Building your own memory manager for C/C++ projects

Arpan Sen
Rahul Kumar Kardam

February 19, 2008

Performance optimization of code is serious business. It's fairly common to see a piece of functionally correct software written in C or C++ that takes way too much memory, time, or, in the worst case, both. As a developer, one of the most powerful tools that C/C++ arms you with to improve processing time and prevent memory corruption is the control over how memory is allocated or deallocated in your code. This tutorial demystifies memory management concepts by telling you how to create your very own memory manager for specific situations.

Before you start

Learn what to expect from this tutorial and how to get the most out of it.

About this tutorial

This tutorial takes a basic approach to building a memory manager for any application. It explains why a memory manager is needed and provides a few ways in which you can write customized memory managers for an application, catering to its specific needs.

Objectives

In this tutorial, you'll learn what considerations you need to take before designing a memory manager, the specific techniques available for creating such a memory manager, and, finally, the method for creating it. You'll also learn about the advantages and disadvantages of various types of memory manager designs.

Prerequisites

This tutorial is written for Linux® or UNIX® programmers whose skills and experience are at a beginning to intermediate level. You should have a general familiarity with using a UNIX command-line shell and a working knowledge of the C/C++ language. Any additional knowledge of internal workings of routines such `malloc`, `calloc`, `free`, `memcpy`, and `memset` (that is, routines that deal with memory allocation, deallocation, and content modification) is a plus.

System requirements

To run the examples in this tutorial, you need a Linux or UNIX box that has the g++ compiler toolchain installed on it. A sufficiently large amount of RAM (approximately 256 MB) is also required.

Why create a custom memory manager?

To appreciate how the control over memory allocation helps your code run faster, first recollect the basics of memory management in C/C++. The standard library functions `malloc`, `free`, `calloc`, and `realloc` in C and the `new`, `new [ ]`, `delete`, and `delete [ ]` operators in C++ form the crux of the memory management in these two languages. With this in mind, there are several things worth your attention here.

Functions such as `malloc` and `new` are general-purpose memory allocators. Your code may be single-threaded, but the `malloc` function it is linked to can handle multithreaded paradigms just as well. It is this extra functionality that degrades the performance of these routines.

In their turn, `malloc` and `new` make calls to the operating system kernel requesting memory, while `free` and `delete` make requests to release memory. This means that the operating system has to *switch between user-space code and kernel code every time a request for memory is made*. Programs making repeated calls to `malloc` or `new` eventually run slowly because of the repeated context switching.

Memory that is allocated in a program and subsequently not needed is often unintentionally left undeleted, and C/C++ don't provide for automatic garbage collection. This causes the memory footprint of the program to increase. In the case of really big programs, performance takes a severe hit because available memory becomes increasingly scarce and hard-disk accesses are time intensive.

Design goals

Your memory manager should satisfy the following design goals:

- Speed
- Robustness
- User convenience
- Portability

Speed

The memory manager must be faster than the compiler-provided allocators. Repeated allocations and deallocations should not slow down the code. If possible, the memory manager should be optimized for handling certain allocation patterns that occur frequently in the code.

Robustness

The memory manager must return all the memory it requested to the system before the program terminates. That is, there should be no memory leaks. It should also be able to handle erroneous cases (for example, requesting too large a size) and bail out gracefully.

User convenience

Users should need to change only a minimal amount of code when integrating the memory manager into their code.

Portability

The memory manager should be easily portable across systems and not use platform-dependant memory management features.

Useful strategies for creating a memory manager

The following strategies are useful when creating a memory manager:

- Request large memory chunks.
- Optimize for common request sizes.
- Pool deleted memory in containers.

Request large memory chunks

One of the most popular memory management strategies is to request for large memory chunks during program startup and then intermittently during code execution. Memory allocation requests for individual data structures are carved out from these chunks. This results in far fewer system calls and boosts the performance time.

Optimize for common request sizes

In any program, certain specific request sizes are more common than others. Your memory manager will do well if it's optimized to handle these requests better.

Pool deleted memory in containers

Deleted memory during program execution should be pooled in containers. Further requests from memory should then be served from these containers. If a call fails, memory access should be delegated to any one of the large chunks allocated during program start. While memory management is primarily meant to speed up program execution and prevent memory leaks, this technique can potentially result in a lower memory footprint of the program because deleted memory is being reused. Yet another reason to write your own memory allocator!

Timing C++ new/delete operators

We'll start with a simple example. Say your code uses a class called `Complex` that is meant to represent complex numbers, and it uses the language provided by the `new` and `delete` operators, as shown in Listing 1.

