Inside memory management

The choices, tradeoffs, and implementations of dynamic allocation

Jonathan Bartlett (johnnyb@eskimo.com)
Director of Technology
New Medio

16 November 2004

Get an overview of the memory management techniques that are available to Linux™ programmers, focusing on the C language but applicable to other languages as well. This article gives you the details of how memory management works, and then goes on to show how to manage memory manually, how to manage memory semi-manually using referencing counting or pooling, and how to manage memory automatically using garbage collection.

Why memory must be managed

Memory management is one of the most fundamental areas of computer programming. In many scripting languages, you don't have to worry about how memory is managed, but that doesn't make memory management any less important. Knowing the abilities and limitations of your memory manager is critical for effective programming. In most systems languages like C and C++, you have to do memory management. This article covers the basics of manual, semi-automatic, and automatic memory management practices.

Back in the days of assembly language programming on the Apple II, memory management was not a huge concern. You basically had run of the whole system. Whatever memory the system had, so did you. You didn't even have to worry about figuring out how much memory it had, since every computer had the same amount. So, if your memory requirements were pretty static, you just chose a memory range to use and used it.

However, even in such a simple computer you still had issues, especially if you didn't how much memory each part of your program was going to need. If you have limited space and varying memory needs, then you need some way to meet these requirements:

- Determine if you have enough memory to process data
- Get a section of memory from the available memory
- Return a section of memory back to the pool of available memory so it can be used by other parts of the program or other programs

The libraries that implement these requirements are called *allocators*, because they are responsible for allocating and deallocating memory. The more dynamic a program is, the more memory management becomes an issue, and the more important your choice of memory allocator becomes. Let's look at the different methods available to manage memory, their benefits and drawbacks, and the situations in which they work best.

C-style memory allocators

The C programming language provides two functions to fulfill our three requirements:

- **malloc:** This allocates a given number of bytes and returns a pointer to them. If there isn't enough memory available, it returns a null pointer.
- **free:** This takes a pointer to a segment of memory allocated by `malloc`, and returns it for later use by the program or the operating system (actually, some `malloc` implementations can only return memory back to the program, not to the operating system).

Physical and virtual memory

To understand how memory gets allocated within your program, you first need to understand how memory gets allocated to your program from the operating system. Each process on your computer thinks that it has access to all of your physical memory. Obviously, since you are running multiple programs at the same time, each process can't own all of the memory. What happens is that your processes are using *virtual memory*.

Just for an example, let's say that your program is accessing memory address 629. The virtual memory system, however, doesn't necessarily have it stored in RAM location 629. In fact, it may not even be in RAM -- it could even have been moved to disk if your physical RAM was full! Because the addresses don't necessarily reflect the physical location where the memory is located, this is called virtual memory. The operating system maintains a table of virtual address-to-physical address translations so that the computer hardware can respond properly to address requests. And, if the address is on disk instead of in RAM, the operating system will temporarily halt your process, unload other memory to disk, load in the requested memory from disk, and restart your process. This way, each process gets its own address space to play in and can access more memory than you have physically installed.

On 32-bit x86 systems, each process can access 4 GB of memory. Now, most people don't have 4 GB of memory on their systems, even if you include swap, must less 4 GB *per process*. Therefore, when a process loads, it gets an initial allocation of memory up to a certain address, called the *system break*. Past that is unmapped memory -- memory for which no corresponding physical location has been assigned either in RAM or on disk. Therefore, if a process runs out of memory from its initial allocation, it has to request that the operating system "map in" more memory. (Mapping is a mathematical term for one-to-one correspondence -- memory is "mapped" when its virtual address has a corresponding physical location to store it in.)

UNIX-based systems have two basic system calls that map in additional memory:

- **brk:**`brk()` is a very simple system call. Remember the system break, the location that is the edge of mapped memory for the process? `brk()` simply moves that location forward or backward, to add or remove memory to or from the process.
- **mmap:**`mmap()`, or "memory map," is like `brk()` but is much more flexible. First, it can map memory in anywhere, not just at the end of the process. Second, not only can it map virtual addresses to physical RAM or swap, it can map them to files and file locations so that reading and writing memory addresses will read and write data to and from files. Here, however, we are only concerned with `mmap`'s ability to add mapped RAM to our process. `munmap()` does the reverse of `mmap()`.

As you can see, either `brk()` or `mmap()` can be used to add additional virtual memory to our processes. We will use `brk()` in our examples, because it is simpler and more common.

