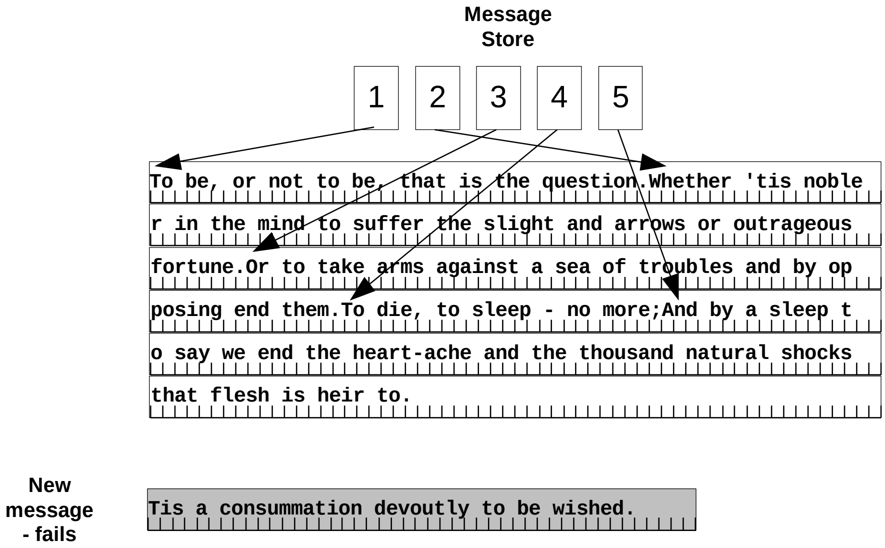
C++'s operator overloading facilities allow us to implement an object with semantics identical to a pointer [Coplien 1994], but which indirects through the object table. Here's an example of a template class that we can use in place of a pointer to an object of class T. The `Handle` class references the object table entry for the underlying object (hence `tableEntry` is a pointer to a pointer), and redefines all the C++ pointer operations to indirect through this entry.

```
template <class T> class Handle {
public:
   Handle( T** p ) : tableEntry( p ) {}
   T* operator->() const { return ptr(); }
   T& operator*() const { return *ptr(); }
   operator T*() const { return ptr(); }
private:
   T* ptr() const { return *tableEntry; }
   T** tableEntry;
};
```

Example

The following Java example extends the `MessageStore` example described in the **FIXED ALLOCATION** and **VARIABLE ALLOCATION** patterns. This version uses memory compaction to permit variable size messages without wasting storage memory. Instead of storing each message in its own separate fixed size buffer, it uses a single buffer to store all the messages, and keeps just the lengths of each message. We've implemented this using **FIXED ALLOCATION,** avoiding `new` outside the constructor.

The figure below shows the new message format:



```
To be, or not to be, that is the question.Whether 'tis noble
r in the mind to suffer the slight and arrows or outrageous
fortune.Or to take arms against a sea of troubles and by op
posing end them.To die, to sleep - no more;And by a sleep t
o say we end the heart-ache and the thousand natural shocks
that flesh is heir to.

Tis a consummation devoutly to be wished.
```

The `CompactingMessageStore` class has a `messageBuffer` to store characters, an array of the lengths of each message, and a count of the number of messages in the store.

```
class CompactingMessageStore {
    protected char[] messageBuffer;
    protected int[] messageLengths;
    protected int numberOfMessages = 0;
```

The constructor allocates the fixed-sized arrays.

```
    public CompactingMessageStore(int capacity, int totalStorageCharacters) {
        messageBuffer = new char[totalStorageCharacters];
        messageLengths =  new int[capacity];
    }
```

We can calculate the offset of each message in the buffer by summing the lengths of the preceding messages:

```
protected int indexOfMessage(int m) {
        int result = 0;
        for (int i = 0; i < m; i++) {
            result += messageLengths[m];
        }
        return result;
    }
```

Adding a new message is simple: we just copy the new message to the end of the buffer. In this implementation, overflow throws an exception rather than overwriting earlier messages as in the **FIXED ALLOCATION** example.

```
    public void acceptMessage(char[] msg, int msgLength) {
        int endOffset = indexOfMessage(numberOfMessages);

        try {
            messageLengths[numberOfMessages] = msgLength;
            System.arraycopy(msg, 0,  messageBuffer,
                             endOffset, msgLength);
        }
        catch (ArrayIndexOutOfBoundsException e) {
            throw new OutOfMemoryError("Message store overflow");
        }

        numberOfMessages++;
    }
```

Retrieving a message is straightforward:

```
public int getMessage(int i, char[] destination) {
    System.arraycopy(messageBuffer, indexOfMessage(i),
                     destination, 0, messageLengths[i]);
    return messageLengths[i];
}
```

The interesting point is what happens when we remove messages from the buffer. To keep everything correct, we have to copy all the messages after the one we've removed forward in the buffer, and move the elements of the `messageLengths` array up one slot:

```
public void deleteMessage(int i) {
    int firstCharToMove = indexOfMessage(i+1);
    int lastCharToMove = indexOfMessage(numberOfMessages);

    System.arraycopy(messageBuffer, firstCharToMove,
                     messageBuffer, indexOfMessage(i),
                     lastCharToMove - firstCharToMove);
    System.arraycopy(messageLengths, i+1, messageLengths, i,
                     numberOfMessages - i - 1);

    numberOfMessages--;
}
```

❖ ❖ ❖

Known Uses

EPOC's Font & Bitmap Server manages large bitmaps **SHARED** between several processes. It keeps the data areas of large bitmaps (>4Kb) in a memory area with no gaps in it – apart from the unused space at the top of the last page. When a bitmap is deleted the Server goes through it's list of bitmaps and moves their data areas down by the appropriate amount, thereby compacting the memory area. The Server then updates all its pointers to the bitmaps. Access to the bitmaps is synchronised between processes using a mutex [Symbian 1999].

The Palm and Macintosh memory managers both use handles into a table of master pointers to objects so that allocated objects can be compacted [Palm 2000, Apple 1985]. Programmers have to be disciplined to lock handles while using the objects to which they refer.

The Sinclair ZX-81 (also known as the Timex Sinclair TS-1000) was based on compaction. The ZX-81 supported an interactive BASIC interpreter in 1K of RAM; the interpreter tables were heavily compacted, so that if you used lots of variables you could only have a few lines of program code, and vice versa. The pinnacle of compaction was in the screen memory: if the screen was blank, it would shrink so that only the end-of-line characters were allocated. Displaying text on the screen caused more screen memory to be allocated, and everything else in memory would be moved to make room.

See Also

FIXED ALLOCATION and **POOLED ALLOCATION** are alternative ways to solve the same problem. By avoiding heap allocation during processing, they avoid fragmentation altogether.

PAGING, REFERENCE COUNTING, and **GARBAGE COLLECTION** can all use compaction and object tables.

Many garbage collectors use for dynamic languages like Lisp, Java, and Smalltalk use **COMPACTION**, with or without objects table and handles. Jones and Lins [1996] presents the most important algorithms. The Smalltalk Blue Book [Goldberg and Robson 1983] includes a full description of a Smalltalk interpreter that uses **COMPACTION** with an object table.

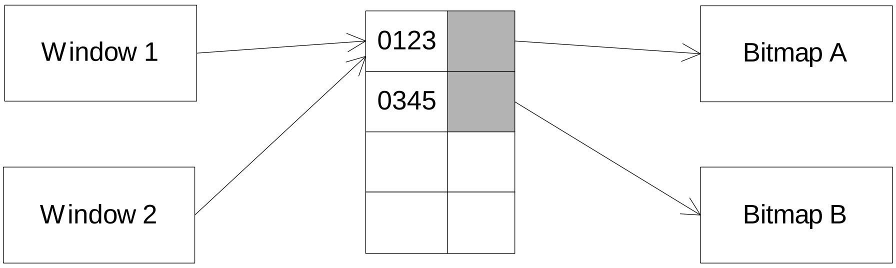
Reference Counting

How do you know when to delete a shared object?

- You are **SHARING** objects in your program
- The shared objects are transient, so their memory has to be recycled when they are no longer needed.
- Interactive response is more important than overall performance.
- The space occupied by objects must be retrieved as soon as possible.
- The structure of shared objects does not form cycles.

Objects are often shared across different parts of components, or between multiple components in a system. If the shared objects are transient, then the memory they occupy must be recycled when they are no longer used by any client. Detecting when shared objects can be deleted is often very difficult, because clients may start or stop sharing them at any time.

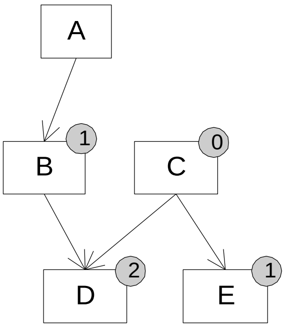
For example, the Strap-It-On Wrist-Mounted PC caches bitmaps displayed in its user interface, so that each bitmap is only stored once, no matter how many windows it is displayed in. The bitmaps are cached in a hash table that maps from bitmap IDs to actual bitmap objects. When a bitmap is no longer required it should be deleted from the cache; in the illustration below, bitmap B is no longer required and can be deleted.



The traditional way to manage memory for bitmaps displayed in windows is to allocate bitmaps when windows are opened, and deallocate bitmaps when windows are closed. This doesn’t work if the bitmaps are cached, because caching aims to use a pre-existing bitmap, if one exists, and to deallocate bitmaps only when all windows that have used them are closed. Deleting a shared bitmap when its first window closes could mean that the bitmap was no longer available to other windows that need it.

Therefore: *Keep a count of the references to each shared object, and delete each object when its count is zero.*

Every shared object needs to have a reference count field which stores the number of other objects that point to it. A reference count must count all references to an object, whether from shared objects, temporary objects, permanent objects, and references from global, local, and temporary variables. The invariant behind reference counting is that an object’s reference count field is accurate count of the number of references to the object. When an object’s reference count is zero it has no references from the rest of the program; there is no way for any part of the program to regain a reference to the object, so the object can be deleted. In the figure below object C has a zero reference count and can be deleted:



When an object is allocated no other objects or variables can refer to it so its reference count is zero. When a reference to the object is stored into a variable or into a field in another object, the object's reference count must be incremented, and similarly when a reference to an object is deleted, the object's reference count must be decremented. When an object's reference count gets back to zero, the object can be deleted and the memory it occupies can be recycled.

There are a couple of important points to this algorithm. First, an assignment to a variable pointing to reference-counted objects involves two reference count manipulations: the reference count of the old contents of the variable must be decremented, and then the reference count of the new contents of the variable must be incremented. Second, if an object itself contains references to reference counted objects, when the object is deleted all the reference counts of objects to which it referred must be decremented recursively. After all, a deleted object no longer exists so it cannot exercise any references it may contain. In the diagram above, once C has been deleted, object E can be deleted and object D will have a reference count of one.

For example, the StrapItOn associates a reference count with each bitmap in the cache. When a bitmap is allocated, the reference count is initialised to zero. Every window which uses a bitmap must first send `attach` to the bitmap to register itself; this increases the bitmap's reference count. When a window is finished with a bitmap, it must send `release` the bitmap; `release` decrements the reference count. When a bitmap's reference goes back to zero, it must have no clients and so deallocates itself.

Consequences

Like other kinds of automatic memory management, reference counting increases the program's *design quality*: you no longer need to worry about deleting dynamically allocated objects. Memory management details do not have to clutter the program's code, making the program easier to read and understand, increasing *maintainability*. Memory management decisions are made *globally,* and implicitly for the whole system, *rather than locally and explicity* for each component. Reference counting also decreases coupling between components in the program, because one component does not need to know the fine details of memory magement implement in other components. This also makes it easier to reuse the component in different contexts with different assumptions about memory use, further improving the system *maintainability*.

The overhead of reference counting is distributed throughout the execution of the program, without any long pauses for running a garbage collector. This provides smooth and predictable performance for interactive programs, and improves the *real-time responsiveness* of the system.

Reference counting works well when memory is low, because it can delete objects as soon as they become unreachable, recycling their memory for use in the rest of the system. Unlike many forms of **GARBAGE COLLECTION**, **REFERENCE COUNTING** permits shared objects to release external resources, such as file streams, windows, or network connections.

However:

Reference counting requires *programmer discipline* to correctly manipulate reference counts in programs. If a reference to an object is created without incrementing the object's reference count, the object could be deallocated, even though the refrence is in use.

Reference counting does not guarantee that memory which the program no longer needs will be recycled by the system, even if reference counts are managed correctly. Reference counting deletes objects which are unrechable from the rest of the program. *Programmer discipline* is required to ensure objects which are no longer required are no longer reachable, but this is easier than working out when shared objects can be manually deleted.

Reference count manipulations imposes large a *runtime* overhead because they occur on every pointer write; this can amount to ten or twenty percent of a programs running time. Allocating memory for reference count fields can increases the *memory requirements* of reference counted objects. Reference counting also increases the program's *memory requirements* for stack space to hold recursive calls when deleting objects objects.

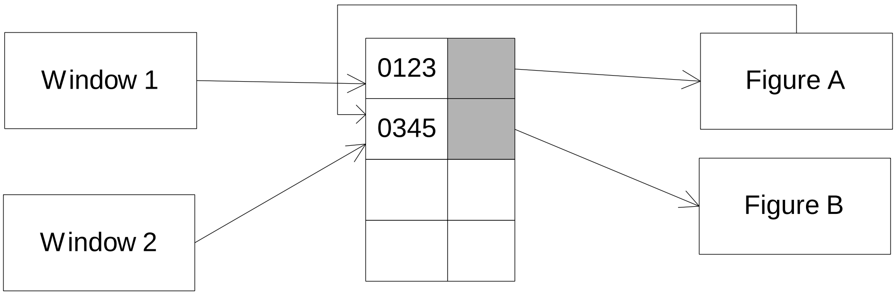
Finally, reference counting doesn't work for cycles of shared objects.

❖ ❖ ❖

Implementation

Reference count manipulations and recursive deletion are basically local operations, affecting single objects and happening at well defined times with respect to the execution of the program. This means that programmers are likely to feel more "in control" of reference counting than other **GARBAGE COLLECTION** techniques. This also means that reference counting is comparatively easy to implement, however, there are a number of issues to consider when using reference counting to manage memory.

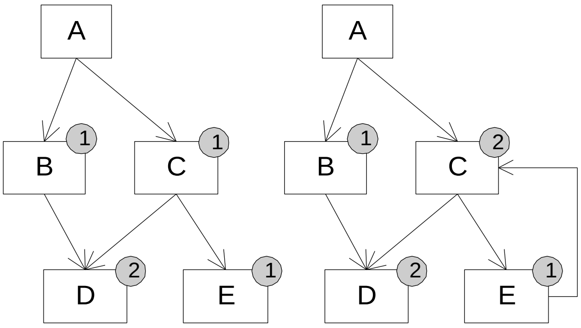
1. Deleting Compound Objects. Shared objects can themselves share other objects. For example, Strap-It-On's bitmap cache could be extended to cache compound figures, where a figure can be made up of a number of bitmaps or other figures (see below, where Figure A includes Figure B).



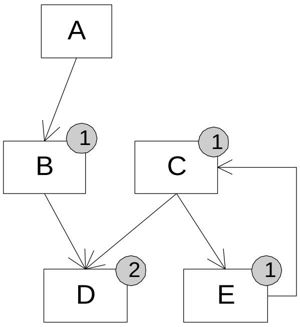
When a reference counted object is deleted, if it refers to other reference counted objects, their reference counts must also be decremented. In the illustration, if Figure A is destroyed, it should decrease the reference count for Figure B.

Freeing reference counted objects is a recursive process: once an object's reference count is zero, it must reduce the reference counts of all the objects to which it refers; if those counts also reach zero, the objects must be deleted recursively. Recursive freeing makes reference counting's performance unpredictable (although long pauses are very rare) and also can require quite a large amount of stack memory. The memory requirements can be alleviated by threading the traversal through objects' existing pointers using pointer reversal (see the **EMBEDDED POINTER** pattern), and the time performance by queuing objects on deletion and recycling them on allocation. This is ‘lazy deletion’ – see Jones and Lins [1996] for more details.

2. Cycles. Objects can form any graph structure and these structures may or may not have cycles (where you can follow references from one object back to itself through the graph). The structure in left illustration is acyclic; in the right illustration, C and E form a cycle.



Reference counting doesn’t work for cycles of shared objects, and such cycles can be quite common: consider doubly linked lists, skip lists, or trees with both upward and downward pointers. If two (or more) objects point to each other, then both of their reference counts can be non-zero, even if there are no other references to the two objects. In the below, objects C and E form a cycle. They will have reference counts of one, even though they should be deleted because they are not accessible from any other object. In fact they would never be deleted by reference counting, even if every other object was removed.



2.1 Breaking Cycles. One way of dealing with cycles is to require programmers to break cycles explicitly, that is, before deleteing the last external reference to a cyclic structure of objects, programmers should overwrite at least one of the references that creates the cycle with `nil`. For example, you could remove the pointer from E to C in the figure above. After this, the object structure no longer contains a cycle, so it can be recycled when the last external reference to it is also removed. This requires *programmer discipline* to remember to nil out the references that cause cycles.

2.2 Garbage Collectors. You can also implement a **GARBAGE COLLECTOR** as a backup to reference counting, because garbage collectors can reclaim cyclic structures. The collection can run periodically or/and whenever reference counting cannot reclaim enough memory. A garbage collector requires more *programmer effort* than reference counting, and computation must be stopped when it runs, costing *processing time* and decreasing *interactive responsiveness* and therefore *usability.*

2.3 Cycle-Aware Reference Counting. Alternatively, there are more complex versions of the basic reference counting algorithm that can handle cyclic structures directly, but they are not often worth the implementation effort [Jones and Lins 1996].

3. Allocating the reference count field

Each reference counted object needs a reference count field which imposes a size overhead. In theory a reference count field needs to be large enough to count references from every other pointer in the system, requiring at least enough space for a full pointer to store the count.

In practice, most objects are pointed to by only a few other objects, so reference counts can be made much smaller, perhaps only one byte. Using a smaller reference count can save a large amount of memory, especially if the system contains a large number of small objects and has a large word size. Smaller refernece counts raise the possibility that the counts can overflow. The usual solution is called *sticky* reference counts: once a count field reaches its maximum value it can never be decreased again. Sticky counts ensure that objects with many references will never be recycled incorrectly, by ensuring they will *never* be collected, at least by reference counting. A backup **GARBAGE COLLECTOR** can correct sticky reference counts and collect once widely shared objects that have since become garbage.

4. Extensions

Because reference counting is a simple but inefficient algorithm, it lends itself to extensions and optimisations.

4.1 Keeping Objects with a Zero Reference Count. Alternatively, if the objects represent external objects and are held in a cache, then it may sometimes make sense to keep objects even if their reference count is zero. This applies if the items are expensive to recreate, and there’s a reasonable chance that they may be needed again. In this case, you use reference counts as a guide in most situations, but you can implement **CAPTAIN OATES** and delete unused objects when the memory cost of keeping them outweighs the cost of recreating them later.

4.2 Finalisation. Unlike more efficient forms of garbage collection, reference counting explicitly deletes unreachable objects. This makes it easy to support *finalisation*, that is, allowing objects to execute special actions just before they are about to be deleted. An objects finalisation action is executed once its reference counts reaches zero but before decrementing the objects it references and before recycling its memory. Finalisation code can increase an object’s reference count, so deletion should only proceed if the reference count is still zero after finalisation.

4.3 Optimisations. Reference counting imposes an overhead on every pointer assignment or copy operation. You can optimise code by avoiding increment and decrement operations when you are *sure* an object will never be deallocated due to a given reference, typically because you have at least one properly reference-counted valid reference to the object. More sophisticated reference counting schemes, such as deferred reference counting [Jones and Lins 1996, Deutsch and Bobrow 1976], can provide the benefits of this optimisation without the difficulties, though with a slightly increased runtime overhead and substantially more programmer effort.

Example

This C++ example implements an object large enough to justify sharing and therefore reference counting. A `ScreenImage` contains a screen display of pixels. We might use it as follows:

```
{
    ScreenImage::Pointer image = new ScreenImage;
    image->SetPixel( 0, 0 );
    // And do other things with the image object...
}
```

When image goes out of scope at the terminating brace, the `ScreenImage` object will be deleted, unless there are other `ScreenImage::Pointer` objects referencing it.

The implementation of `ScreenImage` must have a reference count somewhere. This implementation puts it in a base class, `ReferenceCountedObject`.

```
typedef char Pixel;

class ScreenImage : public ReferenceCountedObject {
    Pixel pixels[SCREEN_WIDTH * SCREEN_HEIGHT];
```

The reference counting pointer template class requires a lot of typing to use; for convenience we define a typedef for it:

```
public:
    typedef ReferenceCountingPointer<ScreenImage> Pointer;
```

And here are a couple of example member functions:

```
    void SetPixel( int i, Pixel p ) { pixels[i] = p; }
    Pixel GetPixel( int i ) { return pixels[i]; }
};
```

1. Implementation of ReferenceCountedObject

The `class ReferenceCountedObject` contains the reference count, and declares member functions to manipulate it. The `DecrementCount` operation can safely delete the object, since it doesn't access its this pointer afterward. Note the virtual destructor, so that deletion invokes the correct destructor for the derived class.

```
class ReferenceCountedObject {
private:
    int referenceCount;
public:
    void IncrementCount() { referenceCount++; }
    void DecrementCount() { if (--referenceCount == 0) delete this; }
protected:
    ReferenceCountedObject() : referenceCount( 0 ) {}
    virtual ~ReferenceCountedObject() { }
};
```

2. Implementation of the smart pointer, ReferenceCountingPointer

This is another example of the C++ Smart Pointer idiom. It uses the pointer operator (`->`) to make an instance have the same semantics as a C++ pointer, and manages the reference counts:

```
template <class T> class ReferenceCountingPointer {
private:
    T* pointer;
    void IncrementCount() { if (pointer) pointer->IncrementCount(); }
    void DecrementCount() { if (pointer) pointer->DecrementCount(); }
```

To keep the reference counts correct, it needs all the 'Canonical Class Form' [Ellis and Stroustrup 1990]: default constructor, copy constructor, assignment operator and destructor:

```
public:
    ReferenceCountingPointer() : pointer( 0 ) {}
    ReferenceCountingPointer( T* p )
        : pointer( p ) { IncrementCount(); }
    ReferenceCountingPointer( const ReferenceCountingPointer<T>& other )
        : pointer( other.pointer ) { IncrementCount(); }
    ~ReferenceCountingPointer() { DecrementCount(); }
```

The assignment operator is particularly complicated, since it may cause the object originally referenced to be deleted. Note how, as always, we have to check for self-assignment and to return a reference to `*this` [Meyers 1992].

```
    const ReferenceCountingPointer<T>&
    operator=( const ReferenceCountingPointer<T>& other ) {
        if (this != &other) {
            DecrementCount();
            pointer = other.pointer;
            IncrementCount();
        }
        return *this;
    }
```

The ‘smart’ operations, though, are simple enough:

```
    T* operator->() const { return pointer; }
    T& operator*() const { return *pointer; }
```

And finally we need a couple more operators if we want to use the smart pointers in STL collections, since some STL implementations require a comparison operator [Austern 1999]:

```
    bool operator<( const ReferenceCountingPointer<T>& other ) const {
        return pointer < other.pointer;
    }
    bool operator==( const ReferenceCountingPointer<T>& other ) const {
        return pointer == other.pointer;
    }
};
```

❖ ❖ ❖

Known Uses

Reference counting is part of the garbage collection implementation provided in some language environments. These implementations are invisible to the programmer, but improve the time performance of memory management by deferring the need for a garbage collection process. The limbo language for programming embedded systems used reference counting, because it doesn’t pause computation, and because it allows external objects (menus and popup windows, for example) to be deleted immediately they are no longer used. [Pike 1997]. Smalltalk-80 and VisualWorks\Smalltalk prior to version 2.5 similarly used reference counting for reasons of interactive performance [Goldberg and Robson 1983; ParcPlace 1994].

Microsoft’s COM framework has a binary API based on Microsoft C++’s VTBL implementation. COM uses reference counting to allow several clients to share a single COM object [Box 1998].

UNIX directory trees provide a good example of a directed acyclic graph. The UNIX `ln` command allows you to create ‘hard links’, alternative names for the same file in different directories. A file is not deleted until there are no hard links left to it. Thus each UNIX low-level file object (`inode`) contains a reference count of the number of links to it, and the `ln` command will fail if you try to use it to create a cycle [Kernighan and Pike 1984].

See also

Modern memory management research has focused on **Garbage Collection** rather than reference counting, to improve system’s overall time performance [Jones and Lins 1996].

The particular implementation we’ve used in the example section is the **Counted Pointer** idiom described in [Buschman et al 1996] and [Coplien 1994].

Garbage Collection

Also Known As: Mark-sweep Garbage Collection, Tracing Garbage Collection.

How do you know when to delete shared objects?

- You are **SHARING** objects in your program
- The shared objects are transient, so their memory has to be recycled when they are no longer needed.
- Overall performance is more important than interactive or real-time responsiveness.
- The structure of the shared objects does form cycles.

You are **SHARING** dynamically allocated objects in your program. The memory occupied by these objects needs to be recycled when they are no longer needed. For example, the Strap-It-On's DailyFreudTimer application implements a personal scheduler and psychoanalyst using artificial intelligence techniques. DailyFreudTimer needs many dynamically allocated objects to record potential time schedules and psychological profiles modelling the way you spend your week. As it runs, DailyFreudTimer continually creates new schedules, evaluates them, and then rejects low-rated schedules in favour of more suitable ones, discarding some (but not all) of the objects it has created so far. The application often needs to run for up to an hour before it finds a schedule which both means you get all you need to done this week, and also that you are in the right psychological state at the right time to perform each task.

Objects are often shared within components, or between multiple components in a system. Determining when shared objects are no longer used by any client can be very difficult, especially if there are a large number of shared objects. Often, too, the structure of those objects forms cycles, making **REFERENCE COUNTING** invalid as an approach.

Therefore: *Identify unreferenced objects, and deallocate them.*

To do garbage collection, you suspend normal processing in the system, and then follow all the object references in the system to identify the objects that are still reachable. Since other objects in the system cannot be referenced now, it follows that they'll never be accessible in future (where could you obtain their references from?). In fact, these unreferenced objects are garbage, so you can deallocate them.

To find all the referenced objects in the system, you'll need to start from all the object references available to the running system. The places to start are called the *root set:*

- Global and static variables,
- Stack variables, and perhaps
- References saved by external libraries

Starting from these, you can traverse all the other *active* objects in your runtime memory space by following all the object references in every object you encounter. If you encounter the same object again, there's no need to examine its references a second time, of course.

There are two common approaches to removing the inactive objects:

- *Mark-sweep Garbage Collectors* visit all the objects in the system, de-allocating the inactive ones.

- *Copying Garbage Collectors* copy the active objects to a different area, **MEMORY DISCARD**ing the inactive ones.

For example, StrapItOn implements mark-sweep collection for its schedule and profile objects in the DayFreudTimer. It keeps a list of every such object, and associates a mark bit each. When DayFreudTimer runs out of memory, it suspends computation and invokes a garbage collection. The collector traces every active object from a set of roots (the main `DayFreudTimerApplication` object), recursively marks the objects it finds, and then sweeps away all unmarked objects.

Consequences

GARBAGE COLLECTION can handle every kind of memory structure. In particular, it can collect structures of objects containing cycles with no special *effort* or *discipline* on behalf of the programmer. There's generally no need for designers to worry about object ownership or deallocation, and this improves *design quality* and thus the *maintainability* of the system. Similarly it reduces the impact of *local* memory-management choices on the *global* system.

GARBAGE COLLECTION does not impose any time overhead on pointer operations, and has negligible memory overhead per allocated object. Overall it is usually more *time efficient* than **REFERENCE COUNTING**, since there is no time overhead during normal processing.

However: Garbage collection can generate big pauses during processing. This can disrupt the *real-time response* of the system, and in User Interface processing tend to impact the *usability* of the system. In most systems it's difficult to *predict* when garbage collection will be required, although special purpose real-time algorithms may be suitable. Garbage objects are collected some time after they become unreachable, so there will always appear to be less free memory available that with other allocation mechanisms.

Compared with **REFERENCE COUNTING**, most garbage collectors will need more free space in the heap (to store garbage objects between collections phases), and will be less efficient when the application's memory runs short. Garbage collectors also have to be able to find the global root objects of the system, to start the mark phase traversal, and also need to be able to find all outgoing references from an object. In contrast, **REFERENCE COUNTING** only needs to track pointer assignments.

Finally, the recursive mark phase of a Mark-sweep collector needs stack space to run, unless pointer reversal techniques are used (see the **EMBEDDED POINTER** pattern).

❖ ❖ ❖

Implementation

Garbage collectors have been used in production systems since the late 1960s, but still people are afraid of them. Why? Perhaps the most important reasons is the illusion of control and efficiency afforded by less sophisticated forms of memory management, such as static allocation, manually deallocated heaps, or stack allocation, especially as many (badly tuned) garbage collectors can impose random pauses in a program's execution. Stack allocation took quite some time to become as accepted as it is today (many FORTRAN and COBOL programmers still shun stack allocation), so perhaps garbage collection will be as slow to make its way into mainstream programming.

In systems with limited memory, garbage collectors have even less appeal than **REFERENCE COUNTING** and other more positive forms of memory management. But if you have complex linked structures then a simple garbage collector will be at least as efficient and reliable as manual deallocation code.

We can only present a brief overview of garbage collection in the space of one pattern. *Garbage Collection,* by Richard Jones with Raphael Lins [1996] is the standard reference on garbage collection, and well worth reading if you are considering implementing a garbage collector.

1. Programming with Garbage Collection

Programming with Garbage Collection is remarkably like programming without garbage collection, except that it is easier, because you don't have to worry about explicitly freeing objects, juggling reference counts, or breaking cycles. It is quite possible to control the lifetimes of objects in a garbage-collected system just as closely as in a manually managed system, following three simple insights:

- If "`new`" is never called, objects are never allocated.
- If objects never become unreachable, they will never be deallocated
- If objects are never allocated or deallocated, the garbage collector should never run.

The first point is probably the most important: if you don't allocate objects you should not need any dynamic memory management. In languages that are habitually garbage collected it can be more difficult to find out when objects are allocated, for example, some libraries will allocate objects willy-nilly when you do not expect them to. Deleting objects in garbage collected systems can be difficult: you must find and break every pointer to the object you wish to delete [Lycklama 1999].

