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.

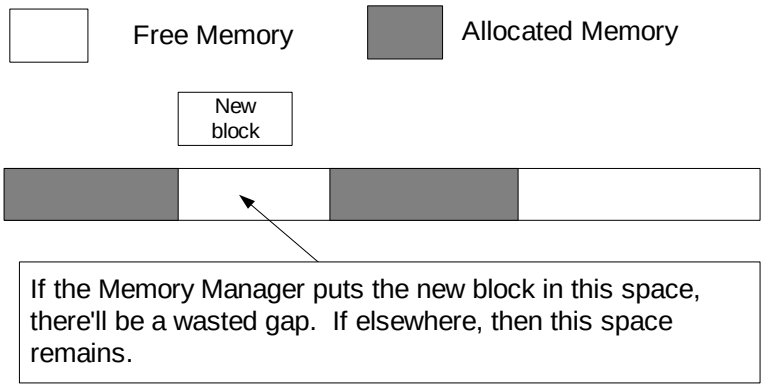
You can also choose between different patterns depending on circumstance. For example, a Smalltalk networking application was required to support a guaranteed minimum throughput, but could improve its performance if it could allocate extra buffer memory. The final design pre-allocated a pool of five 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.



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.

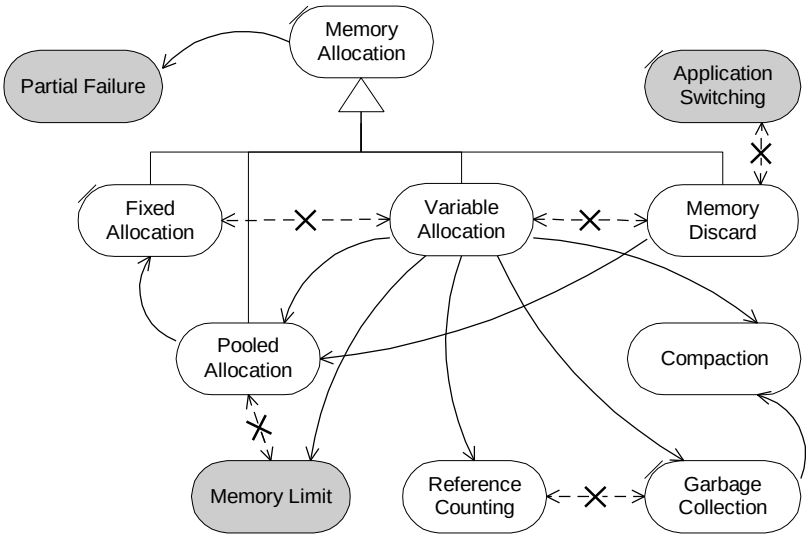


Figure 1: Allocation Pattern Relationships

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

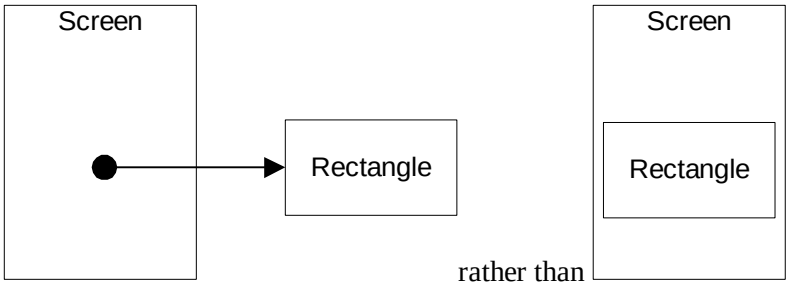
It's straightforward to avoid allocation for classes you've written: just avoid heap operations such as `new` and `delete`. It's more difficult to avoid allocation in library classes, unless the libraries have been designed so that you can override their normal allocation strategies. For example, a Java 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`…").