Implementing a simple allocator

If you've done much C programming, you have probably used `malloc()` and `free()` quite a bit. However, you may not have taken the time to think about how they might be implemented in your operating system. This section will show you code for a simplistic implementation of `malloc` and `free` to help demonstrate what is involved with managing memory.

To try out these examples, copy this code listing and paste it into a file called malloc.c. I'll explain the listing a section at a time, below.

Memory allocation on most operating systems is handled by two simple functions:

- `void *malloc(long numbytes)`: This allocates `numbytes` of memory and returns a pointer to the first byte.
- `void free(void *firstbyte)`: Given a pointer that has been returned by a previous `malloc`, this gives the space that was allocated back to the process's "free space."

`malloc_init` is going to be our function to initialize our memory allocator. It does three things: marks our allocator as being initialized, finds the last valid memory address on the system, and sets up the pointer to the beginning of our managed memory. These three variables are global variables:

Listing 1. Global variables of our simple allocator

```
int has_initialized = 0;

void *managed_memory_start;

void *last_valid_address;
```

As mentioned above, the edge of mapped memory -- last valid address -- is often known as the system break or the *current break*. On many UNIX® systems, to find the current system break, you use the function `sbrk(0)`. `sbrk` moves the current system break by the number of bytes in its argument, and then returns the new system break. Calling it with an argument of `0` simply returns the current break. Here is our `malloc` initialization code, which finds the current break and initializes our variables:

Listing 2. Allocator initialization function

```
/* Include the sbrk function */

#include <unistd.h>

void malloc_init()

{

 /* grab the last valid address from the OS */

 last_valid_address = sbrk(0);


 /* we don't have any memory to manage yet, so
  *just set the beginning to be last_valid_address
  */

 managed_memory_start = last_valid_address;

 /* Okay, we're initialized and ready to go */

  has_initialized = 1;

}
```

Now, in order to properly manage memory, we need to be able to track what we are allocating and deallocating. We need to do things like mark blocks as unused after `free` has been called on them, and be able to locate unused blocks when `malloc` is called. Therefore, the start of every piece of memory returned by `malloc` will have this structure at the beginning:

Listing 3. Memory Control Block structure definition

```
struct mem_control_block {

 int is_available;

 int size;

};
```

Now, you might think that this would cause problems for programs calling `malloc` -- how do they know about this struct? The answer is that they don't have to know about it; we will hide it by moving the pointer past this struct before we return it. This will make the returned pointer point to memory that is not used for any other purpose. That way, from the calling programs' perspective, all they get is free, open memory. Then, when they pass the pointer back via `free()`, we simply back up a few memory bytes to find this structure again.

We're going to talk about freeing before we talk about allocating memory, because it's simpler. The only thing we have to do to free memory is to take the pointer we're given, back up `sizeof(struct mem_control_block)` bytes, and mark it as available. Here is the code for that:

Listing 4. Deallocation function

```
void free(void *firstbyte) {

 struct mem_control_block *mcb;

 /* Backup from the given pointer to find the
  * mem_control_block
  */

 mcb = firstbyte - sizeof(struct mem_control_block);

 /* Mark the block as being available */

 mcb->is_available = 1;

 /* That's It!  We're done. */

 return;
}
```

As you can see, in this allocator, freeing memory is done in constant time, using a very simple mechanism. Allocating memory is slightly harder. Here is the outline of the algorithm:

Listing 5. Pseudo-code for the main allocator

```
1. If our allocator has not been initialized, initialize it.

2. Add sizeof(struct mem_control_block) to the size requested.

3. Start at managed_memory_start.

4. Are we at last_valid address?

5. If we are:

   A. We didn't find any existing space that was large enough
      -- ask the operating system for more and return that.

6. Otherwise:

   A. Is the current space available (check is_available from
      the mem_control_block)?

   B. If it is:

      i)   Is it large enough (check "size" from the
           mem_control_block)?

      ii)  If so:

           a. Mark it as unavailable

           b. Move past mem_control_block and return the
              pointer

      iii) Otherwise:

           a. Move forward "size" bytes

           b. Go back go step 4

   C. Otherwise:

      i)   Move forward "size" bytes
      ii)  Go back to step 4
```

We're basically walking through memory using linked pointers looking for open chunks. Here is the code:

Listing 6. The main allocator

```
void *malloc(long numbytes) {

 /* Holds where we are looking in memory */

 void *current_location;

 /* This is the same as current_location, but cast to a
  * memory_control_block
  */

 struct mem_control_block *current_location_mcb;

 /* This is the memory location we will return.  It will
  * be set to 0 until we find something suitable
  */

 void *memory_location;

 /* Initialize if we haven't already done so */

 if(! has_initialized)  {

  malloc_init();

 }

 /* The memory we search for has to include the memory
  * control block, but the users of malloc don't need
  * to know this, so we'll just add it in for them.
  */

 numbytes = numbytes + sizeof(struct mem_control_block);

 /* Set memory_location to 0 until we find a suitable
  * location
  */

 memory_location = 0;

 /* Begin searching at the start of managed memory */

 current_location = managed_memory_start;

 /* Keep going until we have searched all allocated space */

 while(current_location != last_valid_address)

 {

  /* current_location and current_location_mcb point
   * to the same address.  However, current_location_mcb
   * is of the correct type, so we can use it as a struct.
   * current_location is a void pointer so we can use it
   * to calculate addresses.
   */

  current_location_mcb =
   (struct mem_control_block *)current_location;

  if(current_location_mcb->is_available)

  {

   if(current_location_mcb->size >= numbytes)

   {

    /* Woohoo!  We've found an open,
     * appropriately-size location.
     */

    /* It is no longer available */

    current_location_mcb->is_available = 0;

    /* We own it */

    memory_location = current_location;

    /* Leave the loop */

    break;

   }

  }

  /* If we made it here, it's because the Current memory
   * block not suitable; move to the next one
   */

  current_location = current_location +

   current_location_mcb->size;

 }

 /* If we still don't have a valid location, we'll
  * have to ask the operating system for more memory
  */

 if(! memory_location)

 {

  /* Move the program break numbytes further */

  sbrk(numbytes);

  /* The new memory will be where the last valid
   * address left off
   */

  memory_location = last_valid_address;

  /* We'll move the last valid address forward
   * numbytes
   */

  last_valid_address = last_valid_address + numbytes;

  /* We need to initialize the mem_control_block */

  current_location_mcb = memory_location;
  current_location_mcb->is_available = 0;

  current_location_mcb->size = numbytes;

 }

 /* Now, no matter what (well, except for error conditions),
  * memory_location has the address of the memory, including
  * the mem_control_block
  */

 /* Move the pointer past the mem_control_block */

 memory_location = memory_location + sizeof(struct mem_control_block);

 /* Return the pointer */

 return memory_location;

 }
```

And that is our memory manager. Now, we just have to build it and get it to run with our programs.

To build your `malloc`-compatible allocator (actually, we're missing some functions like `realloc()`, but `malloc()` and `free()` are the main ones), run the following command:

Listing 7. Compiling the allocator

```
gcc -shared -fpic malloc.c -o malloc.so
```

This will produce a file named *malloc.so*, which is a shared library containing our code.

On UNIX systems, you can now use your allocator in place of your system `malloc()` by doing this:

Listing 8. Replacing your standard malloc

```
LD_PRELOAD=/path/to/malloc.so

export LD_PRELOAD
```

The `LD_PRELOAD` environment variable causes the dynamic linker to load the symbols of the given shared library before any executable it loads. It also gives precedence to the symbols in the library specified. Therefore, any application we start from now on in this session will be using our `malloc()` and not the system one. A few applications don't use `malloc()`, but they are the exception. Others, which use the other memory-management functions such as `realloc()` or which make poor assumptions about the internal behavior of `malloc()` will likely crash. The ash shell appears to work just fine using our new `malloc()`.

If you want to be sure that your `malloc()` is being used, you should test it by adding calls to `write()` at the entry points of your functions.

Our memory manager leaves a lot to be desired, but it is good for showing what a memory manager needs to do. Some of its drawbacks include the following:

- Since it operates on the system break (a global variable), it cannot coexist with any other allocator or with `mmap`.
- When allocating memory, in a worst-case scenario it will have to walk across *all* of a process's memory; this may include a lot of memory located on disk as well, which means the operating system will have to spend time moving data to and from the disk.
- There is no graceful handling for out-of-memory errors (`malloc` simply assumes success).
- It does not implement many of the other memory functions, such as `realloc()`.
- Because `sbrk()` may give back more memory than we ask for, we leak some memory at the end of the heap.
- The `is_available` flag uses a full 4-byte word, even though it only contains 1 bit of information.
- The allocator is not thread-safe.
- The allocator can't coalesce free space into larger blocks.
- The allocator's simplistic fitting algorithm leads to a lot of potential memory fragmentation.
- I'm sure there are a lot of other problems. That's why it's only an example!