2. Finalisation and Weak References

Certain kinds of object may own, and thus need to release non-memory resources such as file handles, graphics resources or device connections. These objects need to implement *finalisation* methods to do this release.

Finalisation can be quite hard to implement in many garbage collectors. You generally don't want the main sweeping thread to be delayed by calling finalisation, so you have to queue objects for processing by a separate thread. Even if supported well, finalisation tends to be unreliable because the precise time an object is finalised depends purely on when the garbage collector runs.

Some garbage collectors support *weak references,* references that are not traced by the collector. Unlike normal references, an object pointed to by a weak reference will become garbage unless at least one normal reference also points to the object. If the object is deleted, all the weak references are usually automatically replaced with some nil value. Weak references can be useful when implementing caches, since if memory is low, unused cached items will be automatically released.

3. Mark-Sweep Garbage Collection

A mark-sweep garbage collector works by stopping the program, marking all the objects that are in use, and then deleting all the unused objects in a single clean sweep. Mark-sweep garbage collection requires only one mark bit per object in the system. This bit can often be **PACKED** within some other field in the object – perhaps the first pointer field since this bit is only every set during the garbage collection phases; during the main computation this bit is always zero.

When an object is allocated, its mark bit is clear. Computation proceeds normally, without any special actions from the garbage collector: object references can be exchanged, fields can be assigned to, and, if their last reference is assigned to another object or to nil, objects may become inaccessible. When memory is exhausted, however, the main program is paused, and the marking and sweeping phases of the algorithm are executed.

Figure XX shows a system with five objects. Object A is the root of the system, and objects B and D are accessible from it. Objects C and E are not accessible from A, and so strictly are garbage. However they make up a 'cycle', so **REFERENCE COUNTING**, however, could not delete them.

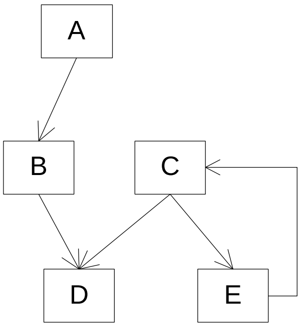


Figure 7: Before Garbage Collection

Mark-sweep garbage collection proceeds in two phases. First, the mark phase recursively traces every inter-object reference in the programming, beginning from a root set, such as all global variables and all variables active on the stack. When the mark phase reaches an unmarked object, it sets the object's mark bit, and recursively visits all the object's children. After the mark phase, every object reachable from the root set is marked: objects unreachable from the root set are unmarked.

The figure below shows the state of the objects after the mark phase. The marked objects are drawn with shaded backgrounds. A, B and D are marked because they are reachable from the root of the system. C and E are not marked.

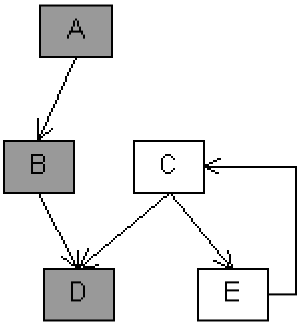


Figure 8: After the mark phase

Second, the sweep phase visits every object on the heap: that is, every object active at the end of the last sweep phase plus every object allocated since then. Whenever the sweep phase finds a marked object, it clears its mark bit (to be ready for the next mark phase). Whenever the sweep phase finds an unmarked object, it recycles the memory used by that object, running the object's finalisation code, if it has any.

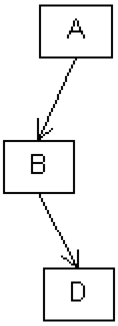


Figure 9: After the sweep phase

Considering the example, the sweep phase visits every object in an arbitrary order, deleting those who do not carry the mark. So after the sweep phase only objects A, B and D remain; the unmarked C and E objects have been deleted.

Mark-sweep works because it explicitly interprets the idea of an active or *live* object. Live objects must be reachable either directly from the root set (global variables, stack variables, and perhaps references saved by external libraries), or via chains of references through objects starting from the root set. The mark phase marks just those objects that meet this criterion, and then the sweep phase eliminates all the garbage objects that do not.

3. Copying Garbage Collectors

Modern workstation garbage collectors are typically based on object copying, rather than mark-sweep, thus implementing a form of **COMPACTION**. A simple copying collector allocates twice as much virtual memory as it needs, in two hemispheres. While the system is running, it uses only one of these hemispheres, called the "fromspace", containing all the existing objects packed together at one end. New objects are allocated following directly after the old objects, simply by incrementing a pointer (just as cheaply as stack allocation). When the fromspace is full, the normal program is paused, and all fromspace objects are traversed recursively, beginning from the roots of the system.

A copying collector's traversal differs from a marking collector's. When a copying collector reaches an object in fromspace for the first time (note that by definition, a reachable object is not garbage) it copies that object into the other hemisphere (the tospace). It then replaces the fromspace object with a forwarding pointer to the tospace version. If the copy phase reaches a forwarding pointer, that pointer must come from an object already copied into the tospace, and the tospace object's field is updated to follow the forwarding pointer into the tospace. Once no more objects can be copied, the tospace and fromspace are (logically) swapped, and the system continues to execute.

More sophisticated copying collectors allocate two hemispheres for only the recently created objects, moving longer-lived objects into a separate memory space [Ungar 1984; Hudson and Moss 1992].

Copying collectors have a number of advantages over Mark-sweep collectors. Copying collectors avoid fragmentation, because the act of copying also compacts all the active objects. More importantly, copying collectors can perform substantially better because they have no sweep phase. The time required to run a copying collector is based on copying all live objects, rather than marking live objects plus sweeping through every object on the heap, allocated and garbage. On the other hand, copying collectors move objects around as the program runs, costing processing time; require more virtual memory than a simpler collector, and are more difficult to implement.

Example

This example illustrates a `CollectableObject` class that supports mark-sweep garbage collection. Every object to garbage collect must derive from `CollectableObject`. The static method `CollectableObject::GarbageCollect` does the **GARBAGE COLLECTION**.

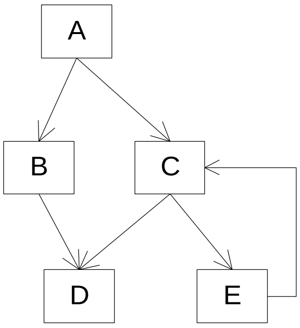
For example, we might implement a Node class with 'left' and 'right' pointers as in Implementation Section "3. Mark-Sweep Garbage Collection". `Node` derives from `CollectableObject`, and every instance must be allocated on the heap:

```
class Node : public CollectableObject {
private:
    Node* left;
    Node* right;
public:
    Node( Node* l = 0, Node* r = 0 ) : left( l ), right( r ) {}
    ~Node() {}
    void SetLinks( Node* l, Node* r ) { left = l; right = r; }
```

The only special functionality Node must provide is a mechanism to allow the Garbage Collector to track all its references. This implementation uses a **TEMPLATE FUNCTION**, `GetReferences`, declared in the base class. `GetReferences` must call `HaveReference` for each of its references [Gamma 1995]. For convenience a call to `HaveReference` with a null pointer has no effect.

```
private:
    void GetReferences() {
        HaveReference( left );
        HaveReference( right );
    }
};
```

Then we can allocate Node objects into structures, and they will be garbage collected. This implementation of Mark-Sweep garbage collection uses the normal 'delete' calls, invoking Node's destructor as normal. For example we might set up the structure in the illustration:



```
    Node* E = new Node( 0, 0 );
    Node* D = new Node( 0, 0 );
    Node* C = new Node( D, E );
    Node* B = new Node( 0, D );
    Node* A = new Node( B, C );
    E->SetLinks( 0, C );
```

An initial Garbage Collection will have no effect:

```
    CollectableObject::GarbageCollect( A );
```

However when we remove the reference from A to C, a second garbage collection will delete C and E:

```
    A->SetLinks( B, 0 );
    CollectableObject::GarbageCollect( A ); // Deletes C and E
```

Finally, we can do a garbage collection will a null root node, to delete all the objects:

```
    CollectableObject::GarbageCollect( 0 );
```

1. Implementation of the Garbage Collector

The garbage collector uses a single class, `CollectableObject`. Every `CollectableObject` maintains one extra field, the `markBit` for use by the collector. Every `CollectableObject` enters a collection `allCollectableObjects`, so that they can be found during the sweep phase.

```
class CollectableObject {
friend int main();
private:
    bool markBit;
public:
    CollectableObject();
    virtual ~CollectableObject();
    virtual void GetReferences() = 0;
    void HaveReference( CollectableObject* referencedObject);
private:
    void DoMark();
```

2. The Garbage Collection Functions

The main Garbage Collector functionality is implemented as static functions and members in the `CollectableObject` class. We use a doubly-linked list, or `deque`, for the collection of all objects, since this is very efficient at adding entries, and at removing them via an iterator:

```
    typedef deque<CollectableObject *> Collection;
    static Collection allCollectableObjects;
public:
    static void GarbageCollect( CollectableObject* rootNode );
private:
    static void MarkPhase(CollectableObject* rootNode);
    static void SweepPhase();
};
```

The main `GarbageCollect` function is simple:

```
/*static*/
void CollectableObject::GarbageCollect( CollectableObject* rootNode ) {
    MarkPhase(rootNode);
    SweepPhase();
}
```

The mark phase calls `DoMark` on the root object, if there is one; this will recursively call `DoMark` on all other active objects in the system:

```
/*static*/
void CollectableObject::MarkPhase(CollectableObject* rootNode)        {
    if (rootNode)
       rootNode->DoMark();
}
```

The sweep phase is quite straightforward. We simply run down every object, and delete them if they are unmarked. If they are marked, they are still in use, so we simply reset the mark bit in preparation for subsequent mark phases:

```
/*static*/
void CollectableObject::SweepPhase() {
    for (Collection::iterator iter = allCollectableObjects.begin();
         iter != allCollectableObjects.end(); )  {
        CollectableObject* object = *iter;
        if (!object->markBit) {
            iter = allCollectableObjects.erase( iter );
            delete object;
        } else {
            object->markBit = false;
            ++iter;
        }
    }
}
```

2. Member functions for CollectableObject:

The constructor for `CollectableObject` initialises the mark bit to 'unmarked', and adds itself to the global collection:

```
CollectableObject::CollectableObject()
     : markBit( false ) {
     (void)allCollectableObjects.push_back(this);
}
```

We also need a virtual destructor, as derived instances will be destructed as instances of `CollectableObject`.

```
/*virtual*/
CollectableObject::~CollectableObject()
{}
```

The `DoMark` function recursively sets the mark bit on this object and objects it references:

```
void CollectableObject::DoMark() {
    if (!markBit) {
        markBit = true;
        GetReferences();
    }
}
```

And similarly the `HaveReference` function is invoked by the `GetReferences` functions in derived classes:

```
void CollectableObject::HaveReference( CollectableObject* referencedObject) {
    if ( referencedObject != NULL)
        referencedObject->DoMark();
}
```

A more robust implementation would replace the recursion in this example with iteration, to avoid the problems of stack overflow. A more efficient implementation might use **POOLED ALLOCATION** of the `CollectableObjects`, or might use an **EMBEDDED POINTER** to implement the `allCollectableObjects` collection.

❖ ❖ ❖

Known Uses

Any dynamic language with real pointers needs some form of garbage collection — Lisp, Smalltalk, Modula-3,and Java are just some of the best-known garbage collected languages. Garbage collection was originally specified as part of Ada, although this was subsequently deferred, and has been implemented many times for C++, and even for C [Jones and Lins 1996].

Mark-sweep garbage collectors are often used as a backup to reference counting systems, as in some implementations of Smalltalk, Java, and Inferno [Goldberg and Robson 1983, Pike 1997]. The Mark-sweep collector is executed periodically or when memory is low, to collect cycles and objects with many incoming references that would be missed by reference counting alone.

See Also

REFERENCE COUNTING is an alternative to this pattern that imposes a high overhead on every pointer assignment, and cannot collect cycles of references.

Systems that make heavy use of **SHARING** may benefit from some form of Garbage Collection. Garbage Collection can also be used to unload **PACKAGES** from Secondary Storage automatically when they are no longer required.

Garbage Collection [Jones and Lins 1996] is a very comprehensive survey of a complex field. Richard Jones' garbage collection web page [Jones 2000] and the *Memory Management*

Reference Page [Xanalys 2000] contain up-to-date information about garbage collection. Paul Wilson [1994] has also written a critical overview of garbage collection techniques.

Appendix: A Discussion of Forces

Version 04/12/99 17:23 - 37

How do you find the patterns relevant to your specific system?

If you're working on a project, the chances are you're already asking yourself "Which of the patterns in this book might I apply to my project?"

No serious software product ever had (or will have) as its mission purely to save memory. If it did, the solution would be simple: write no code! But all real software has other aims and other constraints. In the words of the patterns community [Coplien 1996, Vlissides 1998] you have other *forces* acting on you and your project. Each pattern provides a solution to a problem in the context of the forces acting on you as you make the decision.

A pattern's forces capture the problem's considerations and the pattern's consequences, to help you to decide when to use that pattern rather than another. Each pattern's initial problem statement identifies the major force driving that pattern, discusses of other forces affecting the solution. Then the pattern's Consequences section identifies how the pattern affects the configuration of the forces.

Some forces may be *resolved* by the pattern, that is, the pattern solves that aspect of the problem, and these forces form a pattern's positive benefits. Other forces may be *exposed* by the pattern, that is, applying the pattern causes additional problems, and these forces form a pattern's liabilities. You can then use further patterns to resolve the exposed forces, patterns that in their turn expose further forces, and so on.

This appendix answers the question above by asking in return "What other constraints and requirements do you have?", or,

What are your most important forces?

Identifying your forces can lead you to a set of patterns that you may – or may not – choose to use in your system.

Forces in this book

For all the patterns in this book the most important force is the software's memory requirements. But there are other important forces. The list below summarises most of the forces we've identified. In each case, a "yes" answer to the question generally means a benefit to the project.

The forces are in three categories:

- Non-functional requirements
- Architectural impact on the system
- Effect on the development process

The following tables give a brief summary of each force. The rest of this chapter examines each force in more detail, exploring the patterns that resolve and that expose each one.

Memory Requirements	Does the pattern reduce the overall memory use of the system?
Memory Predictability	Does the pattern make the memory requirements predictable? This is particularly important for real-time applications, where behaviour must be predictable.
Scalability	Does the pattern increase the range of memory sizes in which the program can

	function?
Usability	Does the pattern tend to make the easier for users to operate the system?
Time Performance	Does the pattern tend to improve the run-time speed of the system
Real-time Response	Does the pattern support fixed and predictable maximum response times?
Start-up Time	Does the pattern reduce the time between a request to start the system and its beginning to run?
Hardware and O/S Cost	Does the pattern reduce the hardware or operating system support required by the system?
Power Consumption	Does the pattern reduce the power consumption of the resulting system?
Security	Does the pattern make the system more secure against unauthorised access or viruses?

Table 1: Forces Expressing Non-functional Requirements

Memory waste	Does the pattern reduce the amount of memory in use but serving no purpose?
Fragmentation	Does the pattern reduce the amount of memory lost through fragmentation?
Local vs. Global	Does the pattern tend to help encapsulate different parts of the application, keeping them more independent of each other?

Table 2: Forces Expressing Architectural Impact

Programmer Effort	Does the pattern reduce the total programmer effort to produce a given system?
Programmer Discipline	Does the pattern remove restrictions on programming style, so that programmers can pay less attention to detail in some aspects of programming?
Maintainability and Design Quality	Does the pattern encourage better design quality? Will it be easier to make changes to the system later on?
Testing cost	Does the pattern reduce the total testing effort for a typical project?
Legal restrictions	Will implementing the pattern be free from legal restrictions or licensing costs?

Table 3: Forces Representing the Effect on the Development Process

The table in the back cover summarises a selection of the most important of these forces, illustrating how they apply to each of the patterns in the language. Each cell contains '☺' if the pattern normally has a beneficial effect in that respect (a "yes" answer to the question in the table above), '☹' if the pattern's effect is detrimental. A '😐' indicates that the pattern usually has an effect, but that whether positive or negative depends on circumstances.

The remainder of this chapter examines each force in more detail. For each force, we indicate the patterns that best resolve it, and the patterns that regrettably often expose it.

Forces related to non-functional requirements

The forces in this section concern the delivered system. How will it behave? Will it satisfy the clients' needs by being sufficiently reliable, fast, helpful and long-lived?

Memory Requirements

Does the pattern reduce the overall memory use of the system?

The single most important force in designing systems for limited memory is, unsurprisingly enough, the memory requirements of the resulting system – the amount of memory the system requires to do its job.

Patterns that resolve this force

- All the patterns in this book (Chapters N to M) resolve this force in one way or another.

Memory Predictability

Does the pattern make the memory requirements predictable?

Minimising a program's absolute memory requirements is all very well, but often it is more useful to know in advance whether a given program design can cope with its expected load, precisely what its maximum load will be, and whether it will exhaust the memory available to it. Often, increased memory requirements or reduced program capacity are better than random program crashes. In order to be able to determine that a program can support its intended load, or that it will not run out of memory and crash, you need to be able to audit the program to predict the amount of memory that it will require at runtime.

Predictability is particularly important for systems that must run unattended, where behaviour must be guaranteed and reliability is essential. In particular, life-critical systems must have predictable requirements. See, for example, the discussion in the **FIXED ALLOCATION** pattern.

Patterns that resolve this force

- **FIXED ALLOCATION** ensures your memory requirements do not change while the code is running. You can calculate memory requirements exactly during the design phase.
- **EMBEDDED POINTERS** allow you to calculate memory requirements for linked collections of objects easily.
- **PARTIAL FAILURE** permits pre-defined behaviour when memory runs out.
- A **MEMORY LIMIT** puts a constraint on the amount of memory used by any particular component.
- **DATA FILES** and **APPLICATION SWITCHING** handle only a certain amount of data at a time, potentially removing the chance of memory exhaustion.
- **EXHAUSTION TEST** verifies the system's behaviour on heap exhaustion.
- **CAPTAIN OATES** releases memory from lower priority tasks making it possible to have high priority tasks that complete reliably.

Patterns that expose this force

- **VARIABLE ALLOCATION** encourages a component to use unpredictable amounts of memory.
- **GARBAGE COLLECTION** makes it more difficult to determine in advance precisely when unused memory will be returned to the system.
- **COMPRESSION** (especially **ADAPTIVE COMPRESSION**) reduces the absolute memory requirements for storing the compressed data, but by an unpredictable amount.

- **COPY-ON-WRITE** obscures the amount of memory required for an object — memory only needs to be allocated when an object is first modified.
- **MULTIPLE REPRESENTATIONS** means that the amount of memory allocated to store an object can vary considerably, and in some uses, dynamically.

Scalability

Does the pattern increase the range of memory sizes in which the program can function?

Moore's Law [1997] states that hardware capacity increases exponentially, which means than the amount of memory available at any given cost decreases greatly over time. As a result, the amount of memory available tends to increase over time (or, rarely, the same devices can be sold more cheaply) [Smith 1999]. So long-lasting software needs to be *scalable* to take advantage of more memory if it is available.

Furthermore different users may have different amounts of money to spend, and different perceptions of the importance of performance and additional functionality. Such user choices require scalable software too.

Patterns that resolve this force

- **VARIABLE ALLOCATION** adjusts the memory allocated to a structure to fit the number of objects the structure actually contains, limited only by the available memory.
- **PAGING** and other **SECONDARY STORAGE** patterns allow a program access to more apparent RAM, by storing temporarily unneeded information on secondary storage. Adding more RAM improves the time performance of the system without affecting the functionality.
- **MULTIPLE REPRESENTATIONS** allows the system to size its objects according to the available memory.

Patterns that expose this force

- Designing a **SMALL ARCHITECTURE** requires you to make components responsible for their own memory use and accepting this responsibility can sometimes increase the complexity and decrease the performance of each component. The components bear these costs even when more memory become available.
- **FIXED ALLOCATION** (and **POOLED ALLOCATION** from a fixed sized pool) require you to commit to the size of a data structure early, often before the program is run or before the data structure is used.
- **SMALL DATA STRUCTURES**, especially **PACKED DATA**, trade time performance to reduce memory requirements. It can be hard to redesign data structures to increase performance if more memory is available.

Usability

Does the pattern tend to make the easier for users to operate the system?

Designing systems that use limited amounts of memory requires many compromises, and often these reduce the usability — ease of use, ease of learning, and user's speed, reliability and satisfaction — of the resulting system [Shneiderman 1997].

Usability is a complex, multifaceted concern, and we address it in this book only insofar as the system usability is directly affected by the memory constraints.

Patterns that resolve this force

- **PARTIAL FAILURE** ensures the system can continue to operate in low memory conditions.

- CAPTAIN OATES allows the system to continue to support users' most important tasks by sacrificing less important tasks.
- Other ARCHITECTURAL PATTERNS can make help make a system more consistent and reliable, and so more usable.
- Using SMALL DATA STRUCTURES can increase the amount of information a program can store and manipulate, to users' direct benefit.
- PAGING makes the system's memory appear limitless, so users do not need to be concerned about running out of memory.

Patterns that expose this force

- SECONDARY STORAGE patterns make users aware of different kinds of memory.
- APPLICATION SWITCHING makes users responsible for changing between separate 'applications', even though the may not see any reason for the separation of the system.
- FIXED ALLOCATION can make a system's memory capacity (or lack of it) directly and painfully obvious to the system's users.

Time Performance

Does the pattern tend to improve the run-time speed of the system?

Being small is not enough; your programs usually have to be fast as well. Even where execution speed isn't an absolute requirement, there'll always be someone, somewhere, who wants it faster.

Patterns that resolve this force

- FIXED ALLOCATION can assign fixed memory locations as the program is compiled, so they can be accessed quickly using absolute addressing.
- MEMORY DISCARD and POOLED ALLOCATION support fast allocation and de-allocation.
- MULTIPLE REPRESENTATIONS allows you to have memory-intensive implementation of some objects to give fast performance without incurring this overhead for every instance.
- EMBEDDED POINTERS can support fast traversal and update operations on link-based collections of objects.
- Most GARBAGE COLLECTION algorithms do not impose any overhead for memory management on pointer manipulations.

Patterns that expose this force

- VARIABLE ALLOCATION and deallocation cost processing time.
- COMPRESSION, especially ADAPTIVE COMPRESSION, requires processing time to convert objects from smaller compressed representations to larger computable representations.
- COMPACTION similarly requires processing time to move objects around in memory.
- Most SECONDARY STORAGE patterns, especially PAGING, uses slower secondary storage in place of faster primary storage.
- REFERENCE COUNTING requires up to two reference count manipulations for *every* pointer manipulation.
- PACKED DATA is typically slower to access than unpacked data.

- **SMALL INTERFACES** pass small amounts of data incrementally, which can be much slower than passing data in bulk using large buffer structures.
- **CAPTAIN OATES** can take time to shut down tasks or components.
- Indirect memory accesses via **HOOKS** can reduce the system's time performance.

Real-time Response

Does the pattern support fixed and predictable maximum response times?

Just as predictability of memory use — and the resulting stability, reliability, and confidence in a program's performance — can be as important or more important than the program's absolute memory requirements, so the predictability of a program's time performance can be more important than its absolute speed.

This is particularly important dealing with embedded systems and communications drivers, which may have real-world deadlines for their response to external stimuli.

Patterns that resolve this force

- **FIXED ALLOCATION** , **MEMORY DISCARD,** and **POOLED ALLOCATION** usually have a predictable worst case performance.
- **EMBEDDED POINTERS** can allow constant-time traversals between objects in linked data structures.
- **SMALL INTERFACES** ensure fixed amounts of data can be passed between components in fixed amounts of time
- **SEQUENCE COMPRESSION** can compress and decompress simple data streams in fixed amounts of time per item.
- **REFERENCE COUNTING** amortises memory management overheads at every pointer manipulation, and so does not require random pauses during a system's execution.

Patterns that expose this force

- **VARIABLE ALLOCATION** can require unpredictable amounts of time
- The time required by most **ADAPTIVE COMPRESSION** algorithms is dependent on the content of the information it is compressing.
- Some implementations of **MEMORY COMPACTION** may sporadically require a large amount of time to compact memory. If compaction is invoked whenever a standard allocator cannot allocate enough contiguous memory, then allocation will take varying amounts of time, and this performance will degrade as the free space decreases.
- Many **SECONDARY STORAGE** patterns take extra time (randomly) to access secondary storage devices.
- **COPY-ON-WRITE** requires time to make copies of objects being written to.

Start-up Time

Does the pattern reduce the time between a request to start the system and its beginning to run?

Start-up time is another force that is related to execution time, but clearly independent of both absolute performance and real-time response. For embedded systems, and even more crucially for PDAs and mobile phones, the time between pressing the 'on' switch and accomplishing useful work is vital to the usability and marketability of the system.

Patterns that resolve this force

- **PACKAGES** and **APPLICATION SWITCHING** allow a main module to load and start executing quickly; other modules load and execute later.
- **READ ONLY MEMORY** allows the CPU to access program code and resources immediately a program starts, without loading from secondary storage
- **SHARING** of executable code allows a new program to start up quickly if the code is already running elsewhere; **SHARING** of data reduces initial allocation times.
- **MEMORY DISCARD** can allocate objects quickly at the start of the program.
- **VARIABLE ALLOCATION** defers allocation of objects until they are needed.
- **COPY-ON-WRITE** avoids an initial need to allocate space and copy all objects that might possibly change; copying happens later as and when necessary.

Patterns that expose this force

- **FIXED ALLOCATION** and **POOLED ALLOCATION** require time to initialise objects or pools before the program begins running.
- **COMPRESSION** can require time to uncompress code and data before execution.
- Initialising from **DATA FILES** and **RESOURCE FILES** on **SECONDARY STORAGE** all takes time.

Hardware and Operating System Cost

Does the pattern reduce the hardware or operating system support required by the system?

Hardware or operating systems can provide facilities to directly support some of the patterns we have described here. Obviously, it makes sense to use these facilities when they are provided, if they address a need in your design. Without such support, you may be better off choosing an alternative pattern rather than expending the effort required emulating it yourself.

Patterns that expose this force

- **CAPTAIN OATES** needs a mechanism for individual tasks within a system to determine the system's global memory use, and ideally a means to signal memory-low conditions to all programs.
- **GARBAGE COLLECTION** is often provided in the virtual machines or interpreters for modern programming languages, or as libraries for languages like C++.
- **RESOURCE FILES** and **PACKAGES** need to load binary data such as executable files or font and icon files into running programs. This is easiest if implemented in the operating system.
- **PAGING** is much more efficient if it uses the page and segment tables of your processor, and in practice this requires operating system support.
- Similarly, **COPY-ON-WRITE** is implemented most efficiently if it can use hardware page table write protection faults.

Power Consumption

Does the pattern reduce the power consumption of the resulting system?

Battery-powered systems, such as hand-helds, palmtop computers and mobile phones, need to be very careful of their power consumption. You can reduce power consumption by avoiding polling, avoiding long computations, and by switching off power-consuming peripherals.

Patterns that resolve this force

- **READ-ONLY MEMORY** often requires no power to keep valid.

Patterns that exposethis force

- **SECONDARY STORAGE** devices, such as battery-backed RAM and disk drives, consume power when they are accessed.
- **COMPRESSION** algorithms need CPU power to compress and uncompress data.
- **PAGING** is particularly bad, since it can require secondary storage devices to be running continuously on battery power.

Security

Does the pattern make the system more secure against unauthorised access or viruses?

Security is increasingly important, with the advent of the Internet and private information being stored on insecure desktop or palmtop computers. As with forces like *memory predictablity* and *real-time response*, its generally not enough to claim that a system is secure, you also need to be able to audit the implementation of the system to see how it is built.

Patterns that resolve this force

- Information stored in **READ-ONLY MEMORY** cannot generally be changed so should remain sacrosanct.

Patterns that expose this force

- **SECONDARY STORAGE** devices, especially if used by **PAGING,** may store unsecured copies of sensitive information from main memory.
- **PACKAGES** can allow components of the system to be replaced or extended by other, insecure or hostile, versions.
- **HOOKS** allow nominally read-only code and data to be changed, allowing the introduction of viruses.

Architectural Impact

We can identify a different set of forces that affect the delivered system less directly – they're visible to the developers more than to the end-users.

Memory Waste

Does the pattern reduce the amount of memory in use but serving no purpose?

Some design approaches waste memory. For example low priority tasks may keep unnecessary caches; fully featured, large, objects may be allocated where smaller and more Spartan versions would do; and allocated objects may sit around performing no useful purpose.

Clearly it's generally good to avoid such wasted memory, even if in some cases it's worth accepting the penalty in return for other benefits.

Patterns that resolve this force

- **MULTIPLE REPRESENTATIONS** avoids unnecessarily memory-intensive instances of objects when a more limited representation will do the job.
- **SHARING** and **COPY-ON-WRITE** can prevent redundant copies of objects.
- **PACKED DATA** reduces the amount of memory required by data strucutres.

Patterns that expose this force

- **FIXED ALLOCATION** and **POOLED ALLOCATION** tend to leave unused objects allocated.

- Objects allocated by **VARIABLE ALLOCATION** can become memory leaks if they are no longer used and have not been explicit deleted.
- **REFERENCE COUNTING** and **GARBAGE COLLECTION** can also have memory leaks – objects that are no longer in use, but are still reachable from the system root.
- **MEMORY LIMITS** can waste memory by preventing components from using otherwise unallocated memory.
- **ADAPTIVE COMPRESSION** often needs to uncompress large portions of data into memory, even when much of it isn't required.

Fragmentation

Does the pattern reduce the amount of fragmentation?

Fragmentation causes memory to be unusable because of the behaviour of memory allocators, resulting in memory that is allocated but can *never* be used (internal fragmentation) or that has been freed but can *never* be reallocated (external fragmentation). See ***MEMORY ALLOCATION*** for a full discussion of fragmentation.

Patterns that resolve this force

- **MEMORY COMPACTION** moves allocated objects in memory to prevent external fragmentation.
- **FIXED ALLOCATION** and **POOLED ALLOCATION** avoid allocating variable-sized objects, also avoiding external fragmentation.
- **MEMORY DISCARD** avoids fragmentation – stack allocation has no fragmentation waste and discarding a heap discards the fragmentation along with it.
- **APPLICATION SWITCHING** can avoid fragmentation by discarding all the memory allocated by an application and starting over again.

Patterns that expose this force

- **VARIABLE ALLOCATION** supports dynamic allocation of variable sized objects, causing fragmentation dependent on your memory allocation algorithm.
- **FIXED ALLOCATION** and **POOLED ALLOCATION** generate internal fragmentation when they allocate variable-sized objects.