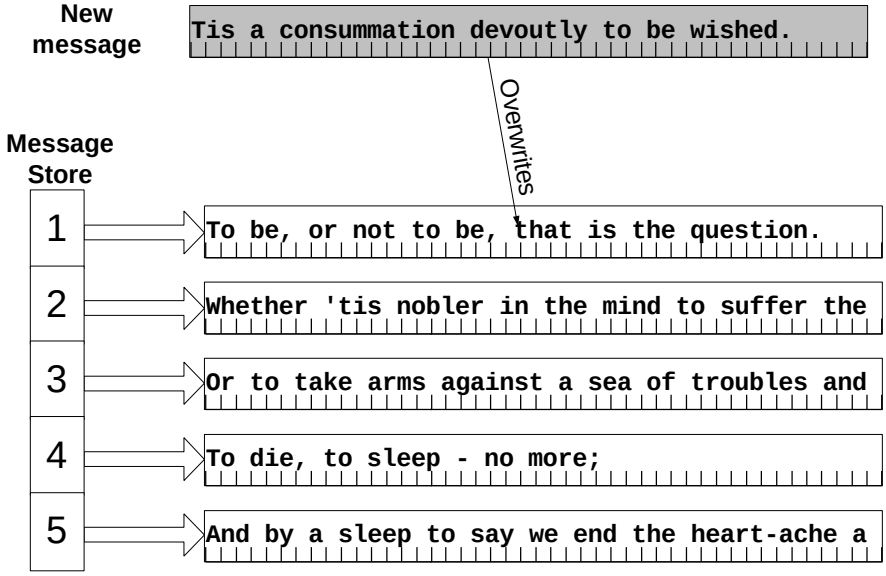


Figure 2: Circular Buffer Message Store

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.

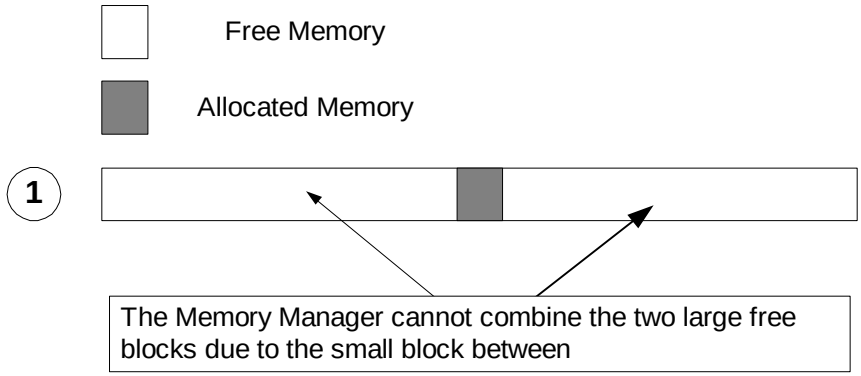
2.2. Exceptions. Far easier for the programmer allocating memory is to signal failure using an exception.. The main benefit of exceptions is that the special case of allocation failure can be handled separately from the main body of the application, while still ensuring the allocation failure does not 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.

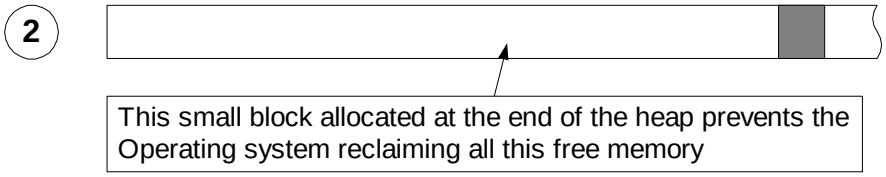
3. Avoiding Fragmentation

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.

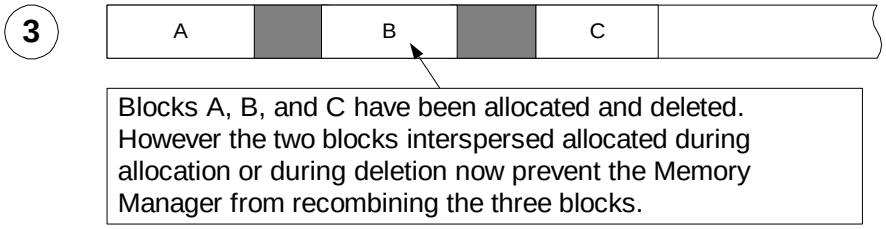
3.1. Variable Sized Allocation. Avoid interspersing allocations of very large and small objects. For example, a small object allocated between two large objects that are then freed will prevent the heap manager combining the two large empty spaces, making it difficult to allocate further larger blocks. 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].



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,



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.



4. Standard Allocation Sizes

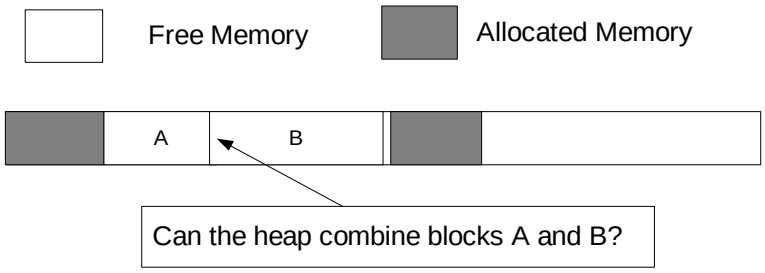
Normally applications tell the heap the sizes of the objects they need to allocate, and the heap allocates memory to store those objects. Another way around fragmentation is for the heap to specify the sizes of memory blocks available to the application. This works if you can afford to waste unused memory 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?