Other malloc implementations

There are many implementations of `malloc()`, each with their own strengths and weaknesses. There are a number of tradeoff decisions when you design an allocator, including:

- Speed of allocation
- Speed of deallocation
- Behavior in a threaded environment
- Behavior when memory is close to filling
- Cache locality
- Bookkeeping memory overhead
- Behavior in Virtual Memory Environments
- Small or large objects
- Real-time guarantees

Each implementation has its own set of benefits and drawbacks. In our simple allocator, it was very slow in allocation but very, very fast in deallocation. Also, because of its poor behavior with virtual memory systems, it works best on large objects.

There are many other allocators available. Some of them include:

- **Doug Lea Malloc:** Doug Lea Malloc is actually an entire family of allocators, including Doug Lea's original allocator, the GNU libc allocator, and `ptmalloc`. Doug Lea's allocator has a basic structure much like our version, but it incorporates indexes to make searching faster and has the ability to combine multiple unused chunks into one large chunk. It also enables caching to make reuse of recently freed memory faster. `ptmalloc` is a version of Doug Lea Malloc that was extended to support multiple threads. A paper describing Doug Lea's Malloc implementation is available in the Resources section later in this article.
- **BSD Malloc:** BSD Malloc, the implementation that was distributed with 4.2 BSD and is included with FreeBSD, is an allocator that allocates objects from pools of objects of pre-determined sizes. It has size classes for object sizes that are a power of two minus a

constant. So, if you request an object of a given size, it simply allocates in whatever size class will fit the object. This provides for a fast implementation, but can waste memory. A paper describing this implementation is available in the Resources section.
- **Hoard:** Hoard was written with the goal of being very fast in a multithreaded environment. Therefore, it is structured around making the best use of locking to keep any process from having to wait to allocate memory. It can dramatically speed up multithreaded processes that do a lot of allocating and deallocating. A paper describing this implementation is available in the Resources section.

These are the best known of the many allocators available. If your program has specific allocation needs, you may prefer to write a custom allocator that matches the way your program allocates memory. However, if you aren't familiar with allocator design, custom allocators can often create more problems than they solve. For a good introduction to the subject, see Donald Knuth's *The Art of Computer Programming Volume 1: Fundamental Algorithms* in section 2.5, "Dynamic Storage Allocation" (see Resources for a link). It is a bit dated, because it doesn't take into account virtual memory environments, but most algorithms are based on the ones presented there.

In C++, you can implement your own allocator on a per-class or per-template basis by overloading `operator new()`. Andrei Alexandrescu's *Modern C++ Design* describes a small object allocator in Chapter 4, "Small Object Allocation" (see Resources for a link).

Shortcomings of malloc()-based memory management

Not only does our memory manager have shortcomings, there are many shortcomings of `malloc()`-based memory management that remain no matter which allocator you use. Managing memory with `malloc()` can be pretty daunting for programs that have long-running storage they need to keep around. If you have lots of references to memory floating around, it is often difficult to know when it should be released. Memory whose lifetime is limited to the current function is fairly easy to manage, but for memory that lives beyond that, it becomes much more difficult. Also, many APIs are unclear as to whether the responsibility for memory management lies with the calling program or the called function.

Because of the problems managing memory, many programs are oriented around their memory management rules. C++'s exception handling makes this task even more problematic. Sometimes it seems that more code is dedicated to managing memory allocation and cleanup than actually accomplishing computational tasks! Therefore, we will examine other alternatives to memory-management.

Semi-automatic memory management strategies

Reference counting

Reference counting is a *semi-automated* memory-management technique, meaning that it requires some programmer support, but it does not require you to know for sure when an object is no longer in use. The reference counting mechanism does that for you.

In reference counting, all shared data structures have a field that contains the number of "references" currently active to that structure. When a procedure is passed a pointer to a data structure, it adds to the reference count. Basically, you are telling the data structure how many locations it is being stored in. Then, when your procedure is finished using it, it decreases the reference count. When this happens, it also checks to see if the count has dropped to zero. If so, it frees the memory.

The advantage to this is that you don't have to follow every path in your program that a given data structure may follow. Each localized reference to it simply increases or decreases the count as appropriate. This prevents it from being freed while it is still in use. However, you must remember to run the reference counting functions whenever you are using a reference-counted data structure. Also, built-in functions and third-party libraries will not know about or be able to use your reference-counting mechanism. Reference counting also has difficulties with structures having circular references.