Local vs. Global Coupling

Does the pattern tend to help encapsulate different parts of the application, keeping them independent of each other?

Some programming concerns can be merely a local concern. For example stack memory is local to the method that allocates it. The amount of memory is determined directly by that method and affects only invocations of that method and any method called from it.

In contrast the amount of memory occupied by heap objects is a global concern. Methods can allocate many objects that exist after the method returns, and so the amount of memory allocated by such a method can affect the system globally. Some patterns can affect the balance between local and global concerns in a program, requiring local mechanisms to achieve global results or, on the other hand, imposing global costs to produce a local effect.

Patterns that resolve this force

- **SMALL ARCHITECTURE** and **SMALL INTERFACES** describe how program modules and their memory consumption can be kept strictly local.

- **PACKED DATA** and other **SMALL DATA STRUCTURES** can be applied to a local design for each structure, allowing redesign without affecting other components.
- **MULTIPLE REPRESENTATIONS** can change data structure representations dynamically, without affecting the rest of the program.
- **POOLED ALLOCATION** and **MEMORY LIMITS** can localise the effects of dynamic memory allocation to within a particular module.
- **MEMORY DISCARD** allows a set of local objects to be deleted simultaneously.
- **PAGING** allows most system code to ignore issues of secondary storage.
- **REFERENCE COUNTING** and **GARBAGE COLLECTION** allow decisions about deleting objects shared globally to also be made globally.

Patterns that expose this force

- **PARTIAL FAILURE** and **CAPTAIN OATES** require local support within programs to provide support for graceful degradation globally throughout the program.
- **VARIABLE ALLOCATION** shares memory between different components over time, so the local memory used by one component affects the global memory available for others.
- **SHARING** potentially introduces coupling between every client object sharing a given item.
- **EMBEDDED POINTERS** require local support within objects so that they can be members external (global) collections.

Development Process

The following forces concern the development process. How easy will it be to produce the system, to test it, to maintain it? Will you get management problems with individual motivation, with team co-ordination, or with the legal implications of using the techniques?

Programmer Effort

Does the pattern reduce the total programmer effort to produce a given system?

The cost of programmer time far exceeds the cost of processor time for all but the most expensive supercomputers (and for all except the cheapest programmers) – unless the software is very widely used. Some patterns tend to increase implementation effort, while others can reduce it.

Patterns that resolve this force

- **VARIABLE ALLOCATION** doesn't require you to predict memory requirements in advance.
- **GARBAGE COLLECTION** means that you don't have to worry about keeping track of object lifetimes.
- **HOOKS** allow you to customise code without having to rewrite it.
- **MEMORY DISCARD** makes it easy to deallocate objects.
- **PAGING** transparently uses secondary storage as extra memory.

Patterns that expose this force

- **PARTIAL FAILURE** and **CAPTAIN OATES** can require you to implement large amounts of checking and exception handling code.

- **COMPRESSION** patterns (especially **ADAPTIVE** Compression) may require you to implement compression algorithms or learn library interfaces.
- **COMPACTION** requires effort to implement data structures that can move in memory.
- Most **SECONDARY STORAGE** patterns require programmers to move objects explicitly between primary and secondary storage.
- **SMALL DATA STRUCTURES** can require you to reimplement parts of your program to optimise its memory use.
- **EMBEDDED POINTERS** can require you to rewrite common collection operations for every new collection of objects.

Programmer Discipline

Does the pattern remove restrictions on programming style, so that programmers can pay less attention to detail in some aspects of programming?

Some patterns depend upon you to pay constant attention to small points of detail, and carefully follow style rules and coding conventions. Following these rules requires a high level of concentration, and makes it more likely you will make mistakes. Of course once learned the rules do not greatly reduce your productivity, or increase the overall effort you will need to make.

Some patterns (like **PAGING**) reduce programmer discipline by using automatic mechanisms, but require effort to implement those mechanisms; others, like **REFERENCE COUNTING,** require discipline to use but do not take much effort to implement.

Patterns that resolve this force

- **GARBAGE COLLECTION** automatically determines which objects are no longer in use and so can be deleted, avoiding the need to track object ownership.
- **PAGING** uses secondary storage to increase the apparent size of main memory transparently, avoiding in many cases the discipline of **PARTIAL FAILURE**.
- **COPY-ON-WRITE** means that clients can safely modify an object regardless of whether it is **SHARED** or in **READ-ONLY MEMORY**.

Patterns that expose this force

- A **SMALL ARCHITECTURE** requires discipline to keep system and component wide policies about memory use, and to use **READ-ONLY MEMORY** and **RESOURCE FILES** as appropriate.
- **PARTIAL FAILURE** requires you to cater for memory exhaustion in almost all the code you write.
- **CAPTAIN OATES** may require you to implement ‘good citizen’ code that doesn’t add to your current component’s apparent functionality.
- **REFERENCE COUNTING** and **COMPACTION** may require you to use special handle objects to access objects indirectly.
- **POOLED ALLOCATION** and **MEMORY DISCARD** require careful attention to the correct allocation, use, and deallocation of objects, to avoid dangling pointers or memory leaks.
- You have to include **HOOKS** into the design and implementation of your components so that later users can customise each component to suit their requirements.

- Using **COMPRESSION** routinely (say for all string literals) makes programming languages literal facilities much harder to use.
- **EMBEDDED POINTERS** require care when objects can belong to multiple collections.

Design Quality and Maintainability

Does the pattern encourage better design quality? Will it be easier to make changes to the system later on?

Some design and programming techniques make it easier for later developers to read, understood, and subsequently change the system.

Patterns that resolve this force

- Taking the time to design a **SMALL ARCHITECTURE** and **SMALL DATA STRUCTURES** increases the quality of the resulting system.
- **HOOKS** allow a program's code to be extended or modified by end users or third parties, even if the code is stored in **READ ONLY MEMORY**.
- **PARTIAL FAILURE** supports other failure modes than memory exhaustion, such as network faults and disk errors.
- **SMALL INTERFACES** reduce coupling between program components.
- **MULTIPLE REPRESENTATIONS** allow objects implementations to change to suit the way they are used.
- **SHARING** reduces duplication between (and within) the components of a system.
- **RESOURCE FILES** and **PACKAGES** allow a program's resources — literal strings, error messages, screen designs, and even executable components — to change without affecting the program's code.
- **REFERENCE COUNTING**, **GARBAGE COLLECTION** and **PAGING** allow you to make global strategic decisions about deleting objects or using secondary storage.
- **APPLICATION SWITCHING** based on scripts can be very easily modified.

Patterns that expose this force

- **FIXED ALLOCATION'S** fixed structures can make it more difficult to change the volume of data that can be processed by the program.
- Code and data stored in **READ ONLY MEMORY** can be very difficult to change or maintain.
- **APPLICATION SWITCHING** can reduce a system's design quality when it forces you to split functionality into executables in arbitrary ways.
- **PACKED DATA** structures can be hard to port to different environments or machines.
- Collections based on **EMBEDDED POINTERS** are hard to reuse in different contexts.

Testing cost

Does the pattern reduce the total testing effort for a typical project?

It's not enough just to code up your program; you also have to make sure it works reliably (unless your product commands a monopoly in the market!). If you care about reliability, choose patterns that decrease the cost of testing the program, so that you can test more often and more thoroughly.

Patterns that resolve this force

- **FIXED ALLOCATIONS** are always the same size independent of program loading, so they always run out of capacity at the same time. This simplifies exhaustion testing.
- **READ-ONLY MEMORY** is easier to test because its contents are unable to change.
- **DATA FILES** and **RESOURCE FILES** help testing because you can use versions of the files to set up different test scenarios.

Patterns that expose this force

- **VARIABLE ALLOCATION, POOLED ALLOCATION, MEMORY DISCARD, MEMORY LIMIT,** and **MULTIPLE REPRESENTATIONS** require testing to check changes their in sizes and representations.
- **PARTIAL FAILURE** and **CAPTAIN OATES** have to be tested to check their behaviour both when memory is scarce, but also when it is abundant.
- **COMPRESSION** implementations should be tested to see that they perform in exactly the same way as implementations that don't use compression.
- Any kind of **SHARING** (including **HOOKS** and **COPY-ON-WRITE**) has to be exhaustively tested from the perspective of all clients of any shared objects, and also for any potential interactions implicitly communicated between clients via the shared object.

Legal restrictions

Will implementing the pattern be free from legal restrictions or licensing costs?

Some programming techniques are subject to legal restrictions such as copyrights and patents. Choosing to use these techniques may require you to pay license fees to the owner of the copyright or patent. Yet using third party software or well-known techniques is a crucial component of good practice in software development — indeed, making existing practices better known is the aim of this book.

Alternatively, some free software (aka Open Source) licences, notably the GNU General Public License, may require you to release some or all of your software with similar licence conditions. However the open source community is actively working to develop alternatives to proprietary techniques that can often be incorporated into all types of software without imposing onerous conditions onto the software development.

Patterns that expose this force

- **FILE COMPRESSION** algorithms from third parties are often subject to copyright or patent restrictions.
- **GARBAGE COLLECTION** and sophisticated **VARIABLE ALLOCATION** libraries usually come as proprietary software.

Pattern Summaries – Small Memory Software

Major Technique: Small Architecture

How can you manage memory use across a whole system? Make every component responsible for its own memory use.

Memory Limit How can you share out memory between multiple competing components? Set limits for each component and fail allocations that exceed the limits.

Small Interfaces How can you reduce the memory overheads of component interfaces? Design interfaces so that clients control data transfer.

Captain Oates How can you fulfil the most important demands for memory? Sacrifice memory used by less vital components rather than fail more important tasks.

Read-Only Memory What can you do with read-only code and data? Store read-only code and data in read-only memory.

Hooks How can you change information in read-only storage? Access read-only information through hooks in writable storage and change the hooks to give the illusion of changing the information.

Major Technique: Secondary Storage

What can you do when you have run out of primary storage? Use secondary storage as extra memory at runtime.

Application Switching How can you reduce the memory requirements of a system that provides many different functions? Split your system into independent executables, and run only one at a time.

Data File Pattern What can you do when your data doesn't fit into main memory? Process the data a little at a time and keep the rest on secondary storage.

Resource Files Pattern How can you manage lots of configuration data? Keep configuration data on secondary storage, and load and discard each item as necessary.

Packages How can you manage a large program with lots of optional pieces? Split the program into packages, and load each package only when it's needed.

Paging Pattern How can you provide the illusion of infinite memory? Keep a system's code and data on secondary storage, and move them to and from main memory as required.

Major Technique: Compression

How can you fit a quart of data into a pint pot of memory? Use a compressed representation to reduce the memory required.

Table Compression Pattern How do you compress many short strings? Encode each element in a variable number of bits so that the more common elements require fewer bits.

Difference Coding Pattern How can you reduce the memory used by sequences of data? Represent sequences according to the differences between each item.

Adaptive Compression Pattern How can you reduce the memory needed to store a large amount of bulk data? Use an adaptive compression algorithm.

Major Technique: Small Data Structures

How can you reduce the memory needed for your data? Choose the smallest structure that supports the operations you need.

Packed Data How can you reduce the memory needed to store a data structure? Pack data items within the structure so that they occupy the minimum space.

Sharing How can you avoid multiple copies of the same information? Store the information once, and share it everywhere it is needed.

Copy-on-Write How can you change a shared object without affecting its other clients? Share the object until you need to change it, then copy it and use the copy in future.

Embedded Pointer How can you reduce the space used by a collection of objects? Embed the pointers maintaining the collection into each object.

Multiple Representations How can you support several different implementations of an object? Make each implementation satisfy a common interface.

Major Technique: Memory Allocation

How do you allocate memory to store your data structures? Choose the simplest allocation technique that meets your need.

Fixed Allocation How can you ensure you will never run out of memory? Pre-allocate objects during initialisation.

Variable Allocation How can you avoid unused empty space? Allocate and deallocate variable-sized objects as and when you need them.

Memory Discard How can you allocate temporary objects? Allocate objects from a temporary workspace and discard it on completion.

Pooled Allocation How can you allocate a large number of similar objects? Pre-allocate a pool of objects, and recycle unused objects.

Compaction How do you recover memory lost to fragmentation? Move objects in memory to remove unused space between them.

Reference Counting How do you know when to delete a shared object? Keep a count of the references to each shared object, and delete each object when its count is zero.

Garbage Collection How do you know when to delete shared objects? Identify unreferenced objects, and deallocate them.

Thinking Small
The Processes for Creating Small Memory Software

Abstract:

This paper describes some process patterns for teams to follow when creating software to run in limited memory.

It is a draft version of a chapter to add to the authors' book Small Memory Software, and follows the structure of other chapters in that book.

Major Technique: Thinking Small

A.k.a Small methodology, 'Real' programming.

How should you approach a small system?

- You're developing a system that will be memory-constrained.
- There are many competing constraints to satisfy
- If different developers take different views on which things to optimise, they will produce an inconsistent system that satisfies *none* of the constraints.

You're working on a project and you suspect there will be resource limitations in the target system. For example, the developers of the 'Super-spy 007' version for the Strap-it-On wrist-mounted computer face a system with only 200 Kb of RAM and 2 Mb ROM. How are they to adjudicate the demands of the voice recognition software, the vocabularies and the software-based radio, to make it a secret agent's dream toy? Should they store the vocabularies in ROM to save RAM space, or keep them in RAM to allow them to change from Russian to Arabic on the fly? What, in short, is important in their system, and what is less so?

In many projects it's clear from the outset that the development team will have to spend at least some time and effort satisfying the system's memory limitations. You have to cut your coat to fit your cloth. Yet if the team just spends lots of effort optimising everything to work in very limited memory, they'll waste a lot of time or maybe produce a product that could have been much better. Worse still the product may fail to work at all because they have been optimising the wrong thing.

For example, any of the following facilities may be limited:

- Heap (RAM) space for the whole system
- Heap space for individual processes (if the maximum heap size of a process is fixed)
- Process stack size
- Secondary storage use
- ROM space (for programs that execute from ROM)

Optimising one of these will often be at a cost from one of the others. In addition techniques that optimise memory use will tend to compromise time-performance, usability or both.

In any system the architects will have to moderate the demands of different components in the system against each other. That is a big and highly sensitive task.

Software programmers tend to take their design decisions seriously, so capricious decisions can cause friction or worse within a development team.

You might hope to use clever techniques to defer the key decisions about these priorities until later in the project, when you'll know more about the implementation. But in practice many of the most important strategic decisions cannot be deferred, as they pervade the entire system and provide a framework for later decisions. Such strategic decisions are reflected, for example, in the interfaces between components, in the trade-off between ROM and RAM, and in the question of whether or not to use Partial Failure in components.

Design decisions about the trade-offs based on just individual designers' foibles, on gut feel or on who shouts loudest will lead neither to consistent successful designs, nor to a harmonious development. You'll need a more objective approach.

Therefore: *Devise a memory strategy for the entire project.*

First draw up a crude **MEMORY BUDGET** of the likely available resources in each of the categories above. If the figures are flexible (for example, if the system is to run on standard PCs with variable configurations and other applications), then estimate or negotiate target values with clients. Meanwhile, also estimate very approximately the likely memory needs of the system you're developing. Identify the tensions between the two. Identify the design decisions that will significantly challenge the memory use, and ensure these decisions happen early.

Based on this comparison you'll be in a position to identify which constraints are most vital. It may be a constraint on one of the forms of memory in the system. Other constraints – time constraints, reliability, usability – may also be as or more important.

Enshrine these priorities as a core 'given' for everyone working on the project. Ensure that absolutely everyone working on the team understands the priorities. Write the strategy in a document; make presentations; distribute the T-shirt! Indoctrinate each new developer who joins the team afterwards with the same priorities.

Once you've identified your priorities, you'll be in a position to plan how to approach the rest of the project. You may need a formal **MEMORY BUDGET**, or perhaps **MEMORY TRACKING**. Or you may choose to leave **MEMORY OPTIMISATION** until near the end of the project. Depending on the nature of the system, you may need to plan for **EXHAUSTION TESTING**, or assign time to **PLUG THE LEAKS**.

For example, the developers of the 'Super-spy 007' decided the important priority was the constraint on RAM, since RAM memory provided the only storage – and a reset might then erase vital information about the Master Villain's plans to destroy the world! The next priority was user response (to give a quick response in dangerous situations). So the components and interfaces are designed to minimise this memory use, and then to give reasonable user response.

Consequences

Every member of the team will understand the priorities. Individual designers will be able to make their own decisions knowing that the decisions will fit within the wider context of the project. Design decisions by different teams will be consistent, adding to the coherence of the system developed.

You can estimate the impact of the memory constraints on project timescales, reducing the uncertainty of the project.

The initial estimates of memory needs can provide a basis for a more formal **MEMORY BUDGET** for the project.

However: Deciding the strategy takes time and effort at an important stage of a project.

Sometimes later design decisions, functionality changes, or hardware modifications may modify the strategy; this invalidates the earlier design decisions, so might leave the project in a worse position than if individuals had taken random decisions.

Implementation Notes

Sometimes it's not necessary to make the strategy explicit. Many projects work in a well-understood context. For example an MS-Windows 'shrink-wrapped' application can assume a total system size of more than 12Mb RAM (and more than 30Mb paged memory), about 50Mb disk and program space – as we can deduce by studying any number of 'industry standard' applications.

So MS Windows developers share an unwritten understanding of the memory requirements of a typical program. The strategy of all these Windows applications and the trade-offs will tend to be similar, and these are often encapsulated in the libraries and development environments or in the standard literature. Given this 'implicit strategy' it may be less necessary to define an explicit one; any programmer who has worked on a similar project or read up the literature will unconsciously choose appropriate trade-offs.

However having an implicit strategy for all applications can cause designers and programmers to overlook lesser but still significant variations in a specific project. For example a Windows photograph editor will randomly access large amounts of memory. So it may have to assume (and explicitly demand) rather more real, rather than paged, memory than other 'standard' applications.

Developers from Different Environments

Programmers and designers used to one strategy often have very great difficulty changing to a different one. For example, many MS Windows programmers coming to the EPOC or Palm operating systems have great difficulty internalising the idea that programs must run indefinitely even if there's a possibility of running out of memory. Windows CE developers have even more of a problem with this, as the environment is superficially similar to normal Windows.

Yet if such programmers continue to program in their former 'large workstation' paradigms, the resulting code has poor quality, and often doesn't satisfy user needs. The developers need to adapt to the new strategies.

One excellent way to promote such 'Thinking Small' is to exaggerate the problem. Emphasise the smallness of the system. Make all the developers imagine the system is smaller than it is! And encourage every team member to keep a very tight control on the memory use. Ensure that each programmer knows which coding techniques are efficient in terms of memory, and which are wasteful. You can use design and code reviews to exorcise wasteful features, habits and techniques.

In this way you can develop a culture where memory saving is a habit. Wonderful!

Guidelines for Small System Development

The following are some principles for designing memory limited software:

Design small, code small	You need to build in memory saving into the design as well as into the code of individual components. The design provides much more scope for memory saving

	than code.
Create bounds	Avoid unbounded memory use. Unlimited recursion, or algorithms without a limit on their memory use, will almost certainly eventually cause irritating or fatal system defects.
Design for the default case	It's always tempting to design your standard object data structures to handle every possible case. But this approach tends to waste memory. It's better to design objects so that their default data structure handles only the simplest case, and have extension objects [Beck ?] to handle special cases.
Minimise lifetimes	Heap- and stack- based objects cease to take up memory when they're deleted. You can save significant memory by ensuring that this happens as early as possible [KenA list in Cacheable Expression]

Extreme vs. Traditional Projects

There are many different styles for teams working on software development. To highlight some of the differences, we'll contrast two opposing styles: 'Traditional Development' and 'Extreme Programming'.

Traditional development [Gilb], [DeMarco] derives its processes and targets from the project controlling techniques used successfully in other engineering disciplines. Each developer is responsible for there own areas of code. A project starts with the team agreeing or receiving a set of specifications from clients at the start of the project – typically as a Functional Specification document. If the project is large enough, a design team will next decide on the architecture and specify the software components for the system. Then separate teams will work on written designs for each component and for the interfaces between them. Finally each team works separately on implementing each component, usually with each programmer responsible for a section of the code and functionality. Either the component programmers or a different team will be responsible for component testing, and then for system testing. Finally the system is 'released' and shipped to the customer, followed by either new projects to modify the functionality, or 'maintenance' to fix defects and shortcomings as required.

In the 'Extreme Programming' style of development, there is a single development team of up to about twelve programmers. Development works in short cycles of a week or so, each cycle culminating in a release – which may potentially be shipped to a customer. The team interacts strongly with their customer to develop only the most important features in each cycle. Programmers always work in pairs, develop complete test code before any implementation, have a strong emphasis on 'refactoring' existing code to satisfy new requirements, and eschew formal design documentation.

One might compare the two approaches to two different ways of house building. A property speculator will create a building by hiring a number of professionals, and arranging for the design to be done first, the builders to ready at the right time to start, the plumbers to be available when the builders have finished the shell, etc.

Extreme programming is more like a couple building their own house. They create the shell, make one room liveable, and take on new projects to improve their facilities and add new rooms when time and money permit.

[Insert here. How you use the patterns in a traditional project. How you use the patterns in an Extreme Project. Get feedback from Kent Beck.]

In a traditional project the architectural strategy is a part of the architect's Vision [?ref. JD?].

In an XP project, the strategy will best be reflected in the project 'metaphor'. [XP ?Wiki]. Individual memory constraints are reflected as 'stories', which become test cases that every future system enhancement must support.

Specialised Patterns

The rest of this chapter introduces six further 'process patterns' commonly used in organising projects with limited memory. Process patterns differ from design patterns in that they describe what you do – the process you go through – rather than the end result.

These patterns apply to virtually all small memory projects, from one-person developments to vast systems involving many teams of developers scattered worldwide. Throughout this chapter we'll use the phrase 'development teams' to mean 'all of the people working on the project'. If you're working alone, you should read this as referring to yourself alone; if a single team, then it refers to just that team; if a large project, it refers to all the teams.

Equally, the patterns themselves work at various levels of a project's organisation. Suppose you're working on the implementation of the Strap-It-OnTM wristwatch computer. The overall project designers ('system architecture team') will use each pattern to examine the interworking of all the components in the system. Each separate development team can use the patterns to control their implementation of their specific component, working within the parameters and constraints defined by the system architecture team.

The patterns are as follows:

Memory Budget How do you keep control in a project where memory is very tight? Draw up a memory budget, and plan the memory use of each component in the system.

Featurectomy How do you ensure you have an implementable set of system requirements given the system restraints? Negotiate with the clients, users and requirements specification teams to produce a specification to satisfy both users needs and the system's memory constraints.

Memory Tracking How do you find out if the implementation you're working on will satisfy your memory requirements? Track the memory use of each release of the system, and ensure that every developer is aware of the current score

Memory Optimisation How do you stop memory constraints dominating the design process to the detriment of other requirements? Implement the system, paying attention to memory requirements only where these have a significant effect on the design. Once the system is working, identify the most wasteful areas and optimise their memory use.

Plugging the Leaks How do you ensure your program recycles memory efficiently? Test your system for memory leakage and fix the leaks.

Exhaustion Test How do you ensure that your programs work correctly in out of memory conditions? Use testing techniques that simulate memory exhaustion.

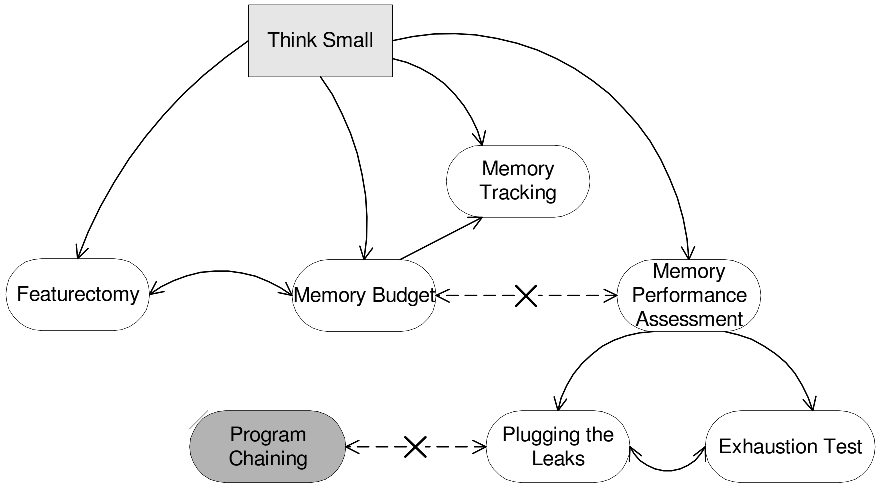


Figure 1: Process Pattern Relationships

Known Uses

The EPOC operating system is ported to many different telephone hardware platforms; each has a different configuration of ROM, RAM and Flash (persistent) memory. So each environment has a different trade-off and application strategy. Some have virtually unlimited non-persistent RAM; others (such as the Psion Series 5) use their RAM for persistent storage so must be extremely parsimonious with it.

In each case, the memory strategy is reflected in the choice of **Data Structures,** in **User Interfaces,** and in the use of **Secondary Storage**. The Psion Series 5 development used an implicit strategy, passed by word of mouth. Later ports have had an explicit strategy documents.

See Also

THINKING SMALL provides a starting point for a project. Most of the other patterns in this book have trade-offs that we can evaluate only in the context of a memory strategy.

Memory Budget Pattern

A.k.a. Memory Costings

How do you keep control in a project where memory is very tight?

- You're doing a project where memory is limited and there's a risk that the project will fail if its memory requirements exceed these limits.
- You have several different components or tasks using memory
- Different individuals or teams may be responsible for each.
- Saving memory costs effort – better let someone else do it!
- Unnecessary optimisation would waste programmer time.

You are working on a software development project, and you've identified that there's a possibility that memory constraints may be a significant problem.

For example, the whole Strap-It-On project is obviously limited by memory from the beginning. The Strap-It-On needs to be as small, as cheap, and as low-powered as possible, but also be usable by computer novices and have enough capacity to be adopted and recommended by experts.

If you don't take sufficient care of the memory constraints in the system design and implementation, bad things will happen to the project. Perhaps the system will fail to work at all; perhaps users will get inadequate performance or functionality; or perhaps the cost of the extra memory hardware will make the software unsaleable.

You could have everyone involved design and code so as to reduce their memory requirements to the bare minimum. That would certainly reduce the risk of the system becoming too big. But there are be costs to this scorched earth approach – if you concentrate on keeping memory low, then you'll have to accept trade-offs elsewhere such as poor time performance, difficult-to-use interfaces or large amounts of developer effort. It would be poor engineering to concentrate on one aspect, memory use, to the exclusion to all others. More importantly, how can you decide what the "bare minimum" actually is? You could save all the memory by deciding not to implement the program!

In almost any modern system you will be developing or using several components, each with its own memory requirements. Some will provide more opportunities for memory saving than others. There's no point in working overtime to save a few bytes in one component, when a minor change in another would save many times that. How do you decide which components to concentrate on?

In many projects there will be several teams each working on different components. Each individual team may feel they have less incentive to save memory than other teams — everyone likes to believe that the problem they are working on is unique, and harder than everyone else's problem. It costs teams programmer effort to reduce memory use – so they'll be tempted to let a different team pay the cost, treating memory as "someone else's problem". How can you share out the pain of saving memory between the teams, so that they can design their software and plan its implementation effectively?

Therefore: *Draw up a memory budget, and plan the memory use of each component in the system.*

Define memory consumption targets for the each component as part of the specification process. Ensure that the targets are measurable [Gilb88], so the developers will be able to check whether they're within budget.

This process is similar to the 'costings' process preceding any major building work. Surveyors estimate costs and time of each part of the process, to determine the feasibility and to negotiate the requirements of the customer.

Ensure that all the teams 'buy into' the budget. Involve them in the process of deciding the figures to budget, estimating the memory requirements and negotiating how any deficits are split between the different teams. Communicate the resulting budget to all the team members and invite their comments. Refer to it while doing **MEMORY TRACKING** during development, and during the **MEMORY PERFORMANCE ASSESSMENT** later in the project. Make meeting the budget a criterion for release of each component. Celebrate when the targets are met!

Consequences

The task of setting and negotiating the limits in the memory budget encourages all the teams to **THINK SMALL**, and sets suitable parameters for the design of each component. The budget forces the team to take an overall view of memory use, increasing the *architectural consistency* of the system. Furthermore, having specific targets for memory use greatly increases the *predictability* of the memory use of the resulting program, and can also reduce the program's absolute *memory requirements*.

Because developers face specific targets, they can make decisions *locally* where there are trade-offs between memory use and other constraints. It's also easy to identify problem areas, and to see which modules are keeping their requirements reasonable, so a budget increases *programmer discipline*.

However: defining, negotiating and managing the budgets requires significant *programmer effort*.

Developers may be tempted to achieve their *local* budgets in ways that have unwanted *global* side effects such as poor *time performance*, off-loading functionality to other modules or breaking necessary encapsulation (see [Brooks75]). Runtime support for testing memory budget requires *hardware or operating system support.*

Setting fixed memory budgets can make it more difficult to take advantage of more memory if it should become available, reducing the *scalability* of the program.

Formal memory budgets can be unpopular with both programmers and managers because the process adds accountability without direct benefits. If the final system turns out over budget then everyone will loose out; if it turns out under budget then the budget will have been 'wrong' – so those doing the budget may loose credibility.

Implementation Notes

Suiting Budget to the Project

Producing and tracking an accurate memory budget for a large system is a large amount of work, and can impose a substantial overhead on even a small project. If memory constraints aren't actually a problem, maintaining budgets is rather a waste of effort that could be better spent elsewhere. And in an informal environment, with less emphasis on up-front design, developers can be actively hostile to a full-scale formal memory budget.

For this reason, many practical memory budgets are just back-of-the envelope calculations – a few minutes work with the team on the whiteboard, summarised as a paragraph in the design documentation. Only if simple calculations suggest that memory will be tight – or tight in certain circumstances – is it worth spending the effort to put together a more formal memory budget.

What to budget?

There are various kinds of memory use; different environments will have different constraints on each. Here are some possibilities:

- RAM memory usage – heap memory, stack memory, system overheads.
- Total memory usage – including memory **PAGED OUT** to disk.
- ROM use – for systems with code and data in ROM
- Secondary storage – disk, flash and similar data storage, network etc.

In addition, the target environment may add further limitations: a limit on each separate process (such as for code using the 'Small', 16-bit addressing model on Intel architectures), or a limit on stack size (imposed by the operating system).