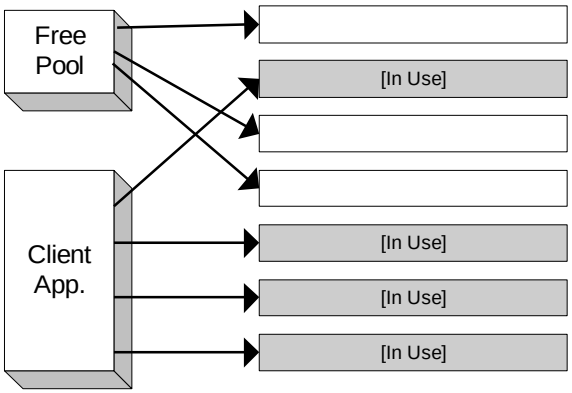
- Your system needs a large number of small objects
- The objects are all roughly the same size.
- The objects need to be allocated and released dynamically.
- You don't, can't, aren't allowed to, or won't used **VARIABLE ALLOCATION**
- Allocating each object individually imposes a large overhead for 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.

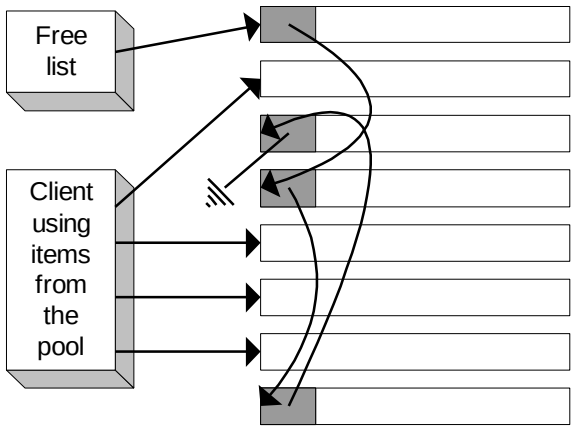


Figure 3: Reusing object memory to implement a free list

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.

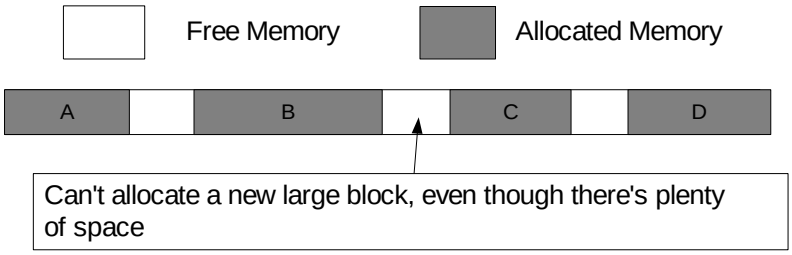


Figure 4: The effect of allocating large buffers

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.

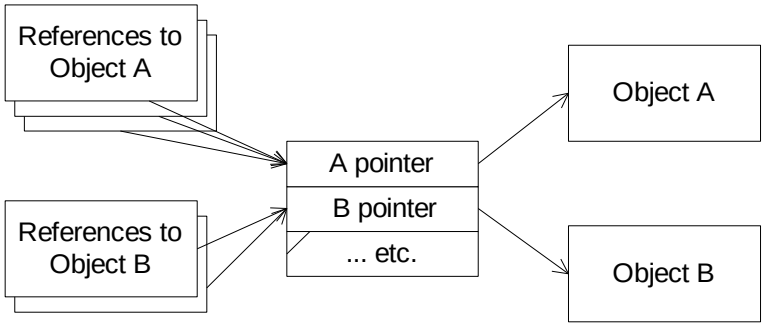


Figure 6: Handles

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

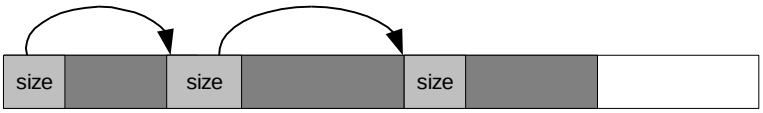
Here are some further issues to consider when implementing the **COMPACTION** pattern:

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.