Listing 1. C++ code for the Complex class

```
class Complex
  {
```

```
  public:
  Complex (double a, double b): r (a), c (b) {}
  private:
  double r; // Real Part
  double c; // Complex Part
  };

int main(int argc, char* argv[])
  {
  Complex* array[1000];
  for (int i = 0;i  <  5000; i++) {
    for (int j = 0; j  <  1000; j++) {
      array[j] = new Complex (i, j);
      }
    for (int j = 0; j  <  1000; j++) {
      delete array[j];
      }
    }
  return 0;
  }
```

Each iteration of the outermost loop causes 1000 allocations and deallocations. 5000 such iterations result in 10 million switches between user and kernel code. Compiling this test on a gcc-3.4.6 in a Solaris 10 machine took an average of 3.5 seconds. This is the baseline performance metric for the compiler-provided implementation of the global `new` and `delete` operators. To create a custom memory manager for the `Complex` class that improves the compiler implementation, you need to override the `Complex` class-specific `new` and `delete` operators.

New/Delete: A detailed look

In C++, organized memory management is all about overloading the `new` or `delete` operator. Different classes in the code might be candidates for different memory allocation policies, which implies that a class-specific `new` is required. Otherwise, the `new` or `delete` global operator must be overridden. Operator overloading can be done in either of the forms, as shown in Listing 2.

Listing 2. Overloading the new or delete operator

```
void* operator new(size_t size);
void   operator delete(void* pointerToDelete);
-OR-
void* operator new(size_t size, MemoryManager& memMgr);
void   operator delete(void* pointerToDelete, MemoryManager& memMgr);
```

The overridden `new` operator is responsible for allocating raw memory of the size specified in the first argument, and the `delete` operator frees this memory. Note that these routines are meant only to allocate or deallocate memory; they don't call constructors or destructors, respectively. A constructor is invoked on the memory allocated by the `new` operator, while the `delete` operator is called only after the destructor is called on an object.

The second variant of `new` is the `placement new` operator, which accepts a `MemoryManager` argument -- basically a data structure to allocate raw memory on which the constructor of an object is finally called. For the purposes of this tutorial, we recommend using the first variant of `new` or `delete` because the placement variation results in a massive proliferation of `MemoryManager&` arguments in the user code, violating the design goal of user convenience.

We use the `MemoryManager` class to do the actual allocation or deallocation with the `new` and `delete` operator routines serving as wrappers for the following `MemoryManager` routines, as shown in Listing 3: `MemoryManager gMemoryManager; // Memory Manager, global variable`.

Listing 3. The new, new[ ], delete, and delete[ ] operators as wrappers

```
MemoryManager gMemoryManager; // Memory Manager, global variable

void* operator new (size_t size)
  {
  return gMemoryManager.allocate(size);
  }

void* operator new[ ] (size_t size)
  {
  return  gMemoryManager.allocate(size);
  }

void operator delete (void* pointerToDelete)
  {
  gMemoryManager.free(pointerToDelete);
  }

void operator delete[ ] (void* arrayToDelete)
  {
  gMemoryManager.free(arrayToDelete);
  }
```

Notes:

- The size passed to the `new[ ]` operator is the size of each individual element of the array multiplied by the number of elements of the array.
- This pointer is not available in any `new`, `delete`, `new[ ]`, or `delete[ ]`class-specific operator. In effect, all four of these routines serve as static methods. You must always keep this in mind for your design.
- Instead of a using a global variable to declare the memory manager, you could, of course, use a singleton.

Based on the discussion so far, Listing 4 is the basic interface of the memory manager class

Listing 4. Memory manager interface

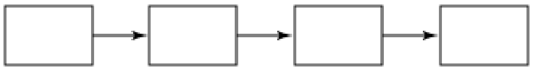
```
class IMemoryManager
  {
  public:
    virtual void* allocate(size_t) = 0;
    virtual void   free(void*) = 0;
  };

class MemoryManager : public IMemoryManager
  {
  public:
    MemoryManager( );
    virtual ~MemoryManager( );
    virtual void* allocate(size_t);
    virtual void   free(void*);
  };
```

We also like to make the `allocate` and `free` routines inline for quicker dispatch.