It's worth considering each constraint in turn, if only to rule most of them out as problems. Often only one or two kinds of memory will be limited enough to cause you problems, and you can concentrate on those.

Dealing with Variable Usage

It's easier to budget ROM usage than RAM. ROM allocation is constant, so you can budget a single figure for each component. Adding these figures together will give the total ROM use for the system.

In contrast, the RAM (and secondary storage) requirements of each component will normally vary with time – unless a component uses only Fixed Data Structures.

One approach is to estimate the worst case memory use of each component and adding the values together, but the result could well be far too pessimistic; in many systems only a few components will be being used heavily at a time. A workstation, for example, will have only a few applications running at any one time – and typically only one or two actively in use.

Yet the memory use of the different components tends not to be independent. For example, if you have an application making heavy use of a, then the applications peak memory usage is likely to coincide with peak memory use in the network driver. How do you deal with this correlation?

To deal with these dependencies, you can identify a number of worst case scenarios for memory use, and construct a budget for the memory use of each component in each scenario. Often, it is enough to estimate an average and a peak memory requirement for each component and then estimate which components are likely to have peak usage for each worst-case scenario. You can then sum the likely use for each scenario; and negotiate a budget so that this sum is less than the total for every one of the scenarios.

Dealing with Uncertainty: Memory Overdraft

Software development in the real world is unpredictable. There's always a possibility for any component that it will turn out to be just too difficult or too expensive in time or other trade-offs to reduce its memory requirements to the budgeted limits. If there are many components, there'll be a good chance that at least one will be over budget, and the second law of thermodynamics [Flanders&Swan] says it is unlikely that components will be correspondingly under budget.

The answer is to ensure that there is some slack in the budget – an overdraft fund. The amount depends on how uncertain the initial estimates are. Typical amounts might be between 5% and 20%. The resulting budget will be more resilient in the face of development realities, increasing the overall *predictability* of the program's memory use. However you must be careful to ensure that programmers don't reduce their *discipline* and take the overdraft for granted, reducing the integrity of the budget.

The OS/360 project included overdrafts as part of their budgets [Brooks75].

Dealing with Uncertainty: A Heuristic Approach

Having a Memory Overdraft to allocate to defaulting components is a good ad-hoc approach to dealing with uncertainty, but it's a bit arbitrary. If you're seriously strapped for memory, allocating an arbitrary amount to a contingency fund isn't exactly a very scientific approach. Should you assign 5% or 30%? If you assign 30%, you're wasting a very large amount of memory that you could more economically assign to a component. If you assign less, how much are you increasing the risk?

The solution is to use a technique publicised as part of the 'Program Evaluation and Review Technique' (PERT). This is normally used to add together time estimates for project management – see [Filipovitch96], but the underlying statistics work equally well for adding together any set of estimated values.

Make three estimates for each figure rather than just one. Estimate a reasonable worst case value, the most likely (median) value, and a reasonable best achievable (i.e. lowest) maximum value. Try to do your estimation impartially so that it's equally likely that each final figure will turn out higher or lower than the median you've estimated. So when you add them together, probably some of the final figures will be higher and some of them will be lower. In effect combining the all the uncertain figures means that some of the uncertainty 'cancels out'.

The arithmetic of this is as follows. If the estimated value for component *i* is e_i, and the maximum and minimum values are a_i and $b_{I,}$, then the best guess, or 'weighted mean' value for each is:

$$(a_i + 4e_i + b_i) / 6$$

And the best guess of the standard deviation of each is:

$$\sigma_i = (b_i - a_i) / 6$$

So the best estimate of the sum is the sum of all the weighted means; and we calculate the standard deviation of the sum, S_I using:

$$S_i \wedge 2 = \mathrm{Sum}_i(\sigma_i \wedge 2)$$

These calculations are very easy to do with a spreadsheet.

For example, Table 1 shows one possible worst-case scenario for the Ring Clock, a kind of watch device worn on the finger than receives radio time checks from transmitters in Frankfurt. This scenario shows it ringing an alarm. Only the Screen Driver and the UI Implementation components are heavily used:

Table 1 : Calculation of the Combination of Several Estimates

A good estimate of the maximum and minimum values for the sum is three standard deviations (the so-called '95% confidence limits') above and below the mean. The table above shows the standard deviation to be roughly 2Kb, which gives the following values for the combined estimates:

Maximum:	41K
Estimate:	35K
Minimum:	29K

So we can be reasonably confident that we shall be able to support this particular worst-case scenario with 41K of memory – much less than the sum of the all the maximum estimates.

If the actual memory available is less, then we might need to work on the most variable estimates (UI Implementation) to produce a more accurate estimate – since the large maximum has contributed much of the uncertainty in the figure. Alternatively we might need to do some **FEATURECTOMY** to reduce the estimated memory requirements of that or other components.

Enforcing the Budget in Software

Some environments provide memory use monitors or resource limits, which you can use to enforce memory budgets. For example IBM UNIX allows you to define a limit on the heap memory of a process, and EPOC's C++ environment can enforce a maximum limit on application heap sizes. The **MEMORY LIMIT** pattern describes how you can implement these limits yourself.

You can use these monitors to enforce the limits on the maximum memory use of each component. Some projects may use these limits for testing only; in other cases they may remain in the runtime system, so that processes or applications will fail (**PARTIAL FAILURE**, or complete failure) rather than exceed their budget.

Of course software limits enforce only the maximum use for each component. Typical worst case scenarios will have only a few components using their maximum memory requirements, so such software limits don't provide a full check that the components are conforming to the budgets.

Example

The Palm Pilot has an interesting budget for its dynamic heap (used for all non-persistent data). Because only one application runs at a time (**PROGRAM CHAINING**), the budget is the same for every application that can run on a given machine.

The following is the Pilot's budget for PalmOs 3.0, for any unit with more than 1 Mbyte of memory [PalmBudget]. Machines with less memory are even more constrained.

24k	System globals (screen buffer, UI globals, database references, etc.)
32k	TCP/IP stack, when active
Variable amount	IrDA stack, "Find" window, other system services
4k (by default)	Application stack (the application can override this amount)
up to 36k	Available for application globals, static data, dynamic allocations, etc.

Table 2: Palm Pilot 3.0 Memory Budget

Known Uses

[Brooks75] discusses the memory budget for the OS/360 project. In that project, the managers found it important to budget for the total size of each module (to prevent paging), and to specify the functionality required of each module as a part of the budgeting process (to prevent programmers from offloading functionality onto other components).

A current mobile phone project has two particular architectural challenges provided by a hardware architecture originally defined for a very different software environment. First, ROM (flash RAM) is very limited. Based on a Memory Budget, the team devised compression and sharing techniques, and negotiated Featurectomy with their clients.

Secondly, though RAM in this phone is relatively abundant, restrictions in the memory management architecture means that each process must have a pre-allocated heap, so every process uses the RAM allocated to it at all times. Thus the team could express the RAM budget in terms of a single figure for each process – the maximum, or worst case, figure.

The Palm documentation specifies a standard memory budget for all Pilot applications. Since only one application runs at a time, this is straightforward.

See Also

There are three complementary approaches to developing a project with restricted memory. The MEMORY BUDGET pattern describes how to tackle the problem up front, by predicting limits for memory, and then implementing the software to keep within these limits. The MEMORY TRACKING pattern gathers memory use statistics from developers as the program is being built, encouraging the developers to limit the contribution of each component. Finally, if memory problems are evident in the resulting program, a MEMORY PERFORMANCE ASSESSMENT the developers uses post-hoc analysis to identify memory use hot spots and remove them.

For some kinds of programs you cannot produce a complete budget in advance, so you may need to allocate memory coarsely between the user and the system, and then MAKE THE USER WORRY about memory.

Components that use FIXED SIZE MEMORY are much easier to budget than those using VARIABLE SIZE MEMORY.

Systems that satisfy their RAM or secondary storage memory budget when they're started may still gradually 'leak' memory over time, so you'll need to Plug the Leaks as well.

[Gilb88] describes techniques for 'attribute specification' appropriate for defining the project's targets.

Featurectomy Pattern

Also known as: Negotiating Functionality

How do you ensure you have realistic requirements for a constrained system?

- Software is 'soft', so the costs of extra functionality are hidden from those who request it.
- Extra functionality costs code and often extra RAM memory
- Specification teams and clients have a vested interest in maximising the functionality received.
- Some functionality confers no benefit to the users.

Software is soft; it's infinitely malleably. Given sufficient time and effort you can make it do virtually anything. But it costs time, effort and memory to achieve this.

Non-programmers are often unaware of this cost (programmers too!). And even if they are aware, or are made aware, many have no means of knowing exactly what the costs are. Will it take more memory to speed up the network performance by 50% than to add a new handwriting input mechanism? It's difficult for a non-programmer to know.

And in most environments the development team - and particularly the specification team if there is one - is under very great pressure to add as much functionality as possible. Functionality, and elegant presentation of functionality, is the main thing that sells systems. From the point of view of the client or specification team the trade-off is simple: if they ask too little functionality they may be blamed for it; if they ask for too much, the development team will take the blame if they don't deliver it. So the pressure is on them to over-specify.

Yet it's rare that all the possible functionality specified is essential, or even beneficial. For example some MS Windows applications contain complicated gang screens; MS Word 6 even includes an entire undocumented adventure game, hidden from all but initiates. Many delivered systems retain some of their debugging code, or checks for errant – and impossible – behaviour. Such additional code costs both code and often RAM memory in the final system. Yet they provide no service at all to the user.

Therefore: *Negotiate a specification to satisfy users within the memory constraints*

Analyse the functionality required of the system both in terms of its priority (how important is it?) and in terms of its memory cost (how much memory will it use?). Based on that, negotiate with your clients to remove – or reduce or modify – the less important and more memory intensive features.

Ensure that you remove any additional code for testing and debugging when you make a final release of the software.

Consequences

The released software needs to do less, so uses less ROM and RAM memory. In systems that implement Paging, the smaller code and memory sizes make for less disk swapping, improving the system's time performance.

There is less functionality to develop and to test, giving reduced development and testing. Because there is less functionality, there can be less interaction between features, leading to a more reliable, and often more usable, system.

However: The system has less functionality, potentially reducing its usability.

Unless the negotiation is handled carefully, the development team can be seen as obstructive to the client's goals, reducing client goodwill.

Implementation Notes

It can be difficult to impress on even technically-minded clients that memory limits are a significant issue for a project. Most people are familiar with the functionality of a standard MS-Windows PC, and find it difficult to appreciate the impact of much lower specification systems.

A good way to approach the negotiations is to prepare a Memory Budget allocating memory costs to each item of functionality – see *Functionality a la Carte* [Adams95]. Although this can of course be difficult to do, it makes negotiation straightforward.

Given this shopping list, and the fixed total budget, then the specification team and customers can make their own decisions about what functionality to include. Often they will have a much better idea of the importance of each feature, so they can make the trade-offs between options.

The Next Release

Frequently people (clients or developers) become 'wedded' to features, perhaps because it was their idea, or because somebody powerful wants it. In that case it becomes very difficult to negotiate such features out of a product no matter how sensible it may appear to everyone else concerned.

In that case a common approach is to agree to defer the feature until the next system release. By then it may well be obvious whether the feature is necessary, but also it will allow a more impartial appraisal once time has gone by.

Supporting Variant Systems

Features that are essential to one set of users may be irrelevant to others. In many cases there won't be any single user who needs all the system functionality.

So you can provide optional features in separate **PACKAGES**, which can be left uninstalled or merely not loaded at run-time. In systems that don't support packages, you might use conditional compilation or source code control branches to tailor different systems to the needs of different sets of users.

Sometimes this results in two-tier marketing of the system: a base-level product with low memory demands, and a high-tier ('professional') product with higher hardware requirements.

Thin Client

One particularly powerful form of **FEATURECTOMY** is possible when there is some form of distribution with a central server and one or more client systems. In such 'client-server' systems the trend until recently has been to have much of the business processing at the clients ('fat clients'), talking to a distributed database. This approach obviously requires a lot of code and data in the client. And it may well be unsuitable if the client has little memory or processing power.

Instead, given such a system, it is often possible to offload much of the processing to the server. You can do this by making the client simply be a graphics workstation (provide an character or X-windows terminal emulation). But often a better approach is to implement a 'thin client', which provides a UI and does simple user input validation, but which passes all the business-specific processing to a central server.

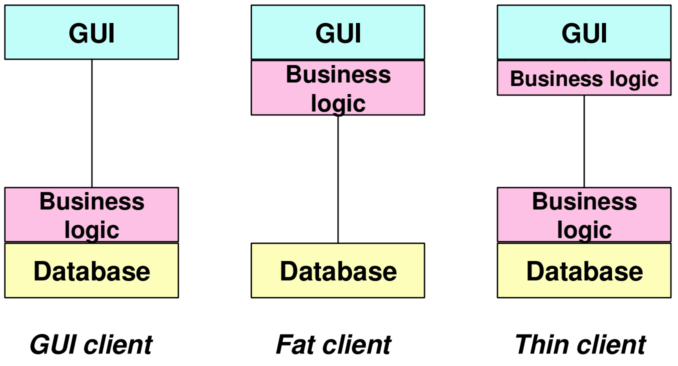


Figure 2: Three kinds of client-server

Other Featurectomy options:

Often there are specification alternatives to simply cutting out a feature altogether. For example you might agree **LOWER QUALITY MULTIMEDIA** for the implementation, or use **FIXED USER MEMORY** in the interface, to reduce the memory demands. You might **MAKE THE USER WORRY** – for example by making the user to explicitly start any functionality required, rather than starting it automatically.

You may be able to cut down on the additional data demands of the system. For example a mapping application might store only a subset at any time of all the maps required; a dictionary application might support only one language or technical subset at a time; an operating system might cut down on the number of simultaneous services available.

Debugging Code

One key set of users whose needs are different from others is the programmers themselves testing and debugging the system. Examples of such code are:

- Tracing code, to show what the program is doing.
- Checking code, to verify that the program is working correctly. Examples are assertions and invariants [Meyer]
- Debugging test harnesses, such as 'main()' functions added to classes for localised testing.
- Debugging support functions, such as functions to allow test code to access 'private' data for 'white box testing' [test]
- Instrumentation macros for memory and performance optimisation (see the Plugging the Leaks Pattern).

Clearly none of this code is vital to the delivered system. It will waste code space, and potentially impact the time performance of the system. So you'll want to remove it from the deliverable product. Ideally, though, you'll want to keep it in your codebase, so that it's available for future testing, debugging and optimisation.

The Eiffel language [Eiffel] is designed specifically to support this kind of dual mode. In debugging, it encourages programmers to define additional checking code: preconditions and postconditions for each function, and invariants for each class. In release mode the compiler doesn't generate this checking code at all.

Conditional Compilation in C++

In C++ the usual technique is to use pre-processor flags and macros. For example

```
#ifdef DO_TRACE
#   define TRACE( x ) printf( x )
#else
#   define TRACE( x )
#endif
```

allows us to use the TRACE throughout the code. When debugging, we can declare the DO_TRACE macro (in a global header file, or on the compiler command line); in the final system we can omit it.

An even more common form of this in C++ is the assert macro, built into the C++ environment (header file assert.h):

```
#ifdef NDEBUG
#   define assert(exp)    ((void)0)
#else
#   define assert(exp) (void)( (exp) || (_assert(#exp, __FILE__, __LINE__),
0) )
#endif  /* NDEBUG */
```

The _assert() function here displays a text message with the text of the assertion, and the location (sometimes in a dialog box). Then you can use expressions like:

```
assert( x== 0 );
```

and in debug mode the assertion is tested; in release mode the line of code is **FEATURECTED**.

Conditional Compilation in Java

Conditional compilation in Java uses a compiler optimisation. Most Java compilers can detect when certain code is 'dead' and will not produce corresponding byte codes. So a test using a static final boolean provides conditional compilation:

```
class Assertions  {
    public static final boolean isEnabled = true;
    // Change to false for release

    public static void assert( boolean assertion, String message ) {
        if (!assertion)
            throw new Error( "Assertion failed: " + message );
    }
}
```

You might use this as follows:

```
    public static void main( String[] args ) {
        try {
            int x=0;
            if (Assertions.isEnabled)
                Assertions.assert( x == 1, "x is non-zero" );
            Assertions.assert( x==1, "second one" );
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
```

Examples

[Matrix for Strap-it-on showing features, estimated peak and average ROM and ROM use to support each, and development time in man-days]. Based on this we decided to exorcise feature X.

Known Uses

In a recent Symbian EPOC mobile phone development, the initial ROM demands were way over budget. The development team used a **ROM BUDGET** and **MEMORY**

TRACKING to analyse the problem, and negotiated **FEATURECTOMY** with the client's specification team. In particular they agreed to have different language variants of the system for different markets, thereby considerably cutting down total size of **RESOURCE FILES**.

Microsoft Windows CE provides pocket versions of MS Word, MS Excel and other applications. In each application, the CE developers have cut down considerably on the functionality. For example Pocket Word 2.0 omits, amongst many other things, the following features of MS Word 97:

- Thesaurus
- Mail-Merge
- Auto Format
- Clip-art
- Support for non-TrueType fonts
- Float-over-text pictures

Pocket Word also uses a different internal file format from any MS Windows version of Word, **MAKING THE USER WORRY** about file conversion.

See Also

Usually you will need a **MEMORY BUDGET** as a basis for Featurectomy negotiations.

MEMORY TRACKING allows you to see the effects on memory use as features are implemented. Featurectomy may be appropriate if things look bad.

Some forms of **UI PATTERNS** may provide Featurectomy without significantly affecting the usability of the system. For example **FIXED USER MEMORY** provides feature with a fixed maximum memory use. And **USER MEMORY CONFIGURATION** allows the user to chose which features are present at run-time.

Alternatives to Featurectomy include **COMPRESSION**, using **SECONDARY STORAGE**, **PACKED DATA** and **SHARING** – and indeed most of the other patterns in this book.

Memory Tracking Pattern

A.k.a. Memory Accountant, Continuous Programmer Feedback

How do you find out if the implementation you're working on will satisfy your memory requirements?

- You're doing a project that may fail if the memory requirements exceed certain limits.
- The development team needs continued motivation to restrict memory use.
- Many teams are hostile to the formality of a full Memory Budget.
- If things are going well there's no point in going to unnecessary effort.

You are working on a software development project for a system with limited memory. Bad things will happen if the final system exceeds its memory constraints.

Yet the programming team – including yourself – may find it difficult to judge how important the problem is. People who've only worked in relatively unconstrained environments often have difficulty coming to terms with tight memory limits and the different programming styles these imply. Alternatively, they may overestimate the danger, and waste effort on unnecessary memory optimisation.

If you're working in a relatively informal environment, you may find that a detailed Memory Budget – with its culture of advanced planning and estimation – may not be welcome to your co-developers. They may resist the process, or simply ignore the results. Yet you still need to bring home the need to Think Small and to design the system to satisfy the memory constraints. Even if the team are willing participants in a budgeting process, that can involve lots of effort and overhead to construct and maintain budgets —more so if you are in a formal environment with lots of paperwork.

Alternatively, if you wait until almost the final system release and do a Memory Performance Assessment, then there's a possibility your designs and implementation may be too inflexible to allow much improvement at the last moment. What should you do?

Therefore: *Track the memory use of each release of the system, and ensure that every developer is aware of the current score.*

With each significant release of the system, use tools to examine the memory use of the entire system and – as far as possible – of each component within it. Publish this knowledge to all of the team.

Use graphs to show how the memory use varies between releases – ensure that everyone understands that a downward pointing graph is desirable, and label any major changes in the memory use of a component with a brief explanation.

If necessary, track the various worst-case scenarios (see Memory Budget), and track separately the figures for each one. If you have a Memory Budget, then compare the current figures with the targets in the budget. Consider creating a memory accountant role to perform this memory tracking. There are advantages if this role is not filled by the main project manager or technical lead — partly to lower their workload, but also to reduce the impression of management checking up on programmers.

Consequences

The feedback of their current memory status encourages every programmer to **THINK SMALL**, without a need to impose the formal limits of a **MEMORY BUDGET**. Doing just **MEMORY TRACKING** can also take *less programmer effort* on an ongoing basis than full budgeting, since programmers will deduce the need to save memory for themselves. So this pattern can be very effective in less formal development environments, creating self-imposed *programmer discipline*, which will help reduce the *absolute memory requirements* of the system.

Having figures for memory use increases the *predictability* of the memory use of the resulting system. It highlights potential *local* problem areas early, so you can address them or schedule time for a Memory Performance Assessment.

If you can produce figures for the separate system components, then you can establish the actual contribution of each component, showing the *local* contribution of each to the *global* memory use.

However: Tracking memory use and producing the feedback needs *programmer effort*. Measuring the memory use may require *hardware or operating system support,* or mean further *programmer effort* to instrument the code accordingly, especially to measure RAM allocation and use.

There's a danger that early figures, when the functionality is incomplete, may be misleadingly low. Or that you may have chosen an unrepresentative set of worst-case scenarios. Either of these factors can cause the figures to be over-optimistic, lulling the team into a false sense of security and discouraging future *programmer discipline*.

Implementation Notes

How do you do the measurements? This section examines tools and techniques to find out the memory use of the various components in a system.

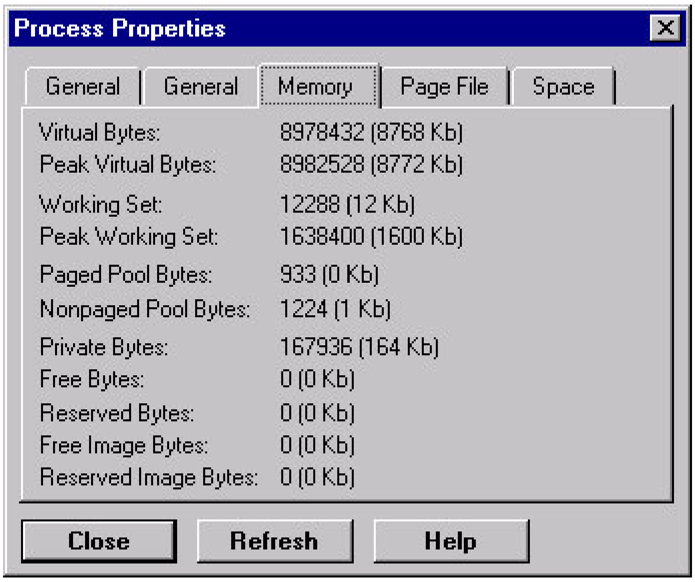
As discussed in the Memory Budget pattern, there are several types of memory use you may want to track including: Total RAM memory, 'Live' RAM usage, ROM use and Secondary storage.

External Tools

It's usually straightforward to measure ROM use – just examine the sizes of the files that make it up, and the size of each ROM image itself. However how are you to measure the memory use that varies with time?

Similarly in most environments you can measure the secondary storage used by the application at a given time using the file system utilities.

Many environments also provide utilities to track the RAM use of each process. For example, Microsoft Developer Studio's Spy++, [MicrosoftSpy97], allows you to examine the memory use of any process under Windows NT. Figure 3 shows an example display. The important figures are "Private bytes", which gives the current heap memory use for the process, and 'peak working set' (see the Paging pattern) which gives the minimum RAM that might be needed.

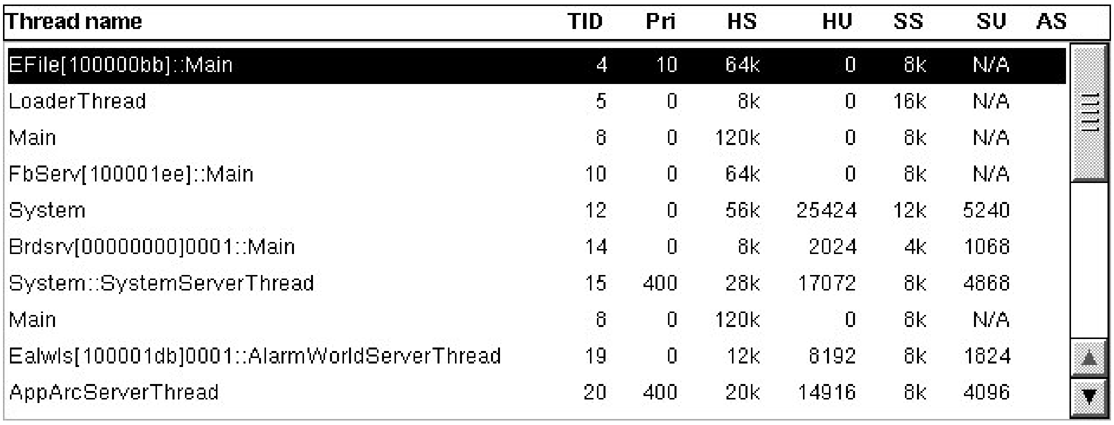


Virtual Bytes: 8978432 (8768 Kb)
Peak Virtual Bytes: 8982528 (8772 Kb)
Working Set: 12288 (12 Kb)
Peak Working Set: 1638400 (1600 Kb)
Paged Pool Bytes: 933 (0 Kb)
Nonpaged Pool Bytes: 1224 (1 Kb)
Private Bytes: 167936 (164 Kb)
Free Bytes: 0 (0 Kb)
Reserved Bytes: 0 (0 Kb)
Free Image Bytes: 0 (0 Kb)
Reserved Image Bytes: 0 (0 Kb)

Figure 3: Microsoft Spy++ Display for a Process

Unix systems similarly provide tools such as `ps` and `top` to list processes' memory use.

The EPOC operating system supports rather more memory-constrained systems, so provides a rather more detailed display of memory use using its 'Spy' tool:



Thread name	TID	Pri	HS	HU	SS	SU	AS
EFile[100000bb]::Main	4	10	64k	0	8k	N/A	
LoaderThread	5	0	8k	0	16k	N/A	
Main	8	0	120k	0	8k	N/A	
FbServ[100001ee]::Main	10	0	64k	0	8k	N/A	
System	12	0	56k	25424	12k	5240	
Brdsrv[00000000]0001::Main	14	0	8k	2024	4k	1068	
System::SystemServerThread	15	400	28k	17072	8k	4868	
Main	8	0	120k	0	8k	N/A	
Ealwls[100001db]0001::AlarmWorldServerThread	19	0	12k	8192	8k	1824	
AppArcServerThread	20	400	20k	14916	8k	4096	

Figure 4: EPOC Spy Display

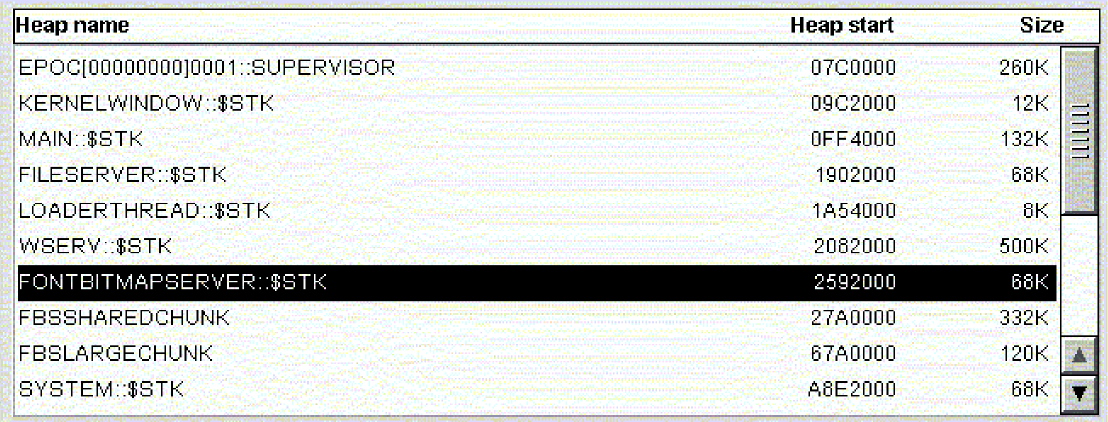
The EPOC Spy Display in Figure 4 shows for each thread, the total heap size (HS), the RAM allocated within each heap (HU), the Stack size (SS), and – where the process protection permits – the maximum stack usage (SU).

Code Instrumentation for RAM use.

System tools can normally only show you an external view of each separate process, showing its total memory use. Perhaps your important components are more fine-grained than individual processes, or perhaps there are no system tools available for your particular environment. What should you do then?

The solution is to 'instrument' your own code, adding test code to track memory allocations and de-allocation. See the Memory Limit pattern for details.

Another possibility, if your environment supports it, is to use separate Heap structures for each component, and use the system tools to examine the memory use of each heap. This is possible using EPOC's SPY tool – for example Figure 5 shows three heaps owned by the Font Bitmap Server process (FONTBITMAPSERVEER:$STK, FBSSHAREDCHUNK and FBSLARGECHUNK)



Heap name	Heap start	Size
EPOC[00000000]0001::SUPERVISOR	07C0000	260K
KERNELWINDOW::$STK	09C2000	12K
MAIN::$STK	0FF4000	132K
FILESERVER::$STK	1902000	68K
LOADERTHREAD::$STK	1A54000	8K
WSERV::$STK	2082000	500K
FONTBITMAPSERVER::$STK	2592000	68K
FBSSHAREDCHUNK	27A0000	332K
FBSLARGECHUNK	67A0000	120K
SYSTEM::$STK	A8E2000	68K

Figure 5: EPOC Spy: Heap Display

Code Instrumentation to find Maximum Stack use

There are at least two approaches to estimating the maximum stack use of a thread.

You can use compiler tools instrument each function call explicitly. For example in debug mode Microsoft Visual C++ inserts calls to a function _chkstk() at the start of each function. If you can replace the (undocumented) implementation of this function, then it's straightforward to find out the current stack use. Many other compilers support similar features.

A less intrusive approach is to ensure the unused portion of the stack is filled with a random value, possibly zero. At the end of the process – or at any other time – you can track to see how much of this memory has been overwritten. EPOC, for example, initialises all stacks to an arbitrary 29 hex, and EPOC Spy uses this when it examines the stack of each readable process to discover its used stack (SU in Figure 4).

Case Study

[Example of the same system as Memory Budget, showing graphs of actual measured memory use plus target for three releases, for several components, both worst case and normal. How do we combine the graphs? this sounds great!]

Known Uses

The EPOC operating system aims to support systems with constrained hardware. In particular, some of the target platforms have hard limits on their ROM memory for storing code. To explore this problem, the design team recently produced a Memory Tracking document, identifying all the components and showing their ROM use over successive releases of the operating system. The document discusses the reasons for each component size change – typically, of course, increases in functionality. It provides an excellent basis for a Memory Budget for any future system, both by suggesting the memory use of combinations of components, and by suggesting the possible gains from featurectomies in specific components.