To implement reference counting, you simply need two functions -- one to increase the reference count, and one to decrease the reference count and free the memory when the count drops to zero.

An example reference counting function set might look like this:

Listing 9. Basic reference counting functions

```
/* Structure Definitions*/

/* Base structure that holds a refcount */

struct refcountedstruct

{

 int refcount;

}

/* All refcounted structures must mirror struct
 * refcountedstruct for their first variables
 */

/* Refcount maintenance functions */

/* Increase reference count */

void REF(void *data)

{

 struct refcountedstruct *rstruct;

 rstruct = (struct refcountedstruct *) data;

 rstruct->refcount++;

}

/* Decrease reference count */

void UNREF(void *data)

{
 struct refcountedstruct *rstruct;

 rstruct = (struct refcountedstruct *) data;

 rstruct->refcount--;

 /* Free the structure if there are no more users */

 if(rstruct->refcount == 0)

 {

  free(rstruct);

 }

}
```

`REF` and `UNREF` might be more complicated, depending on what you wanted to do. For example, you may want to add locking for a multithreaded program, and you may want to extend `refcountedstruct` so that it also includes a pointer to a function to call before freeing the memory (like a destructor in object-oriented languages -- this is *required* if your structures contain pointers).

When using `REF` and `UNREF`, you need to obey these rules for pointer assignments:

- `UNREF` the value that the left-hand-side pointer is pointing to before the assignment.
- `REF` the value that that the left-hand-side pointer is pointing to after the assignment.

In functions that are passed refcounted structures, the functions need to follow these rules:

- REF every pointer at the beginning of the function.
- UNREF every pointer at the end of the function.

Here is a quick example of code using reference counting:

Listing 10. Example using reference counting

```
/* EXAMPLES OF USAGE */


/* Data type to be refcounted */

struct mydata

{

 int refcount; /* same as refcountedstruct */

 int datafield1; /* Fields specific to this struct */

 int datafield2;

 /* other declarations would go here as appropriate */

};


/* Use the functions in code */

void dosomething(struct mydata *data)
```

```
{

 REF(data);

 /* Process data */

 /* when we are through */

 UNREF(data);

}


struct mydata *globalvar1;

/* Note that in this one, we don't decrease the
 * refcount since we are maintaining the reference
 * past the end of the function call through the
 * global variable
 */

void storesomething(struct mydata *data)

{

 REF(data); /* passed as a parameter */

 globalvar1 = data;

 REF(data); /* ref because of Assignment */

 UNREF(data); /* Function finished */

}
```

Since reference counting is so simple, most programmers implement it themselves rather than using libraries. They do, however, depend on low-level allocators like `malloc` and `free` to actually allocate and release their memory.

Reference counting is used quite a bit in high-level languages like Perl to do memory management. In those languages, the reference counting is handled automatically by the language, so that you don't have to worry about it at all except for writing extension modules. This takes away some speed as everything must be reference counted, but adds quite a bit of safety and ease of programming. Here are the benefits:

- It has a simple implementation.
- It is easy to use.
- Since the reference is part of the data structure, it has good cache locality.

However, it also has its drawbacks:

- It requires that you never forget to call the reference counting functions.
- It will not release structures that are a part of a circular data structure.
- It slows down nearly every pointer assignment.
- You must take additional precautions when using exception-handling (like `try` or `setjmp()`/`longjmp()`) while using reference counted objects.
- It requires extra memory to handle the references.

- The reference counter takes up the first position in the structure, which is the fastest to access on most machines.
- It is slower and more difficult to do in a multithreaded environment.

C++ can mitigate some of the programmer error by using *smart pointers*, which can handle pointer-handling details such as reference counting for you. However, if you have to use any legacy code that can't handle your smart pointers (such as linkage to a C library), it usually degenerates into a mess that is actually more difficult and twisted than if you didn't use them. Therefore, it is usually only useful for C++-only projects. If you want to use smart pointers, you really need to read the "Smart Pointers" chapter from Alexandrescu's *Modern C++ Design* book.

Memory pools

Memory pools are another method to semi-automate memory management. Memory pools help automate memory management for programs that go through specific stages, each of which has memory that is allocated for only specific stages of processing. For example, many network server processes have lots of per-connection memory allocated -- memory whose maximum lifespan is the life of the current connection. Apache, which uses pooled memory, has its connections broken down into stages, each of which have their own memory pool. At the end of the stage, the entire memory pool is freed at once.