Our first memory manager in a single-threaded environment for the Complex type

We designed our first memory manager for this tutorial keeping in mind the principles we've discussed so far. To keep matters simple, this memory manager is customized specifically for objects of type `Complex` and works only in single-threaded environments. The basic idea is to keep a pool of `Complex` objects inside the memory manager available and have future allocations happen from this pool. If the number of `Complex` objects that need to be created exceeds the number of objects in the pool, the pool is expanded. Deleted objects are returned to this pool. Figure 1 is a good illustration of things to come.

Figure 1. Creating a pool of Complex objects



Each block in the pool serves a dual purpose:

- It stores a `Complex` object.
- It must be able to connect itself to the next block in the pool.

Storing a pointer inside the `Complex` data structure is not an option because that increases the overall memory footprint of the design. A better option is to wrap the private variables of the `Complex` data structure into a structure and create a union with the `Complex` pointer. When used as part of the pool, the pointer is used to point to the next free element in the pool. When used as standalone `Complex` object, the structure holding the real and complex part is utilized. Listing 5 shows the modified data structure.

Listing 5. Modified data structure to store Complex* with no extra overhead

```
class Complex
  {
  public:
    Complex (double a, double b): r (a), c (b) {}
    private:
    union {
      struct {
        double r; // Real Part
        double c; // Complex Part
        };
      Complex* next;
    };
  };
```

However, this strategy violates the design goal of user convenience because we expect to make minimal changes in the original data structure while integrating the memory manager. Improving on this, we designed a new `FreeStore` data structure that is a wrapper data structure meant to serve as a pointer when kept as part of the pool and as a `Complex` object otherwise. Listing 6 shows the data structure for the `FreeStore` object.

Listing 6. Data structure for FreeStore object

```
struct FreeStore
  {
  FreeStore* next;
  };
```

The pool is then a linked list of `FreeStore` objects, each of which can point to the next element in the pool and be used as a `Complex` object. The `MemoryManager` class must keep a pointer to the head of the first element of the free pool. It should have private methods to expand the size of the pool when needed and a method to clean up the memory when the program terminates. Listing 7 contains the modified data structure for a `Complex` class with `FreeStore` functionality.

Listing 7. Modified data structure for a Complex class with FreeStore functionality

```
#include <sys/types.h>

class MemoryManager: public IMemoryManager
  {
  struct FreeStore
    {
     FreeStore *next;
    };
  void expandPoolSize ();
  void cleanUp ();
  FreeStore* freeStoreHead;
  public:
    MemoryManager () {
      freeStoreHead = 0;
      expandPoolSize ();
      }
    virtual ~MemoryManager () {
      cleanUp ();
      }
    virtual void* allocate(size_t);
    virtual void   free(void*);
  };

MemoryManager gMemoryManager;

class Complex
  {
  public:
    Complex (double a, double b): r (a), c (b) {}
    inline void* operator new(size_t);
    inline void   operator delete(void*);
  private:
    double r; // Real Part
    double c; // Complex Part
  };
```

The following is the pseudocode for the memory allocation:

1. If the free-store hasn't been created yet, create the free-store, and then go to step 3.
2. If the free-store has been exhausted, create a new free-store.
3. Return the first element of the free-store and mark the next element of the free-store as the free-store head.

The following is the pseudocode for the memory deletion:

1. Make the next field in the deleted pointer point to the current free-store head.
2. Mark the deleted pointer as the free-store head

Listing 8 contains the sources for the `new` and `delete` operators for the `Complex` class. Listing 9 shows the expand and cleanup methods of the free-store. A problem still remains though. Can you spot it?

Listing 8. Custom memory allocation or deallocation for the Complex class

```
inline void* MemoryManager::allocate(size_t size)
  {
  if (0 == freeStoreHead)
    expandPoolSize ();

  FreeStore* head = freeStoreHead;
  freeStoreHead = head->next;
  return head;
  }

inline void MemoryManager::free(void* deleted)
  {
  FreeStore* head = static_cast <FreeStore*> (deleted);
  head->next = freeStoreHead;
  freeStoreHead = head;
  }

void* Complex::operator new (size_t size)
  {
  return gMemoryManager.allocate(size);
  }

void Complex::operator delete (void* pointerToDelete)
  {
  gMemoryManager.free(pointerToDelete);
  }
```

The creation of the free-store is not trivial. The trick lies in understanding that the same `FreeStore*` pointer is used as a `Complex` object. Therefore, the size requested for individual `FreeStore` pointers must be the size of whichever is greater, the `FreeStore*` or the `Complex`, as shown in Listing 9.