Brooks?

Roxette project, of ROM.

See Also

Memory Budget can provide a set of targets for developers to compare with the tracked values. Memory Performance Assessment uses similar techniques to determine the memory use at a particular point, as a basis for memory optimisation.

The Memory Limit pattern describes techniques to track memory allocation within your programs at runtime.

Extreme Programming (XP) advocates continual tracking of all aspects of a development project, and stresses the advantages of the auditor role not being part of the project's power structure [Beck, 1999].

Memory Optimisation Pattern

A.k.a. Memory Performance Assessment.

How do you stop memory constraints dominating the design process to the detriment of other requirements?

- You're developing a system that may turn out to be memory-constrained, or you're porting an existing system to a more constrained environment.
- You don't want to devote too much effort to reducing memory requirements early in the project – it may prove unnecessary.
- Often other constraints may be more important than memory constraints
- When the system is near to release, memory performance does turn out to be a problem.

Your system has to meet particular *memory requirements*, but other requirements are more important. A full-scale Memory Budget or Memory Tracking would cost *programmer discipline* and *programmer effort* that could be better directed towards other requirements.

For example, if you're developing a new operating system release for desktop PCs, then integrating a web browser into the desktop, providing AI-based help systems and shipping the system less than a year late, will all be more important than controlling the amount of memory the system occupies.

Yet there's a reasonable likelihood that the system's memory constraints may prove a problem. How do you allow for this possibility in the development?

Therefore: *Implement the system normally, then optimise memory use afterwards.*

Implement the system, paying attention to memory requirements only where these have a significant effect on the design. Once the system is working, identify the most wasteful areas and optimise their memory use.

Using this approach development proceeds much as usual, with not much special attention paid to memory use. You'll usually do a quick informal **MEMORY BUDGET** very early on, just to make sure there are no really pressing concerns. And where the implementation and design costs are reasonably low, you'll use patterns to reduce the memory use – but only where these don't conflict with more vital design constraints.

If the resulting system meets its memory requirements, then that's as far as you need take matters. But if, as often happens, the program does not meet its *memory requirements* you to do a Memory Performance Assessment.

By an examining the code, using profiling tools, and any other appropriate techniques, find out where the system is using most memory. Identify the most promising areas to improve memory use, and implement changes accordingly. Repeat this process until the system satisfies its memory constraints.

Consequences

The team develops the system effectively and faster, because they are not making unnecessary optimisations — a given amount of programmer *effort* gets you more software with better *time performance* and higher *design quality*.

Initial development can be less constrained, improving programmer *motivation.* In addition the performance assessment is a single, short-term task, for the team to achieve – also improving programmer *motivation*.

However: The *memory requirements* of the resulting program will be hard to *predict.* In many cases it requires more *programmer effort* to leave memory optimisation to last than performance optimisation would, because memory optimisation tends to require changes to object relationships that can affect large amounts of code.

Memory optimisation after implementation is more likely to compromise the design, leading to poorer *design quality* than in systems designed from the start to support memory restrictions. Local optimisations may also compromise the global integrity and overall architecture of the system.

Finally the optimised system will need testing, increasing the total *testing cost.*

Implementation Notes

Static much easier than dynamic

Static Analysis

Code analysis. What structures are there? How much memory will each use?

Code and static data size – look for redundant code (execution trace tools; human examination; linker map tables).

Look particularly for any object that will have very large numbers of instances.

Memory Profiling Techniques

Memory Tracking discussed techniques to find the total memory used by each component. How do you examine each component to find out whether and how it's wasting memory?

Tracing as objects are created and destroyed. CodeXXXX shows objects being created and destroyed.

Heapwalk tools (Windows, EPOC) show the objects.

Optimisation Approach

Low hanging fruit.

you should only optimise as and when needed, never in advance. you should define measurable performace criteria, optimise until you meet them, and then stop --- backing out other optimisations if they don't contribute much to the final result.

* you should be able to undo optimisations, and should undo them when they are no longer needed.

* cite the Lazy Optimisation pattern (plopd2 or 3, it's in the almanac) LOTS. if you have time, its worth a read to get their mindset, which seems right on the button.

Optimisation Patterns

Groups of objects – all.

Local changes: Packed data, Memory Pooling, Multiple Representations.

Compression: String Compression, File Compression, (Tokens)

Secondary Storage: Packages, Data Chaining (local temp file)

What is and is not safe to leave until Memory Optimisation

Leave only local optimisations.

Small Interfaces don’t leave. Partial Failure. Fixed DS.

UI Don’t leave (except Notifier)

The deal here is interfaces vs. implementation. You can optimise implementations locally. You have to optimise interfaces globally, which often translates to --- you can't optimise them. also, the flip side of this is, what optimisation techniques are (and are not) safe to do? I.e. you don't want to break modularity when you optimise a local implementations, or else you can't undo them.

Example

[Regclean] provides an example of an assessment of the redundant memory use in an non-optimised workstation product. Quote it?

Known Uses

This pattern occurs very frequently in projects, since it is what happens by default. Typically a team implements a system, then discovers it uses too much memory. Memory optimisation follows. Another common situation is in porting an existing system to an environment with insufficient memory to support the system unchanged. A memory performance assessment must be part of the porting process.

For example in the development of the Rolfe&Nolan ‘Lighthouse’ system to capture financial deals, the team developed the terminal software and got it running reliably. However they then discovered that some users required to keep information about huge numbers of deals in memory at one time – and the cost of paging the memory required made the performance unacceptable. The team did an informal memory assessment, and optimisation. They found they had lots of small, heap-allocated objects. So they purchased an improved heap management library with a much smaller overhead both for each allocated block and for fragmentation. They identified that they didn’t need the full information for each deal, so they used Multiple Representations to reduce the memory each deal takes by default.

[Blank+95] describes the process of taking an existing program and optimising it to run under more constrained memory conditions.

See also

The patterns Lazy Optimisation and Optimise the Right Place in [Auer+95] address speed optimisation, but the techniques apply equally to space optimisation. Two relevant low-level patterns in the same language are Transient Reduction and Object Transformation.

If you are facing particularly tight memory requirements, prefer to think ahead, or are a pessimist, then you should prepare a Memory Budget in advance so you can plan your use of memory. If the system is going to undergo continual change, then you should do Memory Tracking to ensure that members of the programming team are continually aware of the memory requirements.

Plugging the Leaks Pattern

Also known as: Alloc heaven testing.

How do you ensure your program recycles memory efficiently?

- Programmers aren't perfect.
- Heap-based systems can make it easy to have memory leaks
- Systems will run happily and satisfy users even with memory leaks
- But leaks will still progressively drain free memory from the system.

Programmers aren't perfect. It's easy to make mistakes [whyMistakes], and if there's nothing that points out the mistake, very difficult to spot and correct them. An important type of such 'stealth errors' in O-O systems is called a 'memory leak': a heap-based object or data structure that's neither still required by the program, nor returned to the heap.

In C++ and languages without garbage collection this situation is very easy to achieve; allocated blocks only return to the heap if you use the memory freeing mechanism (delete in C++). If the program discard all pointers to a heap object, the object becomes 'lost' – the program will never be able to delete it and return it to the heap. In the phrase used by EPOC developers, the object goes to 'Alloc Heaven'.

[Picture showing continuous heap, with pointers from 'program' and internal, and an orphan block]

Even Java and languages with garbage collection can have memory leaks. The Garbage Collector will automatically clean up objects discarded to Alloc Heaven as above; however it's quite common to have collections still containing pointers to objects that are no longer actually required. GC can't delete these objects, so they remain – they are memory leaks.

Now programs will run happily and satisfy users even with memory leaks. Most Microsoft software, for example, appears to have minor memory leaks under some circumstances. But leaks use up valuable memory that may be needed elsewhere; leaks that are significant over the time of a given process will impact performance and push the system over it's Memory Budget.

Therefore: Test your system for memory leakage and fix the leaks.

During program testing, use tools and libraries to ensure that you detect when objects left on the heap that are no longer required. Track these 'leaks' down to their source and fix them.

Also test to ensure that your secondary storage doesn't progressively increase – unless this is a necessary feature of the system – and fix the problem if it does.

Consequences

The application uses less memory. That makes it less prone to memory exhaustion, so it is more reliable and more predictable. More memory is available for other parts of the system. Applications running in Paged systems will have better time performance.

Fixing memory leaks often solves other problems. For example a 'leaked' object may own other non-memory resources such as a file handle, or do processing when it's cleaned up. Fixing the leak will also solve these problems.

However fixing the leaks requires programmer effort.

The task of fixing the leaks provides no obvious benefit to customers of the system (since all the benefits are indirect) who may resent the time 'wasted'.

Implementation Notes

Determining whether there are Memory Leaks

The most common way to find out if there are memory leaks is to do stress testing [BeizerXXX], and use Memory Tracking tools to see if the heap memory use gradually increases. However, unless the stress testing is very rigorous, this won't find memory leaks caused by very exceptional situations (such as memory exhaustion).

Note, however, that in some environments (C++) heap fragmentation will also cause the heap memory use to increase. If memory use is increasing, but the techniques we describe below don't find any 'leaked' objects, then the problem may well be fragmentation; you can reduce it using Memory Compaction or Fixed Data Structures.

Causes of C++ Memory Leaks

The most common cause of C++ memory leaks is 'alloc heaven': the program discards all references to an object or block of memory without explicitly invoking the destructor.

The best way to fix such leaks is to identify the objects involved, and to insert the appropriate code in the correct place to delete the object. This correct place is often defined in terms of 'ownership' of the object [Ownership].

If there are many small memory leaks, you may well choose to abandon trying to fix them all piecemeal and use Memory Discard (using a scratch heap and throwing the entire heap away) instead.

Causes of Java Memory Leaks

The normal cause of memory leaks in Java is the program retaining spurious references to objects that are in fact no longer required. Ed Lycklama [Lycklama99] calls such objects 'Loiterers', and identifies and names the four most common reasons:

- Lapsed Listener – object added to collection, never removed.
- Lingerer – reference used transiently by long term object.
- Laggard – object changes state; references refer to previous state objects.
- Limbo – stack reference in a long running thread.

The normal way to fix the last three of these is either to rearrange the code so that the situation doesn't happen, or simply to set the offending reference to null at the appropriate point in the code.

Finding Memory Leaks using Checkpointing

The most common way of tracking down memory leaks is to use a memory checkpointing technique. This technique relies on you being able to define two 'checkpoints' in the code, such that all memory allocated after the first checkpoint should be freed before the second checkpoint. You then insert 'probes' (usually function calls or macros) at both checkpoints to verify that this is so.

There are two possible techniques to implement checkpointing. The most common approach is for the heap implementation to support it directly (usually only in debugging mode). Then the first probe stores the state of the heap at that point – typically creating an array of references to all the allocated blocks. And the second probe verifies that the state is unchanged – typically by checking that the set of allocated blocks is the same as before. Of course in garbage collecting languages like Java, both checkpoint functions must invoke a garbage collection before doing their heap check.

The alternative approach, if the heap implementation doesn't support checkpointing directly, is to keep a separate collection of pointers, and ensure that every new object created is added to this collection, and every object deleted is removed from it. Then the first checkpoint function creates this collection, and the second tests if it's empty.

The problem with this is how to intercept every allocation and deletion. In C++ you can implement this by implementing debugging versions of the new and delete operators. In Java you can use a weak reference collection, and replace the constructor for Object to add each object to this collection.

The simplest and most common checkpoints to chose are the start and end of the application or process. Many environments do this by default in debug mode; EPOC applications compiled in debug mode will display a fatal error ('Panic') dialog box with the memory address of objects not deallocated; Microsoft's MFC environment displays information about all remaining allocated objects to the debugging window.

Tracing the Causes of Memory Leaks in C++

The checkpointing technique above will give you the memory references of the leaked objects, but it doesn't tell you why the memory leak occurred. To find out more, you need to track down more information about the objects.

In C++, a good debugger can usually track down the class of an object with a vtbl if you know the likely base class. For example, in EPOC most heap objects derive from the class `CBase`; if the leaked object has address `0xABABABAB`, then displaying `(CBase*)0XABABABAB` will usually show you the class of the allocated object. In MFC, many objects derive from `CObject`, so you can use the same technique.

More helpful, and still simple, MFC defines a macro:

```
#define DEBUG_NEW  new( __FILE__, __LINE__ )
```

The pre-processor expands `__FILE__` and `__LINE__` macros to the file name and line number being compiled, and the corresponding debug versions of the `new` operators – `new( size_t, char*, int)` store this string and integer with each memory allocation. So if you put the following in a header file:

```
#define new DEBUG_NEW
```

Then the tracing information in the checkpointing heap dump can include all the memory items allocated.

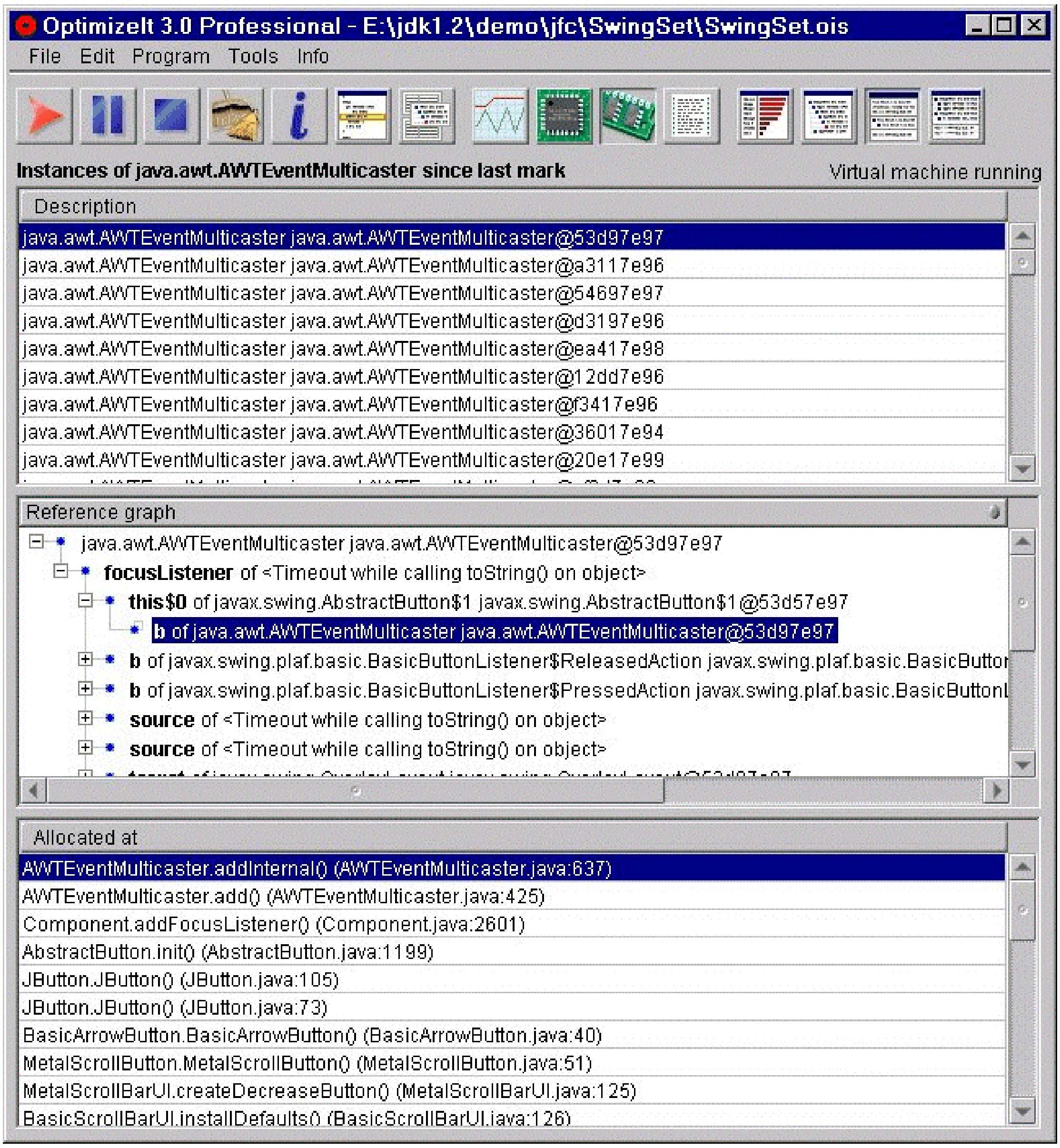
Best of all, if possible, is to use a specialised memory tracking tool, such as Purify [Rational] or BoundsChecker [NuMega]. These tools implement their own versions of the C++ heap libraries, and use debugging techniques to track memory allocation and deletion – as well as tracking invalid references to memory. They also provide programmer-friendly displays to help you track down specific memory leaks.

Tracing the Cause of Memory Leaks in Java

Having located a leaked object in Java, you need to track back to see what objects, static collections, or stack frames still have references to it.

The best way to do this is to use one of the several excellent Java memory checking tools: JProbe from KL Group [jprobe], [Apicella99]. OptimizeIt from Intuitive Systems [optimizeit], or Heap Analysis Tool from Sun [hat]. No doubt others will be available shortly.

Both JProbe and OptimizeIt work by modifying the Java kernel to provide better debug heap management, and provide very user-friendly GUIs to help debugging – see Figure 6, for example; HAT uses a detailed knowledge of Sun's own heap implementation.


OptimizeIt 3.0 Professional - E:\jdk1.2\demo\jfc\SwingSet\SwingSet.ois
File Edit Program Tools Info
Virtual machine running
Description
Reference graph
Allocated at

Instances of java.awt.AWTEventMulticaster since last mark

java.awt.AWTEventMulticaster java.awt.AWTEventMulticaster@53d97e97
java.awt.AWTEventMulticaster java.awt.AWTEventMulticaster@a3117e96
java.awt.AWTEventMulticaster java.awt.AWTEventMulticaster@54697e97
java.awt.AWTEventMulticaster java.awt.AWTEventMulticaster@d3197e96
java.awt.AWTEventMulticaster java.awt.AWTEventMulticaster@ea417e98
java.awt.AWTEventMulticaster java.awt.AWTEventMulticaster@12dd7e96
java.awt.AWTEventMulticaster java.awt.AWTEventMulticaster@f3417e96
java.awt.AWTEventMulticaster java.awt.AWTEventMulticaster@36017e94
java.awt.AWTEventMulticaster java.awt.AWTEventMulticaster@20e17e99

```
⊟ java.awt.AWTEventMulticaster java.awt.AWTEventMulticaster@53d97e97
  ⊟ focusListener of <Timeout while calling toString() on object>
    ⊟ this$0 of javax.swing.AbstractButton$1 javax.swing.AbstractButton$1@53d57e97
      └ b of java.awt.AWTEventMulticaster java.awt.AWTEventMulticaster@53d97e97
    ⊞ b of javax.swing.plaf.basic.BasicButtonListener$ReleasedAction javax.swing.plaf.basic.BasicButtor
    ⊞ b of javax.swing.plaf.basic.BasicButtonListener$PressedAction javax.swing.plaf.basic.BasicButtonL
    ⊞ source of <Timeout while calling toString() on object>
    ⊞ source of <Timeout while calling toString() on object>
```

AWTEventMulticaster.addInternal() (AWTEventMulticaster.java:637)
AWTEventMulticaster.add() (AWTEventMulticaster.java:425)
Component.addFocusListener() (Component.java:2601)
AbstractButton.init() (AbstractButton.java:1199)
JButton.JButton() (JButton.java:105)
JButton.JButton() (JButton.java:73)
BasicArrowButton.BasicArrowButton() (BasicArrowButton.java:40)
MetalScrollButton.MetalScrollButton() (MetalScrollButton.java:51)
MetalScrollBarUI.createDecreaseButton() (MetalScrollBarUI.java:125)
BasicScrollBarUI.installDefaults() (BasicScrollBarUI.java:126)

Figure 6: Tracking references to a Java object using OptimizeIt

Examples

EPOC Exhaustion Test Extension

In most C++ applications, the most likely cause of a memory leak is an exception that destroys stack pointers to heap memory. As discussed in the Exhaustion Test pattern, such memory leaks are difficult to simulate by normal testing but easy with Exhaustion Testing.

This example extends the example code we quoted in the Exhaustion Test pattern to check for memory leaks during each test. It uses the EPOC pair of macros __UHEAP_MARK and __UHEAP_MARKEND. In debug mode, these check that the number of cells allocated at MARKEND is the same as at the corresponding MARK, and display a message to the debug output if not. The pairs of macros can be nested. __UHEAP_MARKEND also checks the heap for consistency.

```
static void MemTest(void (*aF)())
   {
   __UHEAP_MARK;
   for (int iteration=0; ;++iteration)
        {
        __UHEAP_FAILNEXT( iteration );
        __UHEAP_MARK;
        TRAPD(error,(*aF)());  // Equivalent to try...catch...
        __UHEAP_MARKEND;
        if (error==KErrNone)   // Completed without error?
             {
             test.Printf(_L("\r\n"));
             break;
             }
        else
             {
             test.Printf(_L("  --  Failed on %d\r\n"),iteration);
             }
        }
   __UHEAP_MARKEND;
   __UHEAP_RESET;
   }
```

MFC Example

The following pieces of code implement checkpointing using the Microsoft MFC class, CMemoryState. The first checkpoint (corresponding to EPOC's __UHEAP_MARK above) stores the current state:

```
#ifndef NDEBUG
CMemoryState oldState;
oldState.Checkpoint();
#endif
```

The second checkpoint (corresponding to EPOC's __UHEAP_CHECK) checks for differences and dumps details of all the changed objects if it finds any:

```
#ifndef NDEBUG
CMemoryState newState, diffState;
newState.Checkpoint();
if (diffState.Difference( oldState, newState ))
{
   TRACE( "Memory loss - file %s, line %d\n", file, line );
   diffState.DumpStatistics();
   diffState.DumpAllObjectsSince();
}
#endif
```

Known Uses

Plugging the Leaks is a standard part of virtually every project with memory constraints.

See Also

Exhaustion Testing often shows up leaks to plug.

If there are many difficult-to-find small leaks that don't really impact the system in the short term, it can be much easier to use Program Chaining – terminating and restarting a process regularly – instead of Plugging the Leaks.

Exhaustion Test Pattern

Also known as: Out-of-memory testing.

How do you ensure that your programs work correctly in out of memory conditions?

- Functionality that isn't tested probably won't work correctly.
- The functionality of a system includes what it does when it runs out of memory.
- It's difficult to make a program run out of memory on a normal system.
- It can be difficult to reproduce errors caused by limited memory.

Programs that deal gracefully with resource failures — say by using the **PARTIAL FAILURE** or Fixed Memory patterns — have a large number of extra situations to deal with, because each resource failure is a different execution event.

Ideally, you'll test the program in every situation that will arise in execution. But testing for memory exhaustion failures is particularly difficult because these events are by definition exceptional, so they will occur mostly when the system is heavily loaded or has been executing for a long period.

You might create out-of-memory situations by allocating a lot of memory so that the system is resource-strapped and the errors do happen. This has two problems. First it's difficult to get exactly the same situation each time. So it will be difficult to reproduce any errors you have. Secondly, in many environments you'll be running your debugging and possibly development systems in the same environment (and maybe processes belonging to other users of the system too). Forcing the system to be short of memory will prevent these tools from working correctly as well.

Therefore: *Use testing techniques that simulate memory exhaustion.*

Use a version of the memory allocator that fails after a given number of allocations, and verify that the program behaves sanely for all values of this number. Also use another version of the allocator that inserts random failures. Verify that the program implements **PARTIAL FAILURE** by taking alternative steps to get the job done, or **MAKES THE USER WORRY** by reporting to the user if this is not possible.

You can combine partial failure testing with more traditional memory testing techniques such as the use of conservative garbage collectors to verify that the program does not cause resource leaks.

Consequences

Using specialised testing techniques reduces the *testing cost* for a given amount of trust in the program.

It will be easier to replicate the errors, making it easy to debug the problems and verify any fixes. Which reduces the total programmer effort required to get the system working.

However you'll still need a significant testing cost to be reasonably certain that the resulting program will work correctly. This approach also needs *programmer effort* to build the specialised memory allocators to support the tests.

Testing doesn't always detect the results of random and time-dependent behaviour – for example where two threads are both allocating independently.

Implementation Notes

Simulating memory failure

Some systems, such as EPOC and the Purify tool for UNIX and Windows environments, provide a version of the memory allocator that can be set to fail either after a specified number of allocations or randomly. For other systems (such as MFC), it's easy to implement this in C++ by redefining the `new` operator.

Here's an example for the MFC environment. Note that it uses the `DEBUG_NEW` macro, which provides debugging information about allocated blocks (see Plugging the Leaks).

```
#ifdef MEMORY_FAILURE_TEST
BOOL MemoryFailureTime();
# define new MemoryFailureTime() ? (AfxThrowMemoryException(),0) : DEBUG_NEW
#else
# define new DEBUG_NEW
#endif

#ifdef MEMORY_FAILURE_TEST
static int allocationsBeforeFailure;
BOOL MemoryFailureTime()
{
    return allocationsBeforeFailure-- == 0;
}
#endif

BOOL TestApp::InitInstance()
{
#ifdef MEMORY_FAILURE_TEST
   allocationsBeforeFailure = atoi( m_lpCmdLine );
#endif
// ... etc.
```

An alternative is to implement a debug version of the function ::operator new(size_t) with similar behaviour.

Simulating memory failure by external actions

There are two approaches to setting up the test. One common approach is to provide a mechanism where user input to the application or system to cause it to fail. In the example above, the user passes an integer as the command line; allocation will fail after that number of memory allocations.

As an alternative the application might put up a dialog ("Will give allocation failure after how many allocations?") where the user enters a number. Then heap allocation fails after that number of allocations. In EPOC and Purify, this facility is built into the runtime debugging environment. In EPOC, for example, the keystroke XXX brings up this dialog.

The advantage of this approach is that it makes it easy to debug a particular situation without needing to perform all the other tests. It also works well with almost the entire system running normally. The disadvantage however is that for complete testing, the user will have to run the system a very large number of times, entering a different value each time. While it might be possible to automate this process by using external tools to simulate user input, this would be bound to be cumbersome and slow.

Repeatedly simulating memory failure using test harnesses

The alternative approach is to write a test harness that runs a particular function or piece of code repeatedly, with different values of the 'allocations until failure' value.

This approach is mandatory in a system with very strong test requirements, or one using the 'Extreme Programming' approach where every test remains as part of the development environment and all tests are run early and often.

This approach provides much more complete testing of the given functionality. But it's only realistic for specific functions, and ones that require no user or external input to execute.

Example

The following is a (much simplified) extract from the test code for part of an EPOC application – a class we'll call CApplicationEngine.

It uses the heap debugging facility, User::__DbgSetAllocFail(), to set up the application (`EUser`) heap for 'deterministic failure' – i.e. failure after a specific number of allocations.

The function MemTest simply calls a given function many times, so that memory fails after progressively more and more allocations. On EPOC memory failure, the code always does a 'Leave' (EPOC Exception – see Partial Failure), which the code can catch using the macro TRAPD. The test support RTest class simply provides output to the tester and to a test log.

```
RTest test;

static void MemTest(void (*aF)())
   {
   for (int iteration=0; ;++iteration)
        {
        __UHEAP_FAILNEXT( iteration );
        TRAPD(error,(*aF)());  // Equivalent to try...catch...
        if (error==KErrNone)   // Completed without error?
             {
             test.Printf(_L("\r\n"));
             break;
             }
        else
             {
             test.Printf(_L("  --  Failed on %d\r\n"),iteration);
             }
        }
   __UHEAP_RESET;
   }
```

The main function, DoTests(), calls MemTest for each testable function in turn:

```
static void DoTests()
   {
   test.Start(_L("Sheet engine construction"));
   MemTest(Test1);
   test.Next(_L("Test set and read on multiple sheets"));
   MemTest(Test2);
   // ... etc.
   test.End();
   }
```

And a typical testable function might be as follows:

```
static void Test1()
   {
   CApplicationEngine* theApplicationEngine = CApplicationEngine::NewLC();
   theApplicationEngine->SetRecalculationToBeDoneInBackgroundL(EFalse);
   CleanupStack::PopAndDestroy();  // theApplicationEngine
   }
```

Known Uses

Exhaustion testing is a standard part of the development of every component and application released by Symbian. The EPOC environment demands **PARTIAL FAILURE**, so each possible failure mode is a part of the application's functionality.

The Purify environment for C++ supports random and predictable failure of C++'s memory allocator. Other development environments provide tools or libraries to allow similar memory failure testing.

Symbian's EPOC provides debug versions of the allocator with similar features and these are used in module testing of all the EPOC applications.

See Also

Exhaustion Testing is particularly important where the system has specific processing to handle low memory conditions, such as the **PARTIAL FAILURE** and **CAPTAIN OATES** patterns.

C++ doesn't support garbage collection and normally throws an exception on allocation failure, so the most likely consequence of incorrect handling of allocation failure is a Memory Leak as the exception looses stack pointers to allocated memory. So C++ Exhaustion Testing is usually combined with Plugging the Leaks.

References

[Noble+Weir98] *Proceedings of the Memory Preservation Society - Patterns for Small* Machines James Noble. Charles Weir Proceedings of the conference EuroPLoP 1998

[Beizer84] *Software System Testing and Quality Assurance,* Boris Beizer, Van Nostrand Reinhold 1984, 0-442-21306-9

[EPOCSDK]

[whyMistakes] Something about the brain's stack of seven items. Tony Buzan? Use your head.

[Weir96] *Improve your Sense of Ownership*– Charles Weir, ROAD magazine, March 1996. http://www.cix.co.uk/~cweir/papers/owner.ps

[Apicella99] JProbe Suite 2.0 takes the kinks out of your Java programs, InfoWorld May 17, 1999 (Vol. 21, Issue 20) http://www.infoworld.com/archives/html/97-e04-20.78.htm

[OptimizeIt] http://www.optimizeit.com/

[Jprobe] www.klgroup.com

[sun] www.sun.com

[Lycklama99] *Memory Leaks in Java*, Ed Lycklama, Presentation at Java One 1999. Available at http://www.klgroup.com/jprobe/javaonepres/

[Regclean] A memory assessment of MS Regclean in the risks digest http://catless.ncl.ac.uk/Risks/20.37.html

[Brooks75] The Mythical Man Month.

[Blank+95] Blank and Galley (1995) *How to Fit a Large Program Into a Small Machine*, available as http://www.csd.uwo.ca/Infocom/Articles/small.html

[MicrosoftSpy97] Microsoft Visual C++ 5.0 Spy++ Online Help Documentation, Microsoft 1997.