In pooled memory management, each allocation specifies a pool of memory from which it should be allocated. Each pool has a different lifespan. In Apache, there is a pool that lasts the lifetime of the server, one that lasts the lifetime of the connection, one that lasts the lifetime of the requests, and others as well. Therefore, if I have a series of functions that will not generate any data that lasts longer than the connection, I can just allocate it all from the connection pool, knowing that at the end of the connection, it will be freed automatically. Additionally, some implementations allow registering *cleanup functions*, which get called right before the memory pool is cleared, to do any additional tasks that need to be performed before the memory is cleared (similar to destructors, for you object-oriented folks).

To use pools in your own programs, you can either use GNU libc's obstack implementation or Apache's Apache Portable Runtime. GNU obstacks are nice, because they are included by default in GNU-based Linux distributions. The Apache Portable Runtime is nice because it has a lot of other utilities to handle all aspects of writing multiplatform server software. To learn more about GNU obstacks and Apache's pooled memory implementation, see the links to their documentation in the Resources section.

The following hypothetical code listing shows how obstacks are used:

Listing 11. Example code for obstacks

```
#include <obstack.h>

#include <stdlib.h>

/* Example code listing for using obstacks */

/* Used for obstack macros (xmalloc is
   a malloc function that exits if memory
   is exhausted */
```

```
#define obstack_chunk_alloc xmalloc
#define obstack_chunk_free free
/* Pools */
/* Only permanent allocations should go in this pool */
struct obstack *global_pool;
/* This pool is for per-connection data */
struct obstack *connection_pool;
/* This pool is for per-request data */
struct obstack *request_pool;
void allocation_failed()
{
 exit(1);
}
int main()
{
 /* Initialize Pools */
 global_pool = (struct obstack *)
  xmalloc (sizeof (struct obstack));
 obstack_init(global_pool);
 connection_pool = (struct obstack *)
  xmalloc (sizeof (struct obstack));
 obstack_init(connection_pool);
 request_pool = (struct obstack *)
  xmalloc (sizeof (struct obstack));
 obstack_init(request_pool);
 /* Set the error handling function */
 obstack_alloc_failed_handler = &allocation_failed;
 /* Server main loop */
 while(1)
 {
  wait_for_connection();
  /* We are in a connection */
  while(more_requests_available())
  {

   /* Handle request */

   handle_request();

   /* Free all of the memory allocated

    * in the request pool

    */

   obstack_free(request_pool, NULL);

  }

  /* We're finished with the connection, time

   * to free that pool

   */

  obstack_free(connection_pool, NULL);

 }

}

int handle_request()

{

 /* Be sure that all object allocations are allocated
  * from the request pool
  */

 int bytes_i_need = 400;

 void *data1 = obstack_alloc(request_pool, bytes_i_need);

 /* Do stuff to process the request */

 /* return */

 return 0;

}
```

Basically, after each major stage of operation, the obstack for that stage is freed. Note, however, that if a procedure needs to allocate memory that will last longer than the current stage, it can use a longer-term obstack as well, such as the connection or the global one. The `NULL` that is passed to `obstack_free()` indicates that it should free the entire contents of the obstack. Other values are available, but they usually are not as useful.

Benefits of using pooled memory allocation include the following:

- It is simple to manage memory for the application.
- Memory allocation and deallocation is much faster, because it is all done a pool at a time. Allocation can be done in O(1) time, and pool release is close (it's actually O(n) time, but divided by a huge factor that makes it O(1) in most cases).

- Error-handling pools can be preallocated so that your program can still recover if regular memory is exhausted.
- There are standard implementations that are very easy to use.

The drawbacks for pooled memory are:

- Memory pools are only useful for programs that operate in stages.
- Memory pools often do not work well with third-party libraries.
- If program structure changes, the pools may have to be modified, which may lead to a redesign of the memory management system.
- You must remember which pool you need to allocate from. In addition, if you get this wrong, it can be hard to catch.

Garbage collection

Garbage collection is the fully automatic detection and removal of data objects that are no longer in use. Garbage collectors are usually run when the available memory drops below a specific threshold. Generally, they start off with a "base" set of data that is known to be available to the program -- stack data, global variables, and registers. They then try to trace through every piece of data linked through those. Everything the collector finds is good data; everything that it doesn't find is garbage and can be destroyed and reused. To manage memory effectively, many types of garbage collectors require knowledge of the layout of pointers within data structures, and therefore have to be a part of the language itself to function properly.