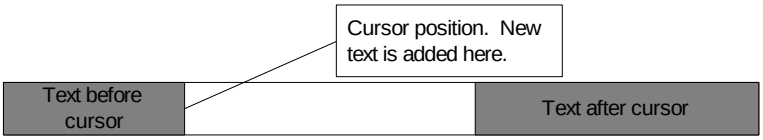
EPOC's Packed Array template class, `CArrayPakFlat`, is one example of this approach [Symbian 1999]. A Packed Array is a sequential array of items; each element in the packed

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**.

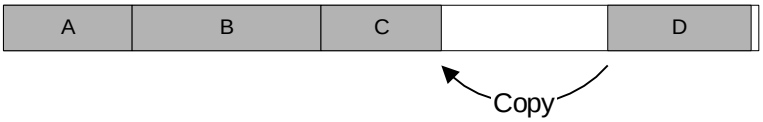
3. Optimising Access to Objects. Using a direct pointer to an object temporarily can increase execution speed compare with indirection through a handle or object table for every access. But, consider the following:

```
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].

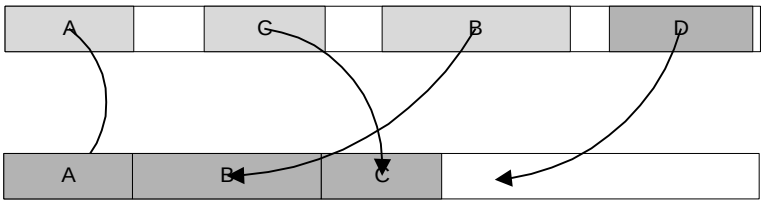
4. Compacting Objects

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.

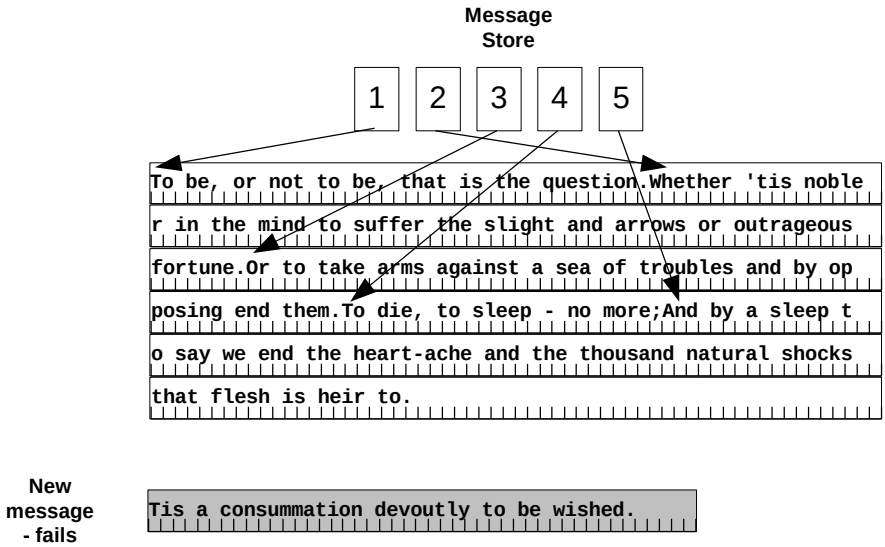
```
template <class T> class Handle {
public:
   Handle( T** p ) : tableEntry( p ) {}
   T* operator->() const { return ptr(); }
   T& operator*() const { return *ptr(); }
   operator T*() const { return ptr(); }
private:
   T* ptr() const { return *tableEntry; }
   T** tableEntry;
};
```

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:



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.

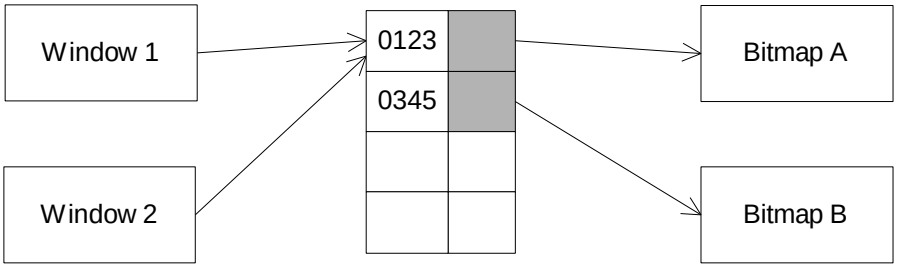
Reference Counting

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.

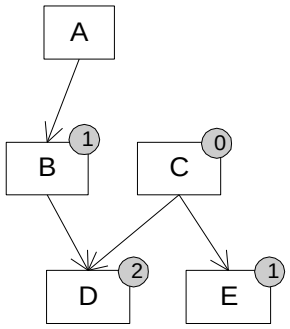
For example, the Strap-It-On Wrist-Mounted PC caches bitmaps displayed in its user interface, so that each bitmap is only stored once, no matter how many windows it is displayed in. The bitmaps are cached in a hash table that maps from bitmap IDs to actual bitmap objects. When a bitmap is no longer required 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.