[Gilb88] Principles of Software Engineering, Tom Gilb, Addison Wesley 1988, 0-201-19246-2

[Flanders&SwanXX] The laws of thermodynamics (song). In 'the collected works of F&S',

DeMarco Project Planning?

[Filipovitch96] Project Management: Introduction, A.J.Filipovitch 1996, http://krypton.mankato.msus.edu/~tony/courses/604/PERT.html

[Baker&Baker98] Complete Idiot's Guide to Project Management, by Sunny Baker, Kim Baker, MacMillan 1998; ISBN: 0028617452

[Kruekeberg&Silvers74] KRUECKEBERG, D.A. & A.L. SILVERS. 1974. "Program Scheduling," pp. 231-255 in Urban Planning Analysis: Methods and Models. NY: John Wiley & Sons, Inc.

[Brooks75] The Mythical Man Month.

[PalmBudget] ID 1136 Palm web.

[Shaw&Garlan] *Software Architecture - Perspectives on a Emerging Discipline*, Shaw and Garlan, Prentice Hall ISBN 0-13-182957-2

[XP] Extreme Programming Explained, Kent Beck, Addison-Wesley 2000, 201-64141-6

Software Requirements and Specifications by Michael Jackson

Peopleware Tom DeMarco

[Adams95] Functionality a la Carte, S.S.Adams, Patterns for Programming Design 1, 7-8.

[Rappel] *to add in text*

[Episodes] to add in text

User Involvement
Techniques for Handling Memory Constraints in the UI

Abstract:

This paper describes some UI design patterns to use when creating software to run in limited memory.

It is a draft version of a chapter to add to the authors' book Small Memory Software, and follows the structure of other chapters in that book.

Major Technique: User Involvement

How can you manage memory in an unpredictable interactive system?

- Memory requirements can depend on the way users interact with the system.
- If you allocate memory conservatively, the systems functionality may be constrained.
- If you allocate memory aggressively, the system may run out of memory.
- The system needs to be able to support different users, who will use the system in quite different ways.
- Users using the system need to perform a number of different tasks, and each task has different memory requirements.
- The system may have to run efficiently on hardware with greatly varying physical memory resources.
- The system's functionality is more important that its simplicity.

In many cases, especially in interactive systems, memory requirements cannot really be predicted in advance. For example, the memory requirements for the Strap-It-On PC's word-processing application Word-O-Matic will vary greatly, depending the features users choose to exercise — once user may want voice output, while another a large font for file editing.

The memory demands of interactive systems are unpredictable because they depend critically on what users choose to do with the system. If you try to produce a generic memory budget, you will over-allocate the memory requirements for some parts of the program, and consequently have to under-allocate memory for others.

For many interactive systems, providing the necessary functionality is more important than making the functionality easy to learn or to use. Being able to use a system to do a variety of jobs without running out of memory is sufficiently important that you can risk making other aspects of the interface design more complicated if it makes this possible. This is especially important because a system's users presumably know how they will use the system when they are actually using it, even through the system's designer may not no know this ahead of time.

Therefore: *Make the system's memory model explicit in its user interface, so that the user makes their own decisions about memory.*

Design a conceptual model of the way the system will use memory. Ideally, this model should be based on the conceptual model of the system and the domain model, but providing extra information about the system's memory use. This model should be expressed in terms of the objects users manipulate, and the operations they can perform on those objects, rather than the objects used directly in the implementation of the system.

Expose this memory model in your program's user interface, and let users manage memory allocation directly — either in the way that they create and store user interface level objects, or more coarsely, balancing memory use between their objects and the program's internal memory requirements.

For example, Word-O-Matic allows the user to choose how much memory should be allocated to store clip-art, and uses all the otherwise unallocated space to store the document. Word-O-Matic also displays the available memory to the user.

Consequences

The system can deliver more behaviour to the user than if it had to make pessimistic assumptions about its use of memory. The user can adjust their use of the system to make the most of the available memory, reducing the memory requirements for performing any particular task. Although the way user memory will be allocated at runtime is unpredictable, it is quarantined within the Memory Budget, so the memory use of the system as a whole is more predictable. Some user interfaces can even User Involvement about *memory fragmentation*.

However: Users now have to worry about memory whether they want to or not, so the system is less usable. Worrying about memory complicates the design of the system and its interface, making it more confusing to users, and distracting them from their primary task. Given a choice, users will choose systems where they do not have to worry about memory. You have to spend programmer effort designing the conceptual model, and making the memory model visible to the user.

Implementation

There are number of techniques which can expose a system's memory model to its users:

- Constantly display the amount of free memory in the system.
- Provide tools that allow users to query the contents of their memory, and the amount of memory remaining.
- Generate warning messages or dialogue boxes as the system runs out of memory, or as the user allocates lots of data.
- Make the user choose what data to overwrite or delete when they need more memory.
- Show the memory usage of different components in the system.
- Tell the user how their actions and choices affect the system's memory requirements.

Conceptual Models and Interface Design

A conceptual model is not the same as an interface design — rather, it is an abstraction of an interface design (see Constantine & Lockwood 1999, Software for Use). Where an interface design describes the way an interface looks and behaves in detail, a conceptual model describes the objects that should be present on a given part of an interface, the information those objects need to convey to users, and the operations users should be able to carry out upon those objects. To design an interface that presents memory constraints to the user, you should first determine what information about memory interface needs to present and how that information is related to the existing information managed by the user interface, and only then consider how to design the appearance and behaviour of the interface to present the information about memory use.

Granularity of Modelling

Conceptual models of memory use can be built with varying degrees of sophistication and differing amounts of detail. A very coarse model — perhaps modelling only users' and the system's memory consumption — will lead to a very simple interface that is easy to operate but may not provide enough information to make good decisions about memory use. A more fine grained model, perhaps associating memory use figures with every domain object in the interface, will give more control to users, but be more complex and more difficult to operate. Very detailed models that reify internal system components as concepts in the interface (so that the "font renderer" or "file cache" are themselves objects that users can manipulate) can provide more control, at a cost of further increase the complexity of the application.

Static and Dynamic Modelling

Conceptual models of memory use can be made statically or dynamically. Static models do not depends on details of a particular program run, so they can embody choices mad as the system was designed, or configuration parameters applied as the system begins running. In contrast, dynamic decisions are made as the program is running. A conceptual model of a system's memory use may describe static parameters, dynamic parameters, or a mixture of both. Generally static models makes it easier to give guarantees about a system's behaviour, because they cannot depend on the details of a particular run of a system. In contrast, precisely because they may depend on differences between runs, dynamic models can be more flexible, changing during the lifetime of the system to suit the way it is used, but also run the risk of running out of memory.

General Purpose Systems

The more general a system's purpose, the more difficult memory allocation becomes. A system may have to support several radically different types of users – say from novices to experts, or from those working on small jobs to those working on big jobs. Even the work of a single user can have different memory requirements depending upon the details of the task performed: formatting and rasterising text for laser printing may have completely different memory requirements to entering the text in the first place. Also, systems may need to run on hardware with varying memory requirements. Often the memory supplied between different models or configurations of the same hardware can vary by a several orders of magnitude and the same program may need to run on systems with 128Kb of memory to systems with 128M or more.

Supporting Different User Roles

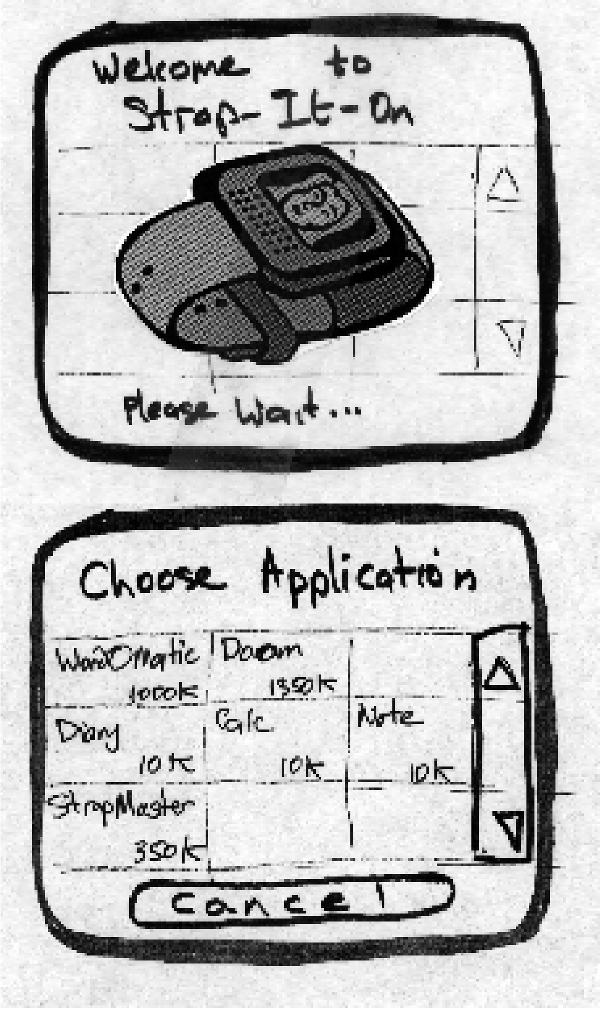
Different users can differ widely in the roles they play with respect to a given system, and often their memory use (and interest in or capability to manage the system's memory use) depends upon the role they play. For example, the users of a web-based information kiosk system would play two main roles with respect to the system — a casual inquirer trying to obtain information from the kiosk, and the kiosk administrator configuring the kiosk, choosing networking protocol addresses, font sizes, image resolutions and so on. The casual inquirer would have no interest in the system's model of memory use, and no background or training to understand or manipulate it, while the kiosk administrator could be vitally concerned with memory issues.

The techniques and processes of User Role Modelling from Usage-Centered Design (Constantine & Lockwood, 1999) can be used to identify the different kinds of users a system needs to support, and to characterise the support a system needs to provide to each kind of user.

Examples

For example, after it has displayed its startup screen, the Strap-It-On wrist mounted PC asks its user to select an application to run. The select an application screen also displays the amount of memory each application will need if it is chosen.

[note: all scanned pictures to be redrawn].



Specialised Patterns

The rest of this chapter contains five patterns that present a range of techniques for making the user worry about the systems memory use. It describes ways that a user interface can be structured, how users can be placed directly in control of a system's memory allocation, anddescribes how the quality of a user interface can be traded off against its memory use.

FIXED SIZED USER MEMORY describes how user interfaces can be designed with a small number of user memories. Controls to access these memories can be designed directly into the interface of the program, making them quick and easy to access. Fixed sized user memories have the disadvantages that they do not deal well with user data objects of varying sizes, or more than about twenty memory locations.

VARIABLE SIZED USER MEMORY allows the user to store variable numbers of varying sized data objects, overcoming the major disadvantages of designs based on fixed sized user memory. The resulting interface designs are more complex than those based on fixed sized memory spaces, because users need ways of navigating through the contents of the memories and must be careful not to exhaust the capacity.

MEMORY FEEDBACK, in turn, addresses some of the problems of variable sized user memory: by presenting users with feedback describing the state of a system's memory use, they can make better use of the available memory. Providing memory feedback has a wider applicability than just managing user memory, as the feedback can also describe the system's use of memory — the amount of memory occupied by application software and system services.

USER MEMORY CONFIGURATION extends Memory Feedback by allowing users to configure the way systems use memory. Often, information or advice about how a system will be used, or what aspects of a system's performance are most important to its users, can help a system make the best use of the available memory.

Finally, **LOW QUALITY MULTIMEDIA** describes how multimedia resources — a particularly memory-hungry component of many systems — can be reduced in quality or even eliminated altogether, thus releasing the memory they would otherwise have occupied for more important uses in the system.

See Also

The memory model exposed to the user may be implemented by **FIXED ALLOCATION or VARIABLE ALLOCATION — FIXED SIZE USER MEMORY** is usually implemented by **FIXED ALLOCATION** and **VARIABLE SIZE USER MEMORY** is usually implemented by **VARIABLE ALLOCATION.**

FUNCTIONALITY A LA CARTE [Adams 95] can present the costs and benefits of memory allocations to the user.

A static **MEMORY BUDGET** can provide an alternative to **USER MEMORY CONFIGURATION** that does not require users to manage memory explicitly, but that will have higher memory requirements to provide a given amount of functionality.

The patterns in this chapter describe techniques for designing user interfaces for systems that have limited memory capacity. We have not attempted to address the must wider question of user interface design generally — as this is a topic which deserves a book of its own. Schneiderman's *User Interface Design* is a general introduction to the field of interface design, and Constantine and Lockwoods' *Software for Use* presents a comprehensive methodology for incorporating user interface design into development processes. Both these texts discuss interface design for embedded systems and small portable devices as well as for desktop applications.

Fixed Size User Memory

Also known As: Fixed Number of User Memories

How can you present a small amount of memory to the user?

- You have a small amount of user-visible memory.
- Users need to store a small number of discrete items in the memory
- Every item users need to store is roughly the same size
- Users need to be able to retrieve data from the memory particularly easily.
- Users cannot tolerate much extra complexity in the interface.

Some systems have only a small amount of memory available for storing the users' data (and presumably only a small amount of data that users can store). This user data is often a series of discrete items — such as telephone numbers or configuration settings where each item is the same size. For example, the Strap-It-On needs to definitions for its voice input feature. Each macro requires enough memory to recognise a three second spoken phrase of the user's choice, and the commands that are to be executed when the voice macro facility recognises that phrase.

Users need to be able to retrieve data from memory quickly and easily — after all, that's why the system is going to the trouble to store such a small amount of data. For example, the point of the Strap-it-On's voice input macros are to make data entry more efficient, streamlined, and "fun to do all day" (to quote the marketing brochure). Similarly music synthesisers store multiple 'patches' so that they can be quickly recalled during a performance, and phones store numbers because people want to dial them quickly.

One approach to this problem is to let the user choose things to store, until the device is out of memory, when it stops accepting things. This is quite easy to implement but gets pretty unsatisfactory when there's only a small amount of memory. Users will tend to get the idea that the device has infinitude of memory, and consequently will be surprised when the system refuses their requests. Also, you'll need some kind of interface to retrieve things from the memory, to delete things that have already been stored, and so on — all of which will just take up more precious memory space.

Therefore: *Provide a small, fixed number of memory spaces, and let the user manage them individually.*

Design a fixed number of "user memories" as explicit parts of the user interface conceptual model. Each user memory should be represented visually as part of the interface, and the design should make clear that there are only a fixed number of user memory spaces — typically by allocating a single interface element (often a physical or virtual button) for each memory. Ideally each memory space should be accessed directly via its button (or via a sequence number) to reinforce the idea that there are only a fixed number of user memories.

The user can store and retrieve items from a memory space by interacting with the interface element(s) that represent that user memory. Ideally the simplest interaction, such as pressing the button that represents a user memory, retrieves the contents of the memory — restoring the device to the configuration stored in the memory, running the macro, or dialling the number held in that memory. Storing into a user memory can be a more complex interaction, because storing is performed much less frequently than retrieval. For example, pressing a “store” button and then pressing the memory button might store the current configuration into that memory. Any other action that uses the memory should access it in the same way.

Finally, an interface with a fixed number of user memories does not need to support an explicit delete action from the memories: the user simply chooses which memory to overwrite.

For example, the Strap-It-On allocates enough storage for nine voice macros. This storage is always available (it is allocated permanently in the memory budget; the setup screen is quickly accessible via the Strap-It-On's operating system, and is designed to show only the nine memory spaces available.

Consequences

The idea that the system has a number of memory spaces into which users can store data is obvious from the its design, making the design *easy to learn*. The fixed number of user memory spaces becomes part of users’ conceptual model of the system, and the amount of memory available is always clear to users. Because the number of memories is fixed, the interface can be designed so that users *can easily and quickly* choose which memory to retrieve. The *graphical layout* of the interface is made easier, because there are always the same number of memories to display.

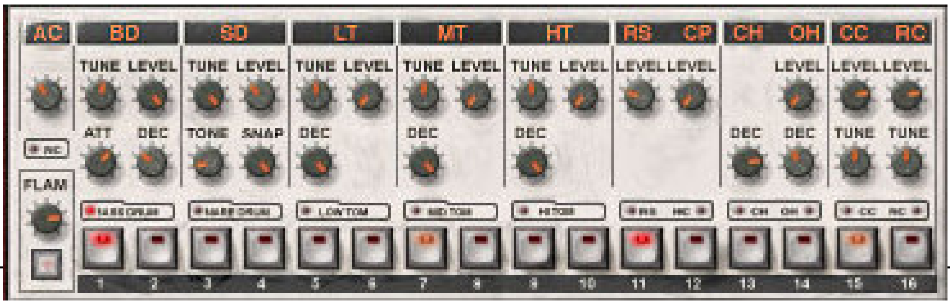
However: The user interface architecture is strongly governed by the memory architecture. This technique works well for a small number of memory spaces but *does not scale* well to allocating more than twenty or thirty memory spaces, or storing objects of more than two or three different sizes. User interfaces based on a fixed number of user memories are generally less *scalable* than interfaces based on some kind of variable allocation. Increasing the size of a variable memory may simply require increasing the capacity of a browser or list view, but increasing the number of fixed user memories can require a redesign of the interface, especially if memories are accessed directly.

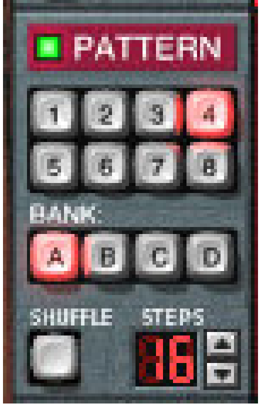
Implementation

There are three main interface design techniques that can be used to access fixed size user memories — *direct access, banked access, and sequential access*. This illustrates the difference between a conceptual model and in interface design — the same conceptual model for fixed sized user memories can be realised in several different ways in an actual user interface design.

Direct Access. For a small number of user memories, allocate a single interface element to each memory. With a single button for each memory, pressing the button can recall the memory directly — a fast and easy operation. Unfortunately, this technique is limited to sixteen or so memories because few interfaces can afford to dedicate too many elements to memory access.

For example, the drum machines in the ReBirth-338 software synthesiser provide a fixed size user memory to store drum patterns. The sixteen buttons across the bottom of the picture below correspond to sixteen memory locations storing sixteen drum beats making up a single


AC
BD
SD
LT
MT
HT
RS
CP
CH
OH
CC
RC
TUNE LEVEL TUNE LEVEL TUNE LEVEL TUNE LEVEL TUNE LEVEL LEVEL LEVEL LEVEL LEVEL LEVEL
ATT DEC TONE SNAP DEC DEC DEC DEC DEC TUNE TUNE
FLAM
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16



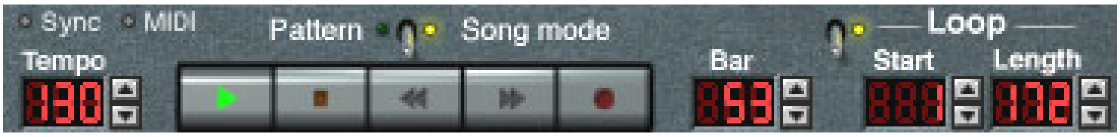
pattern — we can see that the bass drum will play on the first, seventh, eleventh and fifteenth beat of the pattern.

Banked Access. For between ten and one hundred elements, you can use a two dimensional scheme. Two sets of buttons are used to access each memory location — the first set to select a memory bank, and the second set to select an individual memory within the selected bank. Banked access requires many fewer interface elements than direct access given the same number of memory spaces, but is still quite quick to operate.

Again following hardware design, the ReBirth synthesiser can store thirty-two patterns for each instrument in emulates. The patterns are divided up into four banks (A, B, C, D) of eight patterns each, and these patterns are selected using banked access.

Sequential Access. For even less hardware cost (or screen real estate) you can get by with just a couple of buttons to scroll sequentially through all the available memory spaces. This approach is quite common on cheap or small devices because it has a very low hardware cost and can provide access to an unlimited number of memories. However, sequential scrolling is more difficult to use than direct access, because more user gestures are required to access a given memory location, and because the memory model is not as explicit in the interface.

A ReBirth song is made up of a list of patterns. The control below plays through a stored song. To store a pattern into a song, you select the position in the song using the small arrow controls to the left of the “Bar” window, and then choose a pattern using the pattern selector shown above.



Single Memories

Special cases of fixed sized user memory are systems that provide just one memory space. For example, many telephones have a "last number redial" feature — effectively a single user memory set after every number is dialled. Similarly, many laptop computers have a single memory space for storing backup software configuration parameters, so that the machine can be rebooted if it is misconfigured. A single memory space is usually quick and easy to access, but obviously cannot store much information.

Memory Exhaustion

One advantage of fixed size user memory designs is that users should never experience running out of memory — rather, they will just have to decide which user memory to overwrite. Certainly, if all memories (either full or empty) are accessed in the same way, the system never needs to produce any error messages explaining that the system has run out of memory.

Storing Variable Sized Objects

A fixed number of fixed sized memory locations does not cope well when storing objects of varying sizes. If an item to be stored is too small for the memory space, the extra space is wasted. If an item is too large, either it must be truncated, or it must be stored in two or more spaces (presumably wasting memory in the last overflow space), or the user must be prevented from creating such an item in the first place.

Alternatively, if an interface needs to store just two or three different kinds of user data objects (where each kind of object has a different size) the interface design can have a separate set of user memories for each kind of object that needs to be stored. This doesn't avoid the problem completely, since the size and number of the memory spaces must be determined in advance, and it is unlikely that it will match the number of objects of the appropriate kind that each user wishes to store.

Initialising Memories

An important distinction in the design of conceptual models for fixed size user interfaces is whether the system supports a fixed sized number of *memory spaces* or a fixed sized number of *objects*. The differences is that if the system as a number of memory spaces, some of the spaces can be empty, but it doesn't make sense to have a empty object stored in a memory space. In general, designing in terms of objects is preferable to designing in terms of memory spaces. For example, there is no need to support retrieve operations on empty spaces if there can be no empty spaces, For this to work, you need to find good initial contents for the objects to be stored. The memory spaces of synthesisers and drum machines, for example, are typically initialised with useful sounds or drum patterns than can be used immediately, and later overwritten.

One compensating advantage of having the idea of empty memories in a conceptual model is that you can support an *implicit store* operation that stores an object into some empty memory space, without the user having to chose the space explicitly. This certainly makes the store operation easier, but (unlike a store operation that explicitly selects a memory space to overwrite), and implicit store operation can fail due to lack of memory — effectively treating the fixed size memories as if they were variable sized. The Nokia 2210e mobile phone supports implicit stores into its internal phone book, but, if there are no empty memories, users can choose which memory to overwrite.

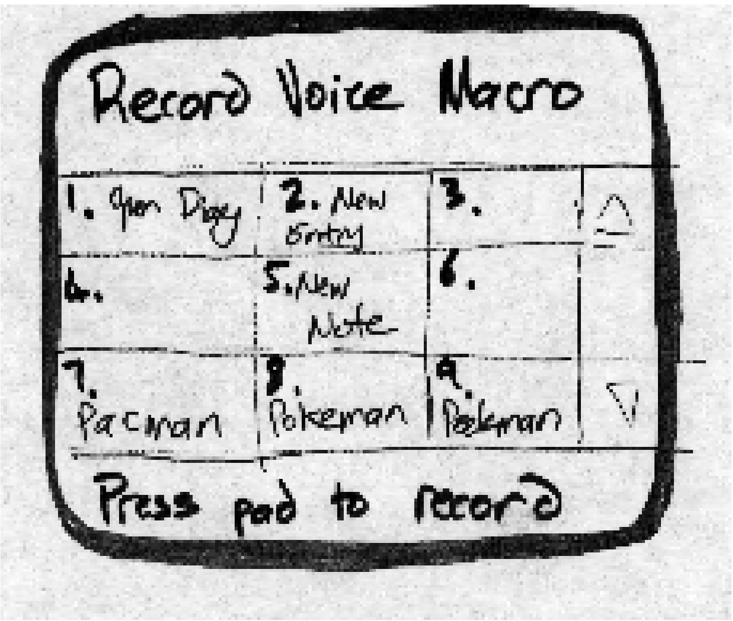
Naming Memories

Where there are more than four or five memories — and certainly where there are more than sixteen or so — you can consider allowing the user to name each memory to make it easier for users to remember what is store in each memory space. A memory name can be quite short — say eight uppercase characters — and so can be stored in a small amount of memory using simple **STRING COMPRESSION** techniques.

Of course, there is still a trade-off between the *memory* requirements for storing the name and the *usability* of a larger number memories, but there is no user providing a system with large numbers of memories if users can't actually find the things they have store in them. If the system includes a larger amount of preset data in stored in **READ-ONLY STORAGE** you can supply names for these memories, while avoiding naming user memories stored in scarce RAM or writeable persistent storage.

Examples

The StrapItOn PC has nine memories that can be used to store voice input macros. Each memory is represented by one of nine large touch-sensitive areas on the StrapItOn's screen. To record a macro, the user touches the are representing the memory where they want to store the macro — if this memory is already in use, the existing macro is overwritten.



The Korg WaveStation synthesiser contains several examples of fixed sized user memories. The most obviously is that its user interface includes five soft keys that can be programmed by the user to move to a particular page in the WaveStation's menu hierarchy. The soft keys are accessed by pressing a dedicated "jump" key on the front panel, and then one of the five physical keys under the display.

The main memory architecture of the WaveStation is also based firmly on fixed sized user memories. Each WaveStation can store thirty-two 'performances', that can refer to up to four 'patches', each of which can play one of thirty-one 'wave sequences'. Patches and performances have different sizes, and each are stored in their own fixed-sized user memories, so if you run out of patch storage, you cannot utilise empty performance memories. Patch and performance memories are addressed explicitly using memory location numbers entered by either cursor keys, turning a data-entry knob, or typing the number directly using a numeric keypad. Patches and performances can also be named --- performances with up to sixteen characters, patches with only ten.

Each wave sequence may refer to up to two hundred waves, however the total number of waves referred to by all wave sequences cannot exceed five hundred. (Approximately five hundred waves are stored in ROM, and a wave sequence describes an order in which the ROM waves should be played). Waves are added implicitly to wave sequences using the "enter" key --- if either of the limits on individual or total wave sequence lengths is exceeded, the WaveStation displays an :"Out of Memory" error message (see the **MEMORY NOTIFIER** pattern.)

[Picture to be drawn from manual -- currently en route from Sydney to Wellington. Numbers to be checked against the manual!]

Known Uses

Music others synthesisers often provide a small fixed number of memory locations for programs, and users think of these systems as having just that number of memories. For example, the Yamaha TX-81Z synthesiser provided 128 preset patches (synthesiser sound programs) organised as four banks of 16 patches each, plus one bank of 16 user-programmable patches. The TX-81Z also included 16 memories for “performances” including references to patches but also information about the global state of the synthesiser (such as MIDI channel and global controller assignments) — performances were user data objects of different sizes to the patches. The TX-81Z also included some other user memories for other things, such as microtonal scales and delay effects.

GSM mobile phone SIM cards are smart cards that store a fixed number of phone memories containing name and number (the number of memories depends on the SIM variant). Users access the memories by number. Many telephones have something similar, though simpler systems don't store names. The same SIM cards can also store a fixed number of received SMS (short message service) text messages — the user is told if a message could not be stored because the store overflowed. SIM cards can also store a fixed number of already read messages in a numbered store. This store is visible to the user and accessed by message number.

The FORTH programming environment stored source code in 1024 character blocks. This allowed the system to allocated a fixed sized buffer for the text editor, made screen layout simple (16 lines of 64 characters that could be displayed on a domestic TV screen) and to store each block in a single 1024 byte sector on a floppy disc. Each block on disc was directly referenced by its sequence number.

An early Australian laser printer required the user to uncompress typefaces into one of a fixed number of memory locations. For example, making Times Roman, Times Roman Italic, and Times Roman Bold typefaces available for printing would require three memory locations into which ROM Times Roman bitmaps could be uncompressed, with some bitmap manipulations to get italic and bold effects. Documents selected typefaces using escapes codes referring to memory locations. Larger fonts had to be stored into two or more contiguous locations, making the user worry about memory fragmentation as well as memory requirements, and giving very interesting results if an escape code tried to print from the second half of a font stored in two locations.

Many video games store just the top 10 scores (and the three initials of the players who scored them). This has a number of advantages: it requires very little memory, allows a constant graphical layout for the high-score screen, and adds automatically overwrites the 11th best score when they have been beaten, increasing player's motivation.

The Ensoniq Mirage sound sampler was the ultimate example of User Involvement. The poor user — presumably a musician with little computer experience — must allocate memory for sound sample storage by entering two digit hexadecimal numbers using only increment and decrement buttons. Each 64K memory bank could hold up to eight samples, provided each sample was stored in a single contiguous memory block. In spite of the arcane and frustrating user interface (or perhaps because of the high functionality the interface supported with limited and cheap hardware) the Mirage was used very widely in the mid-1980s popular music, and maintains a loyal if eccentric following ten years later.

See also

VARIABLE-SIZED USER MEMORY offers an alternative to this pattern that explicitly models a reservoir of memory in the system, and allows users to store varying numbers of variable sized objects.

MEMORY FEEDBACK can be used to show users which memories are empty and which are full, or provide statistics on the overall use of memory (such as the percentage of memories used or free).

Although systems' requirements do not generally specify **FIXED SIZED USER MEMORIES**, by negotiating with your clients you may be able to arrange a **FEATURECTOMY**.

A **MEMORY BUDGET** can help you to design the number and sizes of **FIXED SIZED USER MEMORIES** **your system will support.**

You can use **FIXED ALLOCATION** to implement fixed sized user memories.

Variable-Sized User Memory

How can you present a medium amount of memory to the user?

- You have a medium to large amount of user-visible memory
- Users need to store a varying number of items in the memory
- The items users can store vary in size
- The memory requirements for what the user will need to store are unpredictable.

Some programs have medium or large amounts of memory available for storing user data. For example, the Strap-It-On wrist-portable PC provides a file system to allow users to store application data. Files can vary in size from a few words to several pages, and within the bounds of the systems memory, some users store a few large files while other users store many small files. The behaviour of some users changes over time — one week storing many small files, the next one large file, the next a mixture.

One approach to organising the memory would be to provide **FIXED SIZED USER MEMORY**. The system could allocate a fixed number of fixed-sized spaces to hold memos the user wishes to store. Of course, this suffers from all the problems of fixed allocation: memory spaces holding small memos will waste the rest of the space, and long memos must somehow be split over a number of different spaces. Another alternative would be to require users to pre-allocate space to store memos, but this requires users to be able to accurately estimate the size of a new memo before it is created, and the pre-allocation step will greatly complicate the user interface.