Types of collectors

- **Copying:** These divide memory storage into two parts and allow data to live only on one side. Periodically, they start copying data from one side to the other starting with "base" elements. The newly occupied section of memory now becomes active, and everything on the other side is considered garbage. Also, when this copying occurs, all of the pointers have to be updated to point to the new location of each memory item. Therefore, to use this method of garbage collection, the collector must be integrated with the programming language.
- **Mark and sweep:** Each piece of data is marked with a tag. Occasionally, all tags are set to 0, and the collector walks through the data starting with "base" elements. As it encounters memory, it marks the tag as 1. Everything not tagged 1 at the end is considered garbage and reused for later allocations.
- **Incremental:** Incremental garbage collectors do not require a full run through all data objects. Running through all of memory causes problems because of the all-at-once wait during the collection period and because of the cache problems associated with accessing all current data (everything has to be paged-in). Incremental collectors avoid these problems.
- **Conservative:** Conservative garbage collectors do not need to know anything about the structure of your data to manage memory. They simply look at all data bytes and assume they *could* all be pointers. So, if a sequence of bytes could be a pointer to a piece of allocated memory, it marks it as being referenced. This sometimes leads to problems where memory that isn't referenced is collected if, for example, an integer field contained a value that was the address of allocated memory. However, this is a fairly rare occurrence, and it only wastes

a little memory. Conservative collectors have the advantage that they can be integrated with any programming language.

Hans Boehm's conservative garbage collector is one of the most popular garbage collectors available, because it's free and it's both conservative and incremental. You can use it as a drop-in replacement for your system allocator (using `malloc`/`free` instead of its own API) by building it with `--enable-redirect-malloc`. In fact, if you do this, you can use the same `LD_PRELOAD` trick that we used for our simple allocator to enable garbage collection in almost any program on your system. If you suspect a program is leaking memory, you can use this garbage collector to keep the process size down. Many people used this technique in the early days of Mozilla when it leaked memory heavily. This garbage collector runs under both Windows® and UNIX.

Some advantages of garbage collection:

- You never have to worry about double-freeing memory or object lifetimes.
- You can, with some collectors, use the same API that you used for normal allocation.

The drawbacks include:

- With most collectors, you have no say when your memory is going to be freed.
- In many cases, garbage collection is slower than other forms of memory management.
- Bugs caused by garbage collection errors are hard to debug.
- You can still have memory leaks if you forget to set unused pointers to null.

Conclusion

It's a world of tradeoffs: performance, ease-of-use, ease-of-implementation, and threading capability, just to name a few. There are numerous patterns of memory management at your disposal to match your project requirements. Each pattern has a wide range of implementations, each of which has its benefits and drawbacks. Using the default techniques for your programming environment is fine for many projects, but knowing the available options will help you when your project has special needs. This table compares the memory management strategies covered in this article.

Table 1. Comparison of memory allocation strategies

Strategy	Allocation speed	Deallocation speed	Cache locality	Ease of use	Generality	Usable in real time	SMP and thread-friendly
Custom allocator	Depends on implementation	Depends on implementation	Depends on implementation	Very difficult	None	Depends on implementation	Depends on implementation
Simple allocator	Fast for small memory usage	Very fast	Poor	Easy	Very	No	No
GNU `malloc`	Moderate	Fast	Moderate	Easy	Very	No	Moderate

Hoard	Moderate	Moderate	Moderate	Easy	Very	No	Yes
Reference counting	N/A	N/A	Excellent	Moderate	Moderate	Yes (depends on `malloc` implementation)	Depends on implementation
Pooling	Moderate	Very fast	Excellent	Moderate	Moderate	Yes (depends on `malloc` implementation)	Depends on implementation
Garbage collection	Moderate (slow when collection occurs)	Moderate	Poor	Moderate	Moderate	No	Rarely
Incremental garbage collection	Moderate	Moderate	Moderate	Moderate	Moderate	No	Rarely
Incremental conservative garbage collection	Moderate	Moderate	Moderate	Easy	Very	No	Rarely