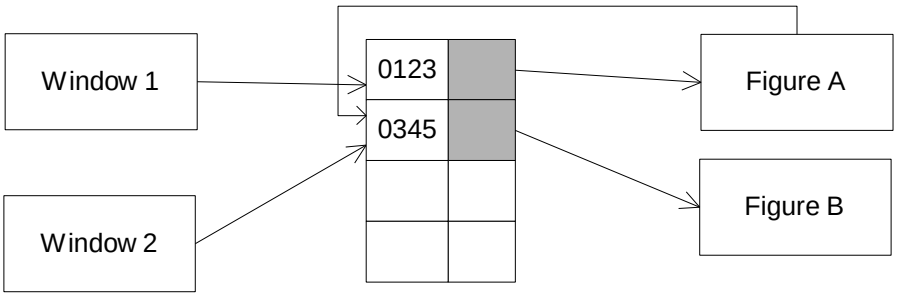
Finally, reference counting doesn't work for cycles of shared objects.

❖ ❖ ❖

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).

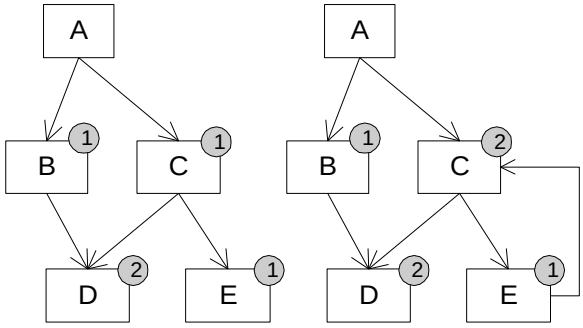


When a reference counted object is deleted, if it refers to other reference counted objects, their reference counts must also be decremented. In the illustration, if Figure A is destroyed, it should decrease the reference count for Figure B.

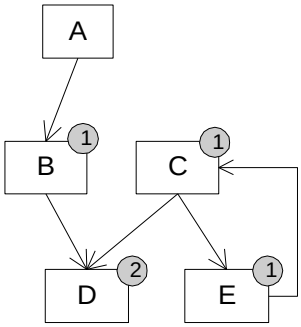
Freeing reference counted objects is a recursive process: once an object's reference count is zero, it must reduce the reference counts of all the objects to which it refers; if those counts also reach zero, the objects must be deleted recursively. Recursive freeing makes reference counting's performance 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.

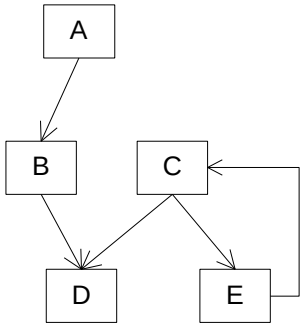


Figure 7: Before Garbage Collection

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.

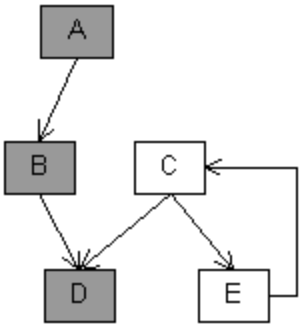


Figure 8: After the mark phase

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.

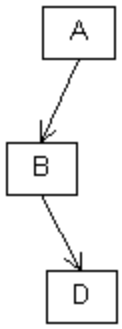


Figure 9: After the sweep phase

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**.

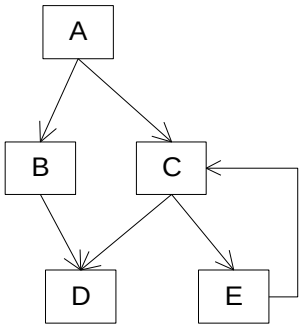
For example, we might implement a Node class with 'left' and 'right' pointers as in Implementation Section "3. Mark-Sweep Garbage Collection". `Node` derives from `CollectableObject`, and every instance must be allocated on the heap:

```
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.