Therefore: *Randomly allocate users' objects from a reservoir of free memory.*

Allow the user to store and retrieve items flexibly from the systems' memory reservoir. The reservoir does not have to be made explicit in the interface design — although it may be. Each item stored in the memory should be treated as an individual object in the user interface, so that users can manipulate it directly. You also need to provide an interface to allow the user to find particular items they want to use, and to explicitly delete objects from the reservoir making the memory space available for the storage of new objects.

For example, the Strap-It-On uses **VARIABLE SIZED USER MEMORY** for its file system. A reservoir large enough to hold ten thousand words (about a hundred thousand characters of storage) is allocated to hold all the users' files. When users create new files they are stored within the reservoir until they are explicitly deleted. The file tool displays the memory used by every file in a browser view, the percentage of free memory left in the reservoir pool in its status line, and also uses error messages to warn the user when memory use exceeds certain thresholds (90%, 95% 99%).

Consequences

Users can store new objects in memory quite *easily,* provided there is enough space for them. Users can make *flexible* use of the main memory space, storing varying numbers and sizes of items to make the most of the systems capacity — effectively reducing the system's *memory requirements*. Users don't need to choose particular locations in which to store items or to worry about accidentally overwriting items or to do pre-allocation, and this increases the system's *usability*.

Variable sized memory allocation is generally quite *scalable*, as it is easier to increase the size of a reservoir (or add multiple separate reservoirs) than to increase the number of fixed size memories.

However: The program's *memory model is exposed.* The user needs to be aware of the reservoir, even though the reservoir may not be explicitly presented in the interface. The user interface will be *more complex* as a result, and the graphical design will be more difficult. Users will not always be aware of how much memory is left in the system, so they are more likely to *run out of memory*.

Any kind of variable allocation decreases the *predictability* of the program's memory use, and increases the possibility of *memory fragmentation*. Variable sized allocation also has a higher *testing cost* than fixed sized allocation.

Implementation

Although a variable sized user memory can give users an illusion of infinite memory, memory management issues must still lurk under this façade: somewhere the system needs to record that the objects are all occupying space from the same memory reservoir, and that the reservoir is finite. Even if the reservoir is not explicit in the interface design, try to produce a conceptual model for the use of memory which is integrated with the rest of the system and the domain model, so that the user can understand how the memory management works. Consider using **MEMORY FEEDBACK** to keep the user informed about the amount of memory used (and more importantly, available) in the reservoir — typically by listing the amount of memory occupied by objects when displaying the objects themselves.

A conceptual model based on variable sized user memory is more sophisticated than a similar model built on fixed sized user memory. A model of variable sized user memory must include not only empty or full memory locations, but also a more abstract concept of "memory space" that can be allocated between newly created objects and existing objects if their size increases. The objects that can be stored in user memory can also be more sophisticated, with varying sizes and types.

Multiple Reservoirs

You can implement multiple reservoirs to model multiple hardware resources, such as disks, flash ram cards, and so on. Each separate physical store should be treated as an individual reservoir. You need to present information about each reservoir individually, as the amount and percentages of free and used memory. You can also provide operations that work on whole reservoirs, such as backing up all the objects in once reservoir into another or deleting all the objects stored in a reservoir.

You will also need to ensure that the user interface associates each object with the reservoir where it is stored — typically by using reservoirs to structure the way information about the objects is presented, by grouping all the objects in a reservoir together. For example, most desktop GUI filing systems show all the files in a single disk or directory together in one window, so that the association between objects and the physical device on which they are stored is always clear.

Fragmentation

As with any kind of **VARIABLE SIZED DATA STRUCTURE**, the reservoir of user object memory may be subject to fragmentation. This can cause problems as the amount of memory that is reported as being available may be less than the amount of memory that can be used in practice. So for example, while the Strap-It-On's file memory may have 50K free characters, the largest single block might be only 10K – not enough to create a 15K email message.

One way to avoid this problem is to show users information about the largest free block of memory in each reservoir, rather than simply the amount of free memory. Another approach is to implement **MEMORY COMPACTION** — say with a user initiated compaction operation, in the same way that PC operating systems include explicit defragmentation operations.

Unfortunately, both these approaches complicate the users' conceptual model of the interface to include fragmentation. An alternative approach is to choose a data structure that does not require explicit compaction, either by compacting the structure automatically on every operation, or using a linked structure that splits objects across separately allocated memory blocks and so does not need compaction to use all the available space.

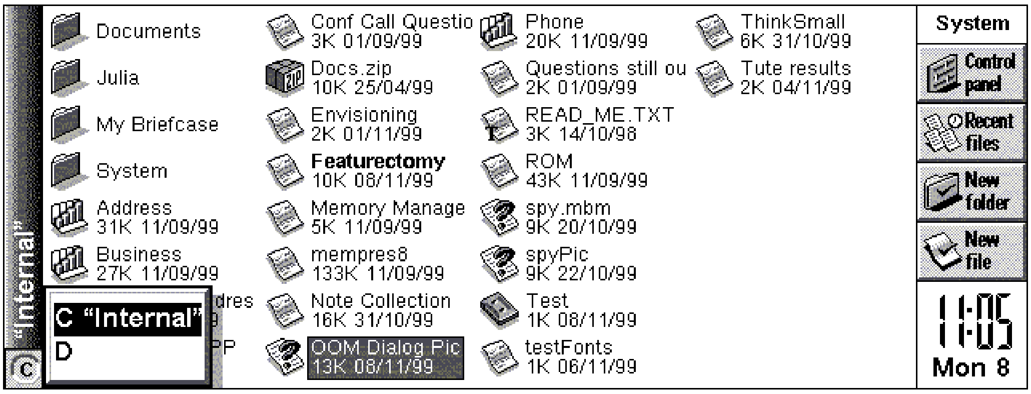
Caching and Temporary Storage

Caching can play havoc with the user's model of the memory space if applications trespass on that memory for caches or temporary storage. For example, many web browsers (including the Strap-It-On's Weblivion) cache pages in user storage, and browsers on palmtop and desktop machines similarly maintain temporary caches in user file storage. The minor problem here is that naïve measurements of the amount of free space will be too low, as some of the space is allocated to caches; the major problems is that unless the memory is somehow released from the caches it cannot be used for application data storage.

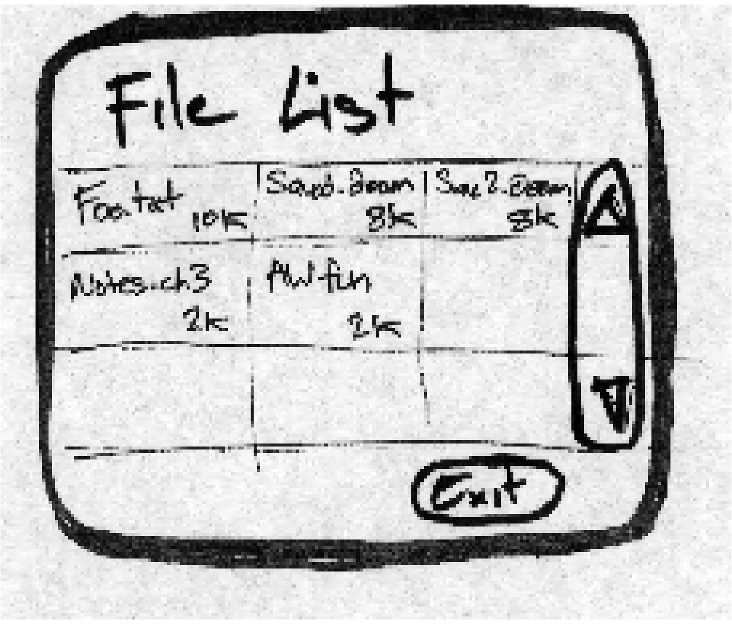
The **CAPTAIN OATES** pattern describes how you can design a system so that applications release their temporary storage when it is required for more important or permanent uses. The key to this pattern is that when memory is low, the system should signal applications that may be holding cached data. In response to receiving the signal, any applications holding cached data should release the caches.

Examples

A Psion Series 5 allows users to store text files and spreadsheet files in persistent storage reservoir (call a "drive" but implemented by battery backed up RAM). The browser interface illustrated in the figure below shows the files stored in a given drive, and the size of each file. Clicking on the "drive letter" in the bottom left hand corner of the screen produces a menu allowing users to view a different reservoir.



The StrapItOn PC can also list all the files in its internal memory, and also always lists their sizes.



[KJX hotmail account smallersoftware@hotmail.com]

Known Uses

Most general purpose computers provide some kind of variable sized user memory for storing users’ data — either in a special region of (persistent) primary storage, such as the PalmPilot, Newton, and Psion Series 5, or on secondary storage, if the form factor permits. Similarly, multitasking computers effectively use variable sized user memory — users can run applications of varying sizes until they run out of memory.

[More examples to do: Akai series of audio samplers; sequences; MP3 players?]

See also

You probably need to use **VARIABLE ALLOCATION** to implement variable sized user memory. **MEMORY FEEDBACK** can help the use avoid running out of memory. The size of the reservoir may be able to be set by **USER MEMORY CONFIGURATION.** **FIXED SIZED USER MEMORY** can be an alternative to this pattern if only a few, fixed sized objects need to be stored.

Memory feedback

How can users make good decisions about the use of memory?

- You have a **VARIABLE SIZED USER MEMORY**
- There is a medium to large amount of memory available
- Memory is proving a constraint on program functionality
- Users are managing some form or memory allocation
- Users memory allocation choices affect the performance or functionality of the system.

Some systems have reasonably large amounts of memory available, but memory allocation is difficult — typically because the memory allocation needs to match users' priorities or tasks and these are unpredictable, changing over time and between different users. For example, the StrapItOn has a fair amount of main memory, but this is quickly used up if users open too many applications simultaneously.

One way to deal with these problems is for users to accept some of the burden of managing the systems memory, perhaps by using **VARIABLE SIZED USER MEMORIES** — indeed, this is the solution adopted by the Strap-It-On's designers. Unfortunately, this solution raises another problem: how can the users get enough information to manage the memory effectively? Presumably memory allocation matters, and is too difficult to leave to the system — this is why users have been given the responsibility. But users need to make good memory allocation choices, or else the system won't work.

Other solutions are to provide information in printed documentation or in the help system about how the users' choices affect the memory of the system. But, in an interactive system where memory use changes dynamically at every run, this isn't really enough, because theoretical knowledge doesn't help work out what is wrong with the system's memory allocation right now, or how any changes to the allocation will affect the functioning of the system[1].

Therefore*: Provide feedback to users about the current memory use, and the consequences of their actions.*

As part of your interface design, you should have produced a conceptual model of the way the system uses memory (see **USER INVOLVEMENT**). Design ways to present the information in this model to users, as and when they need it. Include widgets in the display that tell users how much memory they have left, and lets them know the consequences of their decisions on the use of memory. These controls should be able to be accessed at any time, and integrated with the rest of the interface.

For example, the Strap-It-On user interface includes a dialog showing the user all the applications that are running, and how much memory is allocated to each application. This dialog is easy to reach from any application running on the StrapItOn. Furthermore, when the system runs low on memory, a variant of this dialog is displayed that not only lists all the applications and their memory usage, but also invites the user to terminate one of them to free up some memory space.

[1] Unless you are the kind of person who can intuit memory problems from the feel of the systems command line. There are people like this.

Consequences

Users are in better contact with the memory use of their systems. Their mental models about the way the system uses memory may be more likely to reflect the way the system uses memory. The *usability* of the system as a whole is increased (although still less than a system where users don't have to worry about memory).

Because users know how much memory is available, they are less likely to *exhaust* the system's memory.

However: The user needs a more sophisticated mental model of the system, and its implementation, that is, the way the system uses memory. This isn't really anything to do with the user's work, as it's an artefact of the implementation. The information about memory can confuse or distract the user from their primary task. Programmer *effort* will be required to implement the actual feedback interface elements, and programmer *discipline* may be required so that other parts of the system provide the necessary information to the feedback interface.

Memory feedback has to be tested independently, increasing the *testing cost* of the program.

Implementation

There are four main ways to provide feedback

- Memory report — an explicit report on the systems memory utilisation
- Memory meter— a continuous (unintrusive) meter of the systems memory use
- Memory notifier — a proactive warning when memory exhaustion is near
- Massive feedback — the system partially fails under memory exhaustion

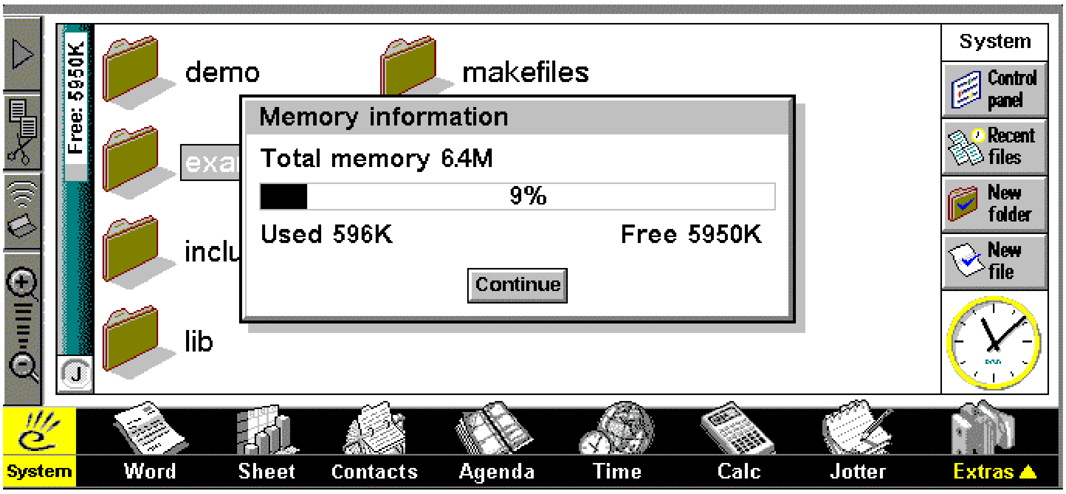
Memory Report

A memory report simply reports the memory use of the system, describing the various components consuming memory, and the amount of memory consumed by each component. Of course, the information presented in the report should be in terms of the system's conceptual model for memory use. Similarly, the units used to describe the memory should be comprehensible by most users, such as percentages or some other measure with typical values in double figures. A memory report can often be combined into the interface used to allocate, delete and browse **FIXED SIZE USER MEMORY OR VARIABLE SIZED USER MEMORY** because the information needed for all these tasks are so similar.

Because there are typically quite a few components in the system consuming memory, a memory report will need to provide a fair amount of detail to the user and take up quite a bit of screen space. So memory reports usually have to be designed as a part of the interface for users to request explicitly. One of the main purposes for the memory report is for the user to find out where the memory is being used in the system, so the report should be sorted in order of the memory consumption of each component, with the largest component first.

In an interactive system is it better if the display can be updated to reflect the instantaneous state of the memory in the system, although this will take code and effort. Alternatively, a memory report may be a passive display that is calculated each time it is requested.

For example, users of the Psion Series 5 can request a dialog box that provides a memory report, giving the total amount of user memory available, the amounts used and free, and the percentage of memory used.



Memory information

Total memory 6.4M

9%

Used 596K Free 5950K

Continue

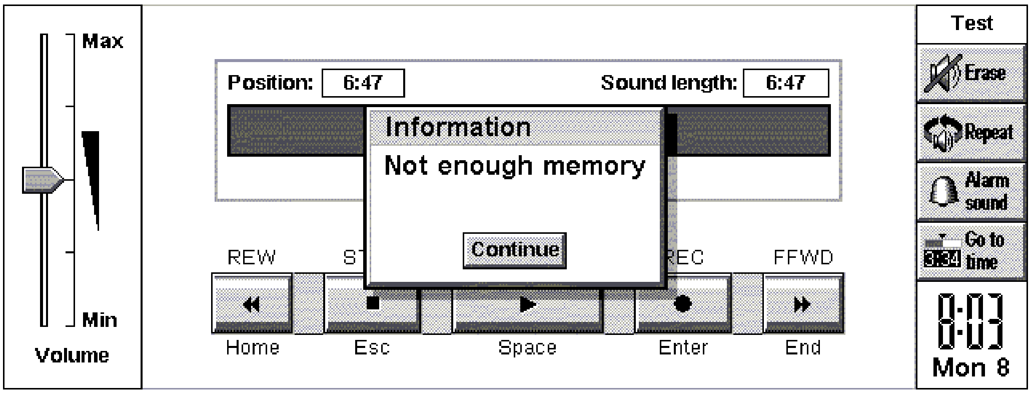
Memory Meter

A memory meter is a continuous display of the amount of free memory in the system. Because a memory meter is displayed continuously, it needs to be small an unintrusive part of the interface, so it cannot present the same amount of information as a memory report. To provide more information, make a memory report easily accessible via the memory meter.

The Psion Series 5 can also constantly display a memory meter listing the amount of free user memory (shown towards the left side of the screen in the above illustration).

Memory Notifier

A memory notifier is a proactive interface element (such as a popup warning window or audio signal) which is used to warn the user that the system's memory use is reaching a critical stage — such as no memory left to allocate! More usefully, memory notifiers can be generated as the systems memory gets low, but before it is totally exhausted, to give the user time to recover, or even as a warning in advance of any operation that is likely to allocate a large amount of memory. Of course, since notifier messages are modal, they will interrupt the work users are trying to do with the system and have to be explicitly dismissed. Because of this, they should be saved for situations where user intervention of some kind really is required, and memory meters used to give warnings in less critical situations. For example the following Figure is displayed by the Psion Series 5 voice recorded application, to indicate that it has run out of memory and consequently has had to stop recording.



Position: 6:47 Sound length: 6:47

Information

Not enough memory

Continue

As with all notifier messages and dialog boxes, a memory notifier should provide information about thereason for the notification and the actions the user can take to resolve the problem — either through an informative message in the notifier, or through a help message associated closely with it. Typically, a memory notifier should suggest something users can do to reduce the system's demands for memory.

The major technical problem with implementing notifier messages (and then contingent help) is that memory notifiers by definition appear when the system is low on memory, so there is often little memory available that can be used to display the notifier. Windows CE, for example, provides a special system call to display an out of memory dialog box in the system shell. The shell preallocates the resources required by this dialog box so that it can always be displayed.

```
if (!(addr = (VirtualAlloc(stuff))) {
   SHShowOutOfMemory(hwndowner, 0);
   return E_OUT_OF_MEMORY;
}
```

Passive Feedback

If temporary memory exhaustion causes computations to suffer Partial Failure or multimedia quality to drop, the results of the computation or multimedia can simply be omitted — output windows remain blank, the frame rate or resolution of generated images is reduced, and so on. Users notice this drop in quality, and free memory by shutting down other tasks so that the more important tasks can proceed.

Passive notification makes memory exhaustion produce the same symptoms as many other forms of temporary resource exhaustion (such as CPU overload, network congestion, or thrashing). This has the advantage that the same remedy (reducing the load on the system) can relieve all of these problems, but the disadvantage that the precise cause of the overload is not immediately clear to users.

Passive Feedback should never be used to report exhaustion of long-term memory — use a memory notifier for this case. Precisely because passive feedback is passive, it may not be noticed by users, which, in the case of long term memory, comes to silently throwing away users' data.

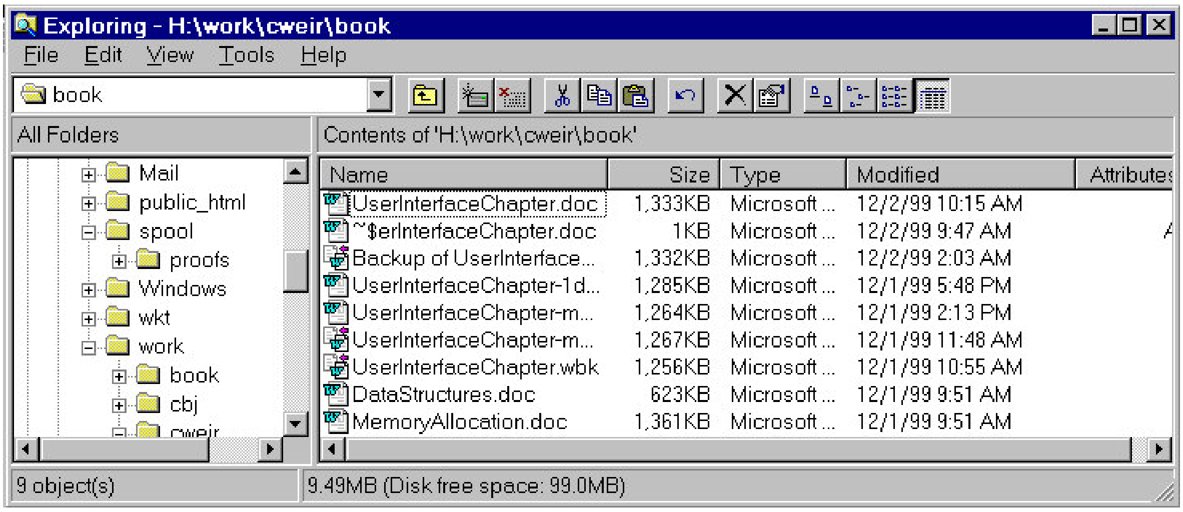
Ward Cunningham's Checks pattern language for information integrity describes a general technique for implementing Passive Feedback. [Cunningham PLOPD2]

Standard interfaces or Special-purpose interfaces

There are generally two alternatives for incorporating memory information into a user interface — either the application's general purpose displays can be extended to include memory information, or special purpose displays can be designed that focus on memory information.

Extending an application's standard, general purpose displays has several advantages: the memory feedback can be tightly integrated with the domain information present by the application; because of this integration, the feedback can be easily incorporated into each user's model of the system; and users don't need to learn a new part of the interface to manage memory.

A good example of this kind of integration can be found in many file system browers (including those of Windows, the Macintosh, and the PalmPilot), that seamless include file size information along with other information about the files. Indeed, size information has listed along with file names for so long that most habitual users of computer systems expect it to be listed, and do not realise that the size of each file is really an implementation concern to help them manage the system's memory. The illustration shows how Windows Explorer lists file sizes as the second most important piece of information about a file, other than the file name. On the other hand, including memory feedback in an integrated view does mean users are present with it whether or not they really need to know about it, and this also takes up screen real estate that could be used for to display more useful information, such as files' full names and types.



Name	Size	Type	Modified	Attributes
UserInterfaceChapter.doc	1,333KB	Microsoft ...	12/2/99 10:15 AM	
~$erInterfaceChapter.doc	1KB	Microsoft ...	12/2/99 9:47 AM	A
Backup of UserInterface...	1,332KB	Microsoft ...	12/2/99 2:03 AM	
UserInterfaceChapter-1d...	1,285KB	Microsoft ...	12/1/99 5:48 PM	
UserInterfaceChapter-m...	1,264KB	Microsoft ...	12/1/99 2:13 PM	
UserInterfaceChapter-m...	1,267KB	Microsoft ...	12/1/99 11:48 AM	
UserInterfaceChapter.wbk	1,256KB	Microsoft ...	12/1/99 10:55 AM	
DataStructures.doc	623KB	Microsoft ...	12/1/99 9:51 AM	
MemoryAllocation.doc	1,361KB	Microsoft ...	12/1/99 9:51 AM	

9 object(s)

9.49MB (Disk free space: 99.0MB)

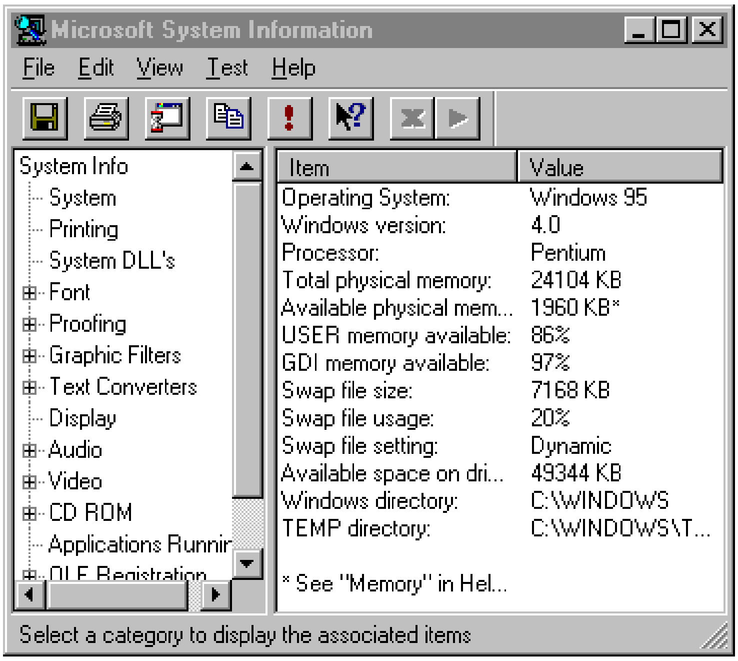
.

Alternatively, you can design special purpose displays to present your memory feedback, rather than integrating then directly into your application's main interface. This avoids cluttering up the display to present users with memory feedback that they do not need, but of course makes it more difficult for users to use, and to learn about, the memory feedback that you do provide. Separate interfaces also make it easier to provide special purpose operations (such as formatting or backing up a storage device) that are not really part of the operations in the system's domain.

As with any user interface, it is important that an interface for displaying memory information is as simple and unintrusive as possible. Obviously, it is not possible for an urgent low memory warning notifier to be unintrusive, but most memory reports and memory meters do not need to be designed to draw users' attention away from the actual work they are doing with the system.

Finally, if you do choose to design one or more separate interface contexts for managing memory, it is important that the information displayed in those interfaces is mutually consistent, and also consistent with the information displayed by the application's main user interface. Microsoft Windows unfortunately provides several counter examples to this principle — with a number of tools that provide information about memory use (from special monitor tools to application about boxes) but where every tool displays quite different, often contradictory information about the system's memory use.

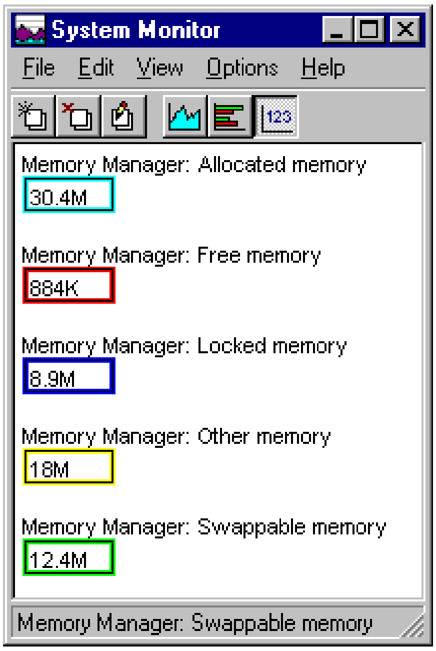
For example, the illustration below shows two of Windows information tools displaying completely unrelated information about memory use. Furthermore, displaying an application "About" box also provides information about memory use — in this case, saying that "Available Physical Memory " is 24104 KB".



Item	Value
Operating System:	Windows 95
Windows version:	4.0
Processor:	Pentium
Total physical memory:	24104 KB
Available physical mem...	1960 KB*
USER memory available:	86%
GDI memory available:	97%
Swap file size:	7168 KB
Swap file usage:	20%
Swap file setting:	Dynamic
Available space on dri...	49344 KB
Windows directory:	C:\WINDOWS
TEMP directory:	C:\WINDOWS\T...

* See "Memory" in Hel...

Select a category to display the associated items



Memory Manager: Allocated memory
30.4M

Memory Manager: Free memory
884K

Memory Manager: Locked memory
8.9M

Memory Manager: Other memory
18M

Memory Manager: Swappable memory
12.4M

Memory Manager: Swappable memory

Display Styles

Memory reports do not have to be textual. Graphical displays can be quite effective, especially for presenting an overview of the state of memory use, such as the pie charts used by Windows to display information about disk space, although this particular example is quite intrusive — the pie does not need to be as large as it is! Other graphical alternatives include bar charts, line charts, and even old fashioned meter dials, used by various tools in Unix and Windows NT.

[kjx pictures of resource meters]

Graphs of resource usage against time can also provide much useful information quite plainly, by showing not only the current state of the system, but also the trends as the system runs. For this reason, many resource meters, including Unix's xload and Windows System Monitor use simple time series charts, either as alternatives or in addition to displays showing the current state.

Displaying information about different kinds of memory

Often a system can have several different constraints that apply to different kinds of memory. In this situation, allocating or freeing space in one kind of memory does not affect the memory consumption in another kind of memory space. For example, any of the following memory spaces may be constrained:

- Global heap space shared by the whole system
- Heap space for each individual process
- Stack space for each individual process
- Reservoirs used to store objects in **VARIABLE SIZED USER MEMORY**.
- Physical secondary storage — in disk, flash cards, or battery backed-up RAM
- Memory buffers or caches used by file or network subsystems, paging, or for loading packages
- Garbage collected heaps (and different regions within the heaps for sophisticated algorithms)

If a system has several different kinds of memory with different characteristics, each of these needs to be treated individually. This will complicate the users' conceptual model of the system, because to understand and operate the system users will have to understand what each different kind of memory is for, how it is used, and how their use of the system affects its use of each different kind memory. Memory reports or memory meters can display information about the different kinds of memory in the system, so that users can manage each kind as necessary.