Resources

Learn

- The obstacks section of the GNU C Library manual gives the programming interface for obstacks.
- The Apache Portable Runtime documentation describes the interface to their pooled allocator.
- Doug Lea's Malloc is one of the more popular memory allocators.
- BSD Malloc is used in most BSD-based systems.
- ptmalloc is derived from Doug Lea's malloc and is used in GLIBC.
- GNU Memory-Mapped Malloc (part of GDB) is a `malloc` implementation that is based on `mmap()`.
- GNU Obstacks (part of GNU Libc) is the most widely installed pooled allocator, since it's on every glibc-based system.
- Apache's pooled allocator (in the Apache Portable Runtime) is the most widely used pooled allocator.
- NetBSD also has its own pooled allocator.
- The Loki C++ Library has a number of generic patterns implemented for C++, including smart pointers and a custom small-object allocator.
- The Hahns Boehm Conservative Garbage Collector is the most popular open source garbage collector, which can be used in regular C/C++ programs.
- A New Virtual Memory Implementation for Berkeley UNIX by Marshall Kirk McKusick and Michael J. Karels discusses BSD's VM system.
- Mel Gorman's Linux VM Documentation discusses the Linux VM system.
- Malloc in Modern Virtual Memory Environments by Poul-Henning Kamp talks about BSD's `malloc` and how it interacts with BSD virtual memory.
- Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel by Marshall Kirk McKusick and Michael J. Karels discusses kernel-level allocators.
- A Memory Allocator by Doug Lea gives an overview of the design and implementation of allocators, including design choices and tradeoffs.
- Memory Management for High-Performance Applications by Emery D. Berger talks about custom memory management and how it affects high-performance applications.
- Some Storage Management Techniques for Container Classes by Doug Lea describes writing custom allocators for C++ classes.
- The Measured Cost of Garbage Collection by Benjamin Zorn presents hard data on garbage allocation and performance.
- Memory Allocation Myths and Half-Truths by Hans-Juergen Boehm presents the myths surrounding garbage collection.
- Space Efficient Conservative Garbage Collection by Hans-Juergen Boehm is a paper describing his garbage collector for C/C++.
- The Memory Management Reference contains numerous references and links to papers on memory management.
- OOPS Group Papers on Memory Management and Memory Hierarchies is a great set of technical papers on the subject.

- Memory Management in C++ discusses writing custom allocators for C++.
- Programming Alternatives: Memory Management discusses several choices programmers have for memory management.
- Richard Jones's Garbage Collection Bibliography has links to any paper you ever wanted about garbage collection.
- *C++ Pointers and Dynamic Memory Management* by Michael Daconta covers numerous techniques on memory management.
- *Memory as a Programming Concept in C and C++* by Frantisek Franek discusses techniques and tools for developing effective memory use and gives the role of memory-related errors the prominence it deserves in computer programming.
- *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* by Richard Jones and Rafael Lins describes the most common algorithms for garbage collection in use.
- Section 2.5, "Dynamic Storage Allocation" from *Fundamental Algorithms*, Volume 1 of *The Art of Computer Programming* by Donald Knuth describes several techniques for implementing basic allocators.
- Section 2.3.5, "Lists and Garbage Collection" from *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming* by Donald Knuth discusses garbage collection algorithms for lists.
- Chapter 4, "Small Object Allocation" from *Modern C++ Design* by Andrei Alexandrescu describes a high-speed small-object allocator that is quite a bit more efficient than the C++ standard allocator.
- Chapter 7, "Smart Pointers" from *Modern C++ Design* by Andrei Alexandrescu describes the implementation of smart pointers in C++.
- Jonathan's Chapter 8, "Intermediate Memory Topics" from *Programming from the Ground Up* contains an assembly-language version of the simple allocator used in this article.
- Self-manage data buffer memory (developerWorks, January 2004) outlines a pseudo-C implementation of a self-managing, abstract data buffer for managing memory.
- A framework for the user defined malloc replacement feature (developerWorks, February 2002) shows how to take advantage of a facility in AIX that lets you replace the memory subsystem with one of your own design.
- Mastering Linux debugging techniques (developerWorks, August 2002) describes debugging methods you can use in four different scenarios: segmentation faults, memory overruns, memory leaks, and hangs.
- In Handling memory leaks in Java programs (developerWorks, February 2001), learn what causes Java memory leaks and when they should be of concern.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and spaces.

About the author

Jonathan Bartlett

Jonathan Bartlett is the author of the book *Programming from the Ground Up*, an introduction to programming using Linux assembly language. He is the lead developer at New Medio, developing Web, video, kiosk, and desktop applications for clients.