The TeX batch-mode document formatting system uses pooled allocation, so when it is configured different amounts of memory must be set apart to represent strings, characters within strings, words, control sequences, fonts and font information, hyphenation exceptions, and five different kinds of stack. Every time TeX completes running it produces a memory report in a log file that itemises its use of each different kind of memory. Typically, users only read these log files when TeX runs out of memory, and the information in the log can help determine precisely why TeX ran out of memory. For example, if font memory is exhausted then the document has used too many fonts — this can only be fixed by configuring a version of TeX with more font space, or by using **DATA CHAINING**, and subdividing the document so each part uses fewer fonts. Alternatively, a stack overflow generally means the user has accidentally written a recursive macro call, and needs to find and fix the macro definition.

```
Here is how much of TeX's memory you used:
 523 strings out of 10976
 5451 string characters out of 73130
 56812 words of memory out of 263001
 3434 multiletter control sequences out of 10000+0
 16468 words of font info for 45 fonts, out of 200000 for 1000
 14 hyphenation exceptions out of 1000
 23i,10n,21p,311b,305s stack positions out of 300i,100n,500p,30000b,4000s
```

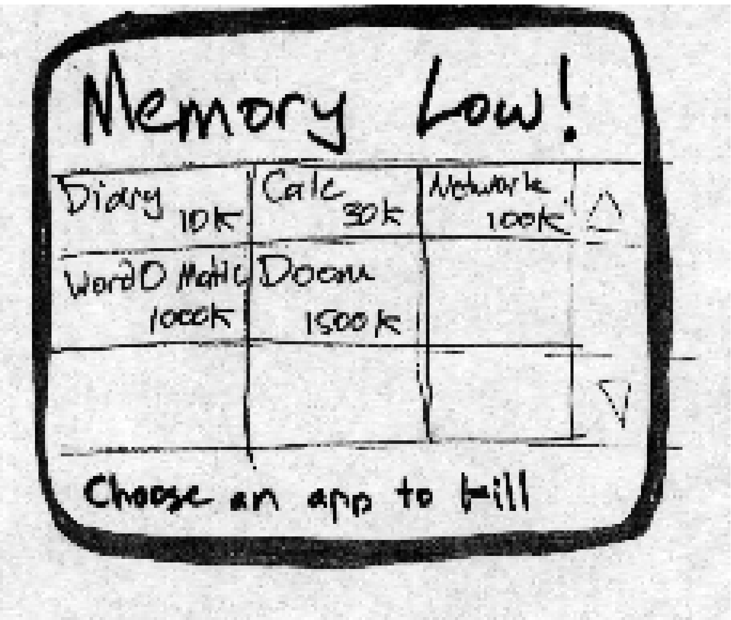
The Self programming language (Ungar and Smith, 1999) includes a sophisticated Spy tool that visualises Self's use of many different sorts of memory.

[kjx pictures of self spy]

Examples

The users' conceptual model for the Strap-It-On PC includes explicit objects that represent running instances of applications, and includes the amount of memory used by the application as one of the attributes of that object. Every time users display applications, the amount of memory they occupy is displayed with them, to ensure users understand the relationships between applications and memory consumption.

The figure below illustrates how this information is included into the Strap-It-On's "Please Kill an Application" dialog. This dialog is presented to the user whenever the system is running low on memory, and lists all the running application in order of their memory use (with the most profligate listed first). Using the wrist-mounted touch screen, uses can choose which application they wish to kill, thereby releasing memory for tasks they presumably consider more important.



Known Uses

The Macintosh and MS-Windows systems display the amount of free disk space in every desktop disk window. As secondary storage has become less of a constraint, the prominence of the disk space meter has declined (from the title bar on the top of a Macintosh window to a footnote in a status line in Windows95).

The virtual machine for the Self programming language incorporated a sophisticated memory meter. The Spy window graphically displayed the memory usage of a number of different heap regions, plus details of paging and garbage collector performance (for example, memory that is swapped out is displayed greyed out).

The Macintosh "About Box" shows the memory usage of each application and the operating system. MS-Windows applications typically also provides some memory use information in their about boxes: their top of the line applications have an over-the-top system information utility in their about boxes – no doubt to assist telephone support engineers working with the likes of Dilbert™'s boss! Windows also provides a series of memory monitors and system tools that are mostly useless (because the statistics they display don't make sense to most users, and don't seen to agree with each other) – but some applications do have their own 'about boxes' that display useful memory information.

See also

USER MEMORY CONFIGURATION describes how memory feedback can be combined with an interface used for adjusting the memory configuration.

If you have a **VARIABLE SIZED USER MEMORY** the feedback can be combined with the interface used for accessing the user memory.

If you are doing **LOW QUALITY MULTIMEDIA** at runtime you can provide feedback about the trade-offs users make between quality and memory.

User Memory Configuration

Sometimes the system needs to adjust its memory requirements to suit user tasks

- You have a medium to large amount of system memory
- The system needs to allocate that memory for a number of different purposes related to the internal functions of the system.
- The allocation depends upon the tasks the user will perform.
- The allocation affects the relative performance or quality of the systems support for some user tasks.

Some programs have medium or large amounts of memory available, but this memory needs to be divided up to service a number of competing demands. For example, the Strap-It-On needs to allocate memory to store temporary copies of users documents as they are being edited, font caches to speed up rendering of those documents when they are displayed, image caches to store images rendered at screen resolution, and so on. Making the memory allocation between these competing demands will alter the performance, quality of user experience, or even the functionality of the system. If for example, you allocate no memory to the font cache perhaps the system won't display proportional fonts; if you allocate no memory to the sounds buffers, it won't play any sounds, and so on.

A simple approach to handling this is to just allocate a fixed amount of memory to each system component — either an absolute amount of memory, or perhaps some proportion of the system's overall available memory. While this is at least a memory allocation it doesn't really solve the problem: memory will be allocated to services in which users are uninterested, and the services the user considers more important will be starved of memory. Even if the system tries to dynamically juggle memory between various different uses, it still effectively has to guess what users' preferences are going to be.

So these approaches really don't solve the problem. The allocation of memory needs to depend upon the current user's current tasks: if editing a document, perhaps fast font rendering is most important; if playing a video game, perhaps stereo sound is important. The system cannot make these choices in advance because it cannot know the priorities of the particular user.

Therefore: *Let users choose the system's priorities for memory.*

Design a conceptual model for the system's memory use. Ideally the model should be related to (or created from) the conceptual model of the interface and the user domain model. For if users have to manage memory themselves, you'll want the interface for it to be as closely integrated with the rest of the system as possible. A Memory Budget is a good basis for such a model.

Design an interface through which users can indicate how they would like memory to be allocated based on the conceptual model underlying the whole interface. Using this interface, users can then juggle memory themselves, allocating memory to those parts of the system they consider important.

User memory configuration can also manage trade-offs between memory demands and other requirements (typically time performance), as well as between competing demands for memory.

For example, the Strap-It-On interface includes a "system properties" tool that displays details of the attached hardware, software, and system services. This tool also allows users to adjust the amount of memory allocated to system services (including the font renderer, disk cache, sound player, and so on), and even to turn unwanted services off, completely removing them from memory.

Consequences

The system can allocate memory just where users wants it, making more *efficient* use of the available memory, because the system doesn't have to guess the users preferred allocation of memory space. This effectively reduces the program's *memory requirements* to support any given task.

Users can tailor memory configurations to suit hardware that was not envisaged by a program's original designers. This increases the *scalability* of the system, because if more memory is available, it can be put to use where it will provide the most benefit.

However: The user's conceptual model of the system now has to incorporate quite a sophisticated model of systems use of memory. Interactions with the system are more *complicated* because users have to adjust the memory configuration from time to time. If users don't make a good memory allocation then the system will be worse off than if it had tried to do the allocation itself. These problems can easily reduce the system's *usability*.

The implementation must be reliable enough to cope when the memory allocation is changed, costing *programmer effort* to implement the flexibility, and *programmer discipline* and *testing costs* to make sure every component is well behaved. By definition, allowing the user control over a system's memory use must decrease the *predictability* of the system's memory use.

Supporting centralised user memory configuration encourages *global* rather than local control of memory use.

Implementation

A system's user memory configuration needs to be tightly coupled to its **MEMORY FEEDBACK**. Before users can decide how they would like the system to allocate its memory, they need information about the system's current use of memory and the current configuration. Therefore the interface for memory configuration must be as close to the interface for memory feedback as possible — ideally, both patterns should be implemented in the same part of the interface; at a minimum, both interfaces should share similar terms and an underlying conceptual model. **MEMORY FEEDBACK** can also give direct instructions on configuration. For example, MS Windows out of memory message includes the instructions to "close down some applications, then expand your page file".

Alternatively, if the user memory configuration may be a task normally performed by a specialised user role, such as a system administrator, rather than a more normal user. If this is the case, then it makes sense for memory configuration to be supported in the same way as any other configuration task required by the application, typically in a special part of the interface used by the administrator to accomplish all configuration tasks.

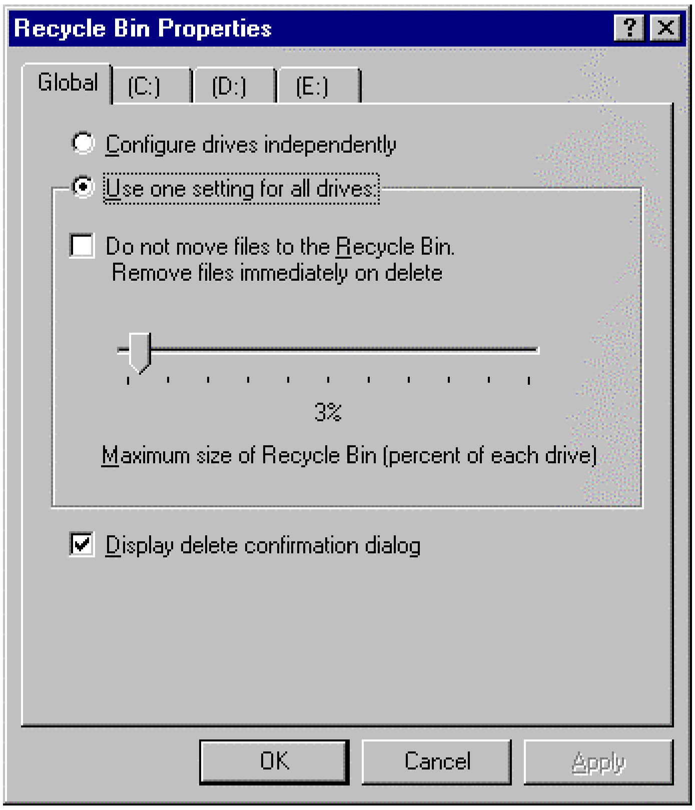
More generally, there are four main techniques you can use to design interfaces for user memory configuration:

- Editable graphical displays
- Dialog boxes with text fields
- Binary choices rather than continuous values.
- Textual configuration files.

Graphical Displays

Where the device hardware supports bitmapped displays and graphical user interfaces, and where configuration is part of the standard use of the system, then the interface for memory configuration should match that of the rest of the program. That is, users should interact with appropriate graphical widgets to supply the information. Generally, some kind of slider control should be chosen to represent the kind of continuous numeric data usually required by configuration parameters.

Microsoft windows and the Palm Pilot all make great use of sliders to configure their memory use. For example, the Windows dialog box below shows a slider use to allocate the secondary storage space for caching deleted files in the 'Recycle Bin'.



Recycle Bin Properties

○ Configure drives independently

● Use one setting for all drives:

□ Do not move files to the Recycle Bin.
Remove files immediately on delete

3%

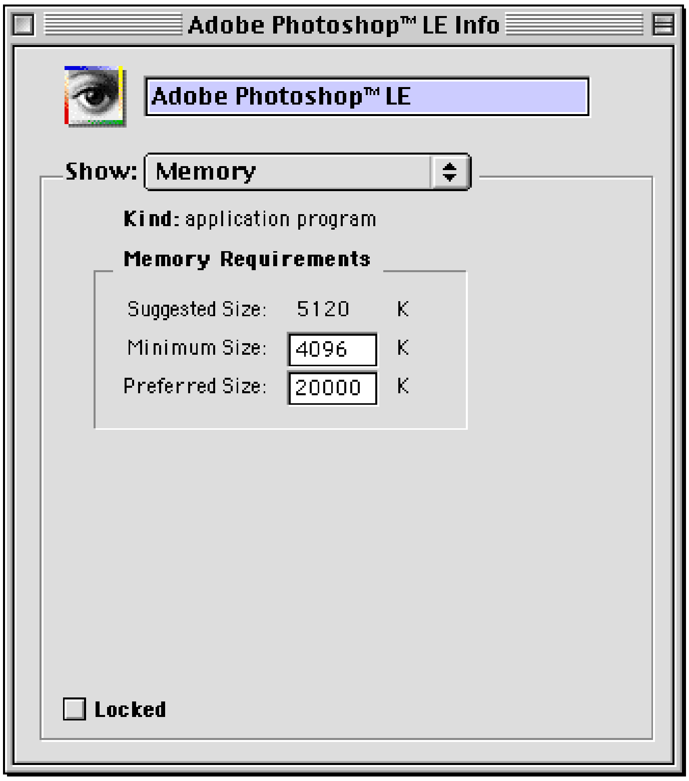
Maximum size of Recycle Bin (percent of each drive)

☑ Display delete confirmation dialog

Text Fields

Technically, graphical sliders and other widgets can have some problems: they can make it difficult for users to enter precise values for configuration parameters, and can be hard to implement correctly. Simple text files can be a common alternative to sliders, especially if a large range of values many need to be entered (from 100K to 500M is not unreasonable) or if precision is more important than graphical style.

For example, Apple Macintosh computers allow users to configure the amount of memory that should be allocated to each program when it is executing, via text entry fields in the application's information box.



Adobe Photoshop™ LE

Show: Memory

Kind: application program

Memory Requirements

Suggested Size:	5120	K
Minimum Size:	4096	K
Preferred Size:	20000	K

□ Locked

Binary Choices

Another alternative is to present binary choices about memory use, rather than continuous scalar choices. Binary questions are often easier for users to understand because they can be phrased in terms of a simple conceptual model of the application. Consider the difference between a scalar choice (to allocate between zero and ten megabytes to an antialiased font cache) and a binary choice (to turn font antialiasing on or off). For users to understand the continuous choice, they must not only understand what font antialiasing is, and why they might want it, but also what a byte or a kilobyte is, how many they have at their disposal, and how many they wish to spend on antialiasing. Once they have chosen or altered the configuration value, it may not be easy to see the difference the parameter makes (say between a one hundred kilobyte cache and a one hundred and three kilobyte cache).

Alternatively, a binary choice is simpler than a continuous choice: users need only have a rough idea about what font antialiasing is, and the effects of either value of the parameter will be quickly apparent.

For example, Windows NT allows you to turn various system services on or off, but not to have a service half-enabled. PKZIP offers a choice between optimise for space and optimise for speed.

[kjx example, installer for something? NT SERVER manager]

Textural Configuration Files

The simplest way to implement user memory configuration, especially at configuration time, is to read in a configuration file defining the values of the configuration parameters. If your environment has an operating system, you may be able to use environment variables or registry parameters as alternatives to reading in a file. These options have the advantages that the are trivial to implement, and for users who are software engineers or system administrators may be at least as easy to use as more extensive graphical facilities. On the other hand, they are difficult to use for the less technically savvy, and are typically only suitable for parameters to be set at configuration time.

[kjx example, config.sys or similar?]

Static versus Dynamic Configuration

As with many decisions about memory use, user memory configuration can be either static supported at configuration time before the system begins running, or dynamically adjustable while the system is running. Generally, static configuration is easiest to implement (allowing dynamic configuration of

Handling memory configuration at installation or configuration time also has the advantage that that is when the software's resource consumption will be foremost in users minds, and so they are most likely to be interested in making configuration decisions. The disadvantage is that without having used the software, they will not necessarily be in a good position to make those decisions.

Automatic and Default Configuration

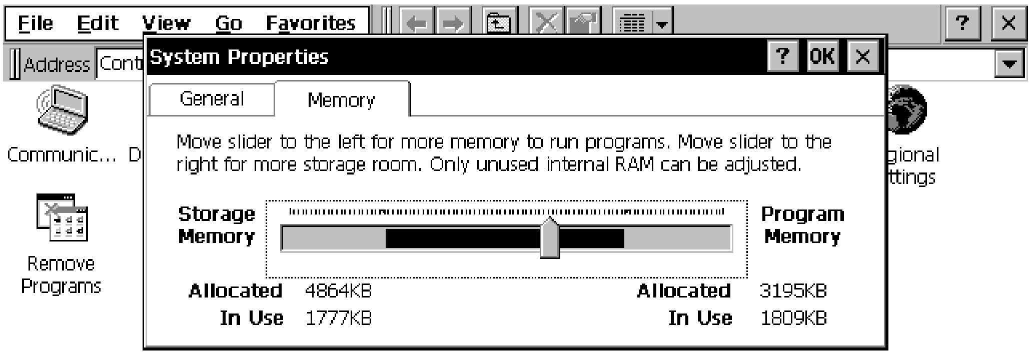
One way to mitigate the negative effects of this pattern is to combine it with some level of automatic allocation, or, at the very least provide some sensible default allocations. This way, unsophisticated users who don't need or want to care about memory allocation will at least receive some level of performance from the system, while those users who are willing to tailor the memory allocation strategy can receive better performance. Also, when users take control of memory allocation, it can be useful to provide some sanity checking, so that, for example, you cannot deallocate all the memory currently used by the running program.

For example most MS Windows installations dialogs allow the user to optimise the use of secondary storage by choose between several pre-set installation configurations, or by designing their own custom configuration. Most users simply use the "workstation" configuration that is selected by default

[kjx wizard?]

Examples

Windows CE uses a slider to configure its most important memory parameter — balancing the amount of memory allocated to storing users data ("Storage Memory") versus the amount of memory allocated to running applications ("Program Memory"). Users can adjust the mix by dragging the slider, within the bounds of the memory currently occupied by storage and programs.

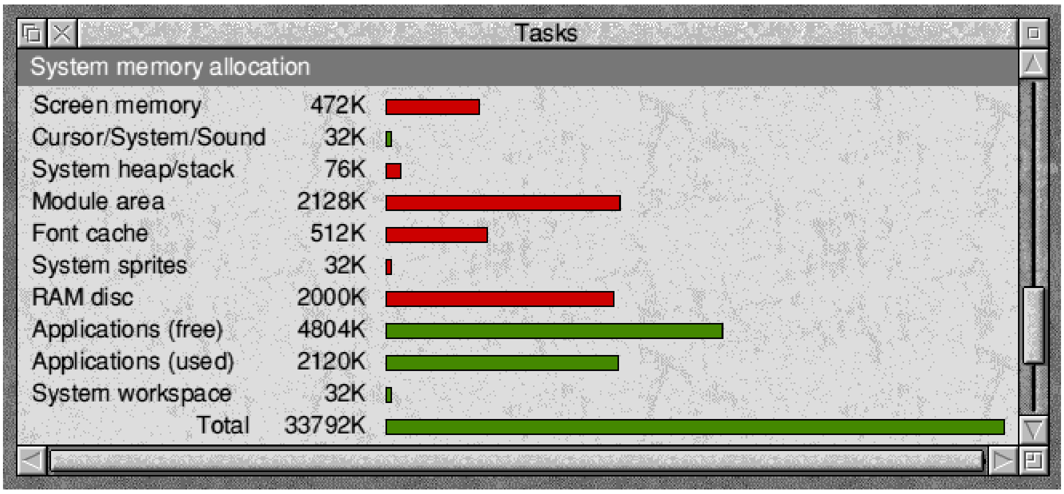


Move slider to the left for more memory to run programs. Move slider to the right for more storage room. Only unused internal RAM can be adjusted.

Storage Memory		Program Memory	
Allocated	4864KB	Allocated	3195KB
In Use	1777KB	In Use	1809KB

The Acorn Archimedes operating system has a paradigmatic example of combined memory feedback and memory configuration. The system can display a bar graph showing the system's current memory configuration and memory use for a whole range of parameters —

for example, 1000K might be configured for a font cache, but only 60K of the cache is actually occupied. The user can then drag the bars representing tuneable parameters to change the system's memory configuration. System services are automatically disabled if the amount of memory allocated to them is zero.



Tasks

System memory allocation

Screen memory	472K
Cursor/System/Sound	32K
System heap/stack	76K
Module area	2128K
Font cache	512K
System sprites	32K
RAM disc	2000K
Applications (free)	4804K
Applications (used)	2120K
System workspace	32K
Total	33792K

Known Uses

Web browsers, such as Netscape and Internet Explorer, allow users to configure the size of page caches in primary or secondary storage.

Venerable operating systems such as MS-DOS, VMS, and Unix have traditionally provided configuration files or command line parameters that can be used to tweak memory use. DOS's CONFIG.SYS is a classic example. Users (for some special meaning of the word) can increasing the number of buffers, open files supported at the cost of decreasing the memory available to applications. More cuddly operating systems, such as Windows NT, sometimes provide configuration dialogue boxes that offer similar features. For example, NT's service manager window allows users to turn operating system services on and off, and turning a service off releases that service's memory. The operating system BeOS takes this one step further and allows users to turn CPUs on and off — mostly off in practice, unfortunately, since this option instantly crashes the machine!

See also.

MEMORY FEEDBACK can provide users with the information they need to make sensible configuration choices when configuring a system's memory use.

Users can be allowed to reduce the amount of memory allocated to multimedia in a program if the program supports dynamically applying **LOW QUALITY MULTIMEDIA.**

Allowing users to alter memory allocation to other parts of the program can be implemented with help from the **PARTIAL FAILURE, CAPTAIN OATS** and **MEMORY LIMIT** patterns.

USER MEMORY CONFIGURATION is the run-time complement to a **MEMORY BUDGET**.

You probably need to use some form of **VARIABLE ALLOCATION** and **MEMORY LIMITS** to implement user memory configuration successfully.

Low Quality Multimedia

Also Know As: Space-time Trade-off; Never Mind the Quality, Feel the Width.

How can you reduce the memory requirements for multimedia?

- You need to support graphics, sound, or other multimedia, to enhance the user experience.
- The memory requirements for the multimedia are too big for your memory budget.
- The broad details of the multimedia are more important than the fine details.
- The multimedia is peripheral to the focus of the program.
- The program needs to provide real-time or near real-time response.

Some programs need to include multimedia presentation or interaction. For example, the original specifications for the StrapItOn required it to play a video of an imploding supernova being consumed by a black whole (complete with 3D graphics and stereo-surround-sound effects by a chart-topping grunge band) every time a memo was deleted from its database. Arguably this should be a candidate for a **FEATURECTOMY** but since competing products support a similar feature, management ruled it had to be included. This pleased Gerald the Geek, who had spent the last three months tweaking the photo-relativistic star implosion simulation after Our Founder's video was cancelled.

Unfortunately, graphics (particularly video animations), sound, three-dimensional models and other multimedia resources occupy large amounts of memory — secondary storage, perhaps, when they are not in use, but also large amounts of primary memory when then are being displayed, played, rendered, or otherwise presented to users. The memory requirements for multimedia can often make a large contribution to the memory requirements of the program as a whole.

Many multimedia resources are, however, only peripheral to the users' tasks the program is supposed to support — that is, they may help ensure user's enjoy using the program, but they don't actually help users get their work done. Microsoft Windows contains a number of examples of such peripheral uses of multimedia — animations in dialog boxes for file move or delete, the irritating help "wizards", and musical phrases played as the system starts up and shuts down. A typical installation of Windows 95 requires six megabytes for sound and music files alone, but will operate quite successfully with sound output muted and wizards deactivated.

So, how can you support complex multimedia presentations while remaining within your memory budget?

Therefore*: Reduce the quality — bits per pixel, sample length, size, complexity, or detail — of the user experience multimedia.*

The memory requirements of a multimedia resource depend upon the size and quality of the resource. If you increase the size or the quality, you will increase the memory requirements. More to the point, if you decrease the quality, you can decrease the memory requirements for a given resource size. So, to fit a given amount of multimedia into a fixed amount of memory — never mind the quality, feel the width — that is, decrease the quality of the multimedia to fit the memory available.

Often, multimedia resources are constructed when the program is being built, and so you can process them in advance — often simply by changing them to use a lower quality format. If you are using dynamic memory allocation, or cannot process all the multimedia resources in advance, consider using **Partial Failure** to let quality degrade gradually as resources are consumed, by determining the amount of memory available and reducing quality until they fit.

For example, by reducing the Strap-It-On imploding supernova animation to a 32 pixel square using only 256 colours and showing only 10 frames, the whole animation could be fitted into 10K. Gerald was still upset, until someone else pointed out it meant a) they could include lots more animations if the quality was kept low, and, b) they would need lots of work in advance to compress and tweak the animations.

Consequences

By reducing the quality of the multimedia used in a program, the *memory requirements* for the multimedia, and thus the program as a whole, are reduced. Using lower-quality multimedia can also increase the *time performance* and especially the *real-time response* of the program.

If several pieces of multimedia can be treated in the same way (such as requiring all images to be black and white, and a maximum 200 pixels square) can increase the *predictability* of the program's memory use. Removing truly unnecessary multimedia can increase the program's *usability.*

However: The *quality* of the presentation is reduced, and this may reduce the program's *usability*. Reducing multimedia quality can make it harder to scale the program up to take advantage of environments where the higher quality could be supported, reducing the program's *scalability*.**Programmer *effort* is required to process the multimedia to reduce its quality. Reducing quality dynamically will take even more *effort*, and increase *testing cost*.** Some techniques for reducing multimedia's quality of storage requirements may suffer from *legal* problems, such as software patents.

Implementation

By carefully tailoring your multimedia quality to match the output device, you can keep perceived quality high, while lowering memory requirements. The simplest case is that it is never worth storing data at a higher quality than the output device can reproduce. For example, if you only have a 75dpi screen, why store images at 300dpi? 75dpi images will look exactly the same, but take up much less space. Similarly, if you only have an 8-bit laptop speaker, why store 16-bit sound?

Many sound and image manipulation programs explicitly allow multimedia to be stored in different formats, so that you can make the trade-off between quantity and quality — these formats often use **Compression** to save memory. Specialist tools are also available for automatically adjusting the quality of image files to be served on the web, to reduce their memory size and thus download time.

Unfortunately, tailoring multimedia to fit a particular device is a one-way trip: if a better device becomes available then low-resolution multimedia will not be able to take advantage of it. Similarly, if you need to enlarge an image, for example, once the resolution has been reduced it cannot be increased.

Other Space Trade-offs

This pattern can also be used to trade off other qualities against memory use and multimedia quality. In particular, multimedia presentations can often require large amounts of CPU time or regular (soft-real time) amounts of it. Meeting these requirements can also reduce a program's absolute time performance or real-time response. Downloading large multimedia presentations requires a large amount of network bandwidth. Reducing the quality of the multimedia in your system can simultaneously increase a program's responsiveness while reducing its requirements for bandwidth, secondary storage, and main memory.

Quality vs. cost trade-offs can be made statically while the program is being written, when the program is installed or configured, or dynamically as the program runs. For example, an animated user interface may ideally need to produce 16-20 frames per second. If resources are not available, it could dynamically either produce fewer frames and display each frame for longer, reducing the frame rate, or it could maintain the frame rate but reduce the amount of detail in each frame. Making trade-offs dynamically means that the multimedia quality can be adjusted precisely to suit the system that is displaying it, but has the disadvantage that the full quality presentation needs to be stored somewhere, typically on secondary storage, or may need to be downloaded over a slow network link. On the other hand, reducing quality at design time means that only the lower quality multimedia needs to be stored or transmitted, and any computation necessary to reduce quality (such as **ADAPTIVE FILE COMPRESSION** can be carried out before hand, without tight constraints on CPU time or memory use.

User memory configuration

With some care, you can design your software so that users can choose what multimedia features will be included as the program runs, and which will be left as options. For example, a web browser could let users choose:

- Whether or not to download images automatically
- If not, when they would like images to be downloaded
- Whether or not to play sounds found on any web pages
- Whether or not to download or show any animated images found on web pages.

As discussed in the **USER MEMORY CONFIGURATION** pattern, these choices could be binary or continuous. While binary choices are easier to understand than a continuous choice, continuous choices could also be used — for example, an option to download only images below a certainly size would enable small images to be downloaded quickly without user intervention, while requiting an explicit user request to download large images.

The Virtue of Prudence

Once you have chosen to reduce the quality of the multimedia, or even the presentation quality of your user interface, you do not have to settle for an interface that feels low-quality overall. In fact, with clever interface design it is possible to make a low-quality interface a virtue, rather than a liability. The early Macintosh user interface design is a classic example of this. Early Macintoshes had physically smaller screens than competing IBM PCs, did not support colour graphics, did not have nearly as many keys on the keyboard or expansion slots inside the chassis. With clever design (and marketing!) all of these were turned in the Macintoshes favour, so that various PC software manufacturers eventually copied its interface to remain competitive. More recently, the PalmPilot is similarly making a virtue of graphics display that has at least ten times less overall resolution than a PC screen. The Playstation is another example — although it cannot match the resolution of a high-quality PC monitor because it must display its output on domestic TV receivers, its games are carefully designed to suit the resulting "grungy" low-resolution feel (and to take advantage of 3D hardware most PCs do not support). In the music industry, recent synthesisers include features to reduce the quality of a sound sample to provide a lower-fidelity distorted sound highly sought after by "alternative" musicians.

Examples

Different kinds of multimedia formats can require vastly different amounts of memory to store, but when it comes to getting an idea across (rather than impressing users with production values) the benefits of higher resolution formats may not be worth the effort.

For example, the ASCII text to describe a small rodent occupies five bytes

```
mouse
```

and a logically expanded version occupies only three bytes

```
rat
```

A simple graphical representation occupies about forty-six bytes:

```
    () ()______
    /**        )    _
  O_\\-m--m-/_____)
```

and a bitmapped version about 108 bytes in GIF format.

A line drawing in postscript occupies rather more space:

[Addison-Wesley to provide an line-drawn icon for a mouse]

and a photorealistic picture, even more.

[Addison-Wesley to provide a library photograph of a mouse]

On a larger scale, Microsoft PowerPoint can save presentations in a number of formats, for different screen types, for printing onto overhead transparencies, or for displaying on the Worldwide web. The space occupied by a presentation can change by a factor of ten depending upon how it is stored.

[need actual examples here]

Known Uses

The Non-Designers Web Book [ndwb] describes how images to be presented on web page can be tailored to reduce their memory consumption (and thus download time) while maintaining good

The Windows95 sound recorder tool makes trade-offs between quality and memory requirements explicit, showing how much memory a given sound occupies, and allowing users to choose a different format that will change the sound file's memory use.

Many desktop computers, from the Apple][series to most modern PC clones, allow users to choose the resolution and colour depth used by the display screen. In many of these machines, including the including the BBC Micro and Acorn Archimedes, choosing a lower resolution or fewer colours (or black-and-white only) left more memory free for running programs.

Web browsers, including Netscape and Internet Explorer, allow users to tailor the display quality to match their network bandwidth, by choosing not to download images automatically or caching frequently accessed pages on local secondary storage.

See also

COMPRESSION can provide an alternative to lowering multimedia quality. Rather than reducing memory requirements by reducing quality, reduce the requirements by using a more efficient data representation. Unfortunately, while many types of multimedia resources can be compressed efficiently for storage, then need to be uncompressed before they can be used, and can cost more *processor time* and *temporary memory* than using the uncompressed resource directly.

A good **FEATURECTOMY** may let you remove unnecessary multimedia from the program completely, while **USER MEMORY CONFIGURATION** will let your users choose how much memory to spend on multimedia, as against getting useful work done.