Listing 9. Code for expanding and cleaning up the free-store

```
#define POOLSIZE 32

void MemoryManager::expandPoolSize ()
  {
  size_t size = (sizeof(Complex) > sizeof(FreeStore*)) ?
    sizeof(Complex) : sizeof(FreeStore*);
  FreeStore* head = reinterpret_cast <FreeStore*> (new char[size]);
  freeStoreHead = head;

  for (int i = 0; i < POOLSIZE; i++) {
    head->next = reinterpret_cast <FreeStore*> (new char [size]);
    head = head->next;
    }

  head->next = 0;
  }
```

```
void MemoryManager::cleanUp()
  {
  FreeStore* nextPtr = freeStoreHead;
  for (; nextPtr; nextPtr = freeStoreHead) {
    freeStoreHead = freeStoreHead->next;
    delete [] nextPtr; // remember this was a char array
    }
  }
```

Interesting note

The elements of the free-store are still created using the global `new` and `delete` operators. However, you can use the `malloc` and `free` combination here as well. The average time performance remains the same in both cases.

That's it! You've created your customized memory manager for the `Complex` class. Recall that the baseline test took 3.5 seconds. Compiling and running that same test (no need to make any changes to the client code in the main routine) now shows that the program execution takes a stunning 0.67 seconds! Why is there such a dramatic performance improvement? Primarily for two reasons:

- The number of switches between the user and kernel code has been reduced markedly because it reuses the deleted memory by pooling it back to the free-store.
- This memory manager is a single-threaded allocator that suits the purposes of the execution. We don't intend to run the code in a multithreaded environment, yet.

Earlier we asked you to spot a problem in the design. The issue is that if a `Complex` object created using `new` is not deleted, then there's no way to reclaim the lost memory in this design. This is, of course, the same if the developer uses the compiler-provided `new` and `delete` global operators. However, a memory manager is not just about performance improvement, it should also be able to remove memory leaks from the design. The `cleanUp` routine is only meant to return memory back to the operating system for cases in which the `Complex` object created using `new` has been explicitly deleted. This problem can only be resolved by requesting bigger memory blocks from the operating system during program initialization and storing them. Requests for memory using `Complex`'s `new` operator should be carved out from these blocks. The `cleanUp` routine should work on freeing up these blocks as a whole, not individual objects that are created from these memory blocks.

Bitmapped memory manager

An interesting refinement over the original fixed-size memory allocation scheme is the bitmapped memory manager. In this scheme, memory is requested from the operating system in relatively big chunks and memory allocation is achieved by carving out from these chunks. Deallocation is done by freeing the entire block at a single go, thus sealing off any potential leaks. Yet another advantage of this approach is that it can support array versions of the `new` and `delete` operations.

In this approach, a large chunk of memory is requested by the memory manager. This chunk is further subdivided into smaller fixed-size blocks. In this case, the size of each of these blocks is equal to the size of the `Complex` object. Based on the number of blocks in the chunk, a bitmap is maintained separately to indicate the status (free or allocated) of each block, with the number of bits equal to the number of blocks. When the program requests a new `Complex` object, the memory manager consults the bitmap to obtain a free block. All free blocks have their corresponding bit set to 1. Occupied blocks have their bits reset to 0. To provide for the `new[ ]` and `delete[ ]` operators, you need to maintain an auxiliary data structure to help keep track of how many bits in the bitmap need to set or reset when a `Complex` array object is created or destroyed.

Creating a bitmapped memory manager

The memory manager requests a sufficiently large chunk of memory from the operating system. Future allocations are carved out from this block. If this request fails, then the memory manager bails out and passes an appropriate message to the user signaling its exit. All entries in the bitmap at this stage are set to 1.

If the memory manager runs out of free blocks, it makes further requests to the operating system for large memory chunks. Every bitmap in the `MemoryManager` data structure is meant to augment the corresponding memory chunk. However, when a delete is called, the corresponding block is made available to the user. Thus non-sequential delete calls create what is known as *fragmented memory* and blocks of suitable size can be provided from this memory.

The basic structure of the `MemoryManager` class is given in Listing 10. It contains the `MemoryPoolList`, which holds the start addresses of memory chunks requested from the operating system. For each chunk, there is an entry in the `BitMapEntryList`. `FreeMapEntries` is a data structure to facilitate quick searching of next available free blocks for non-array versions of `new` calls.

Listing 10. MemoryManager class definition

```
class MemoryManager : public IMemoryManager
  {
    std::vector<void*> MemoryPoolList;
    std::vector<BitMapEntry> BitMapEntryList;
    //the above two lists will maintain one-to-one correspondence and hence
    //should be of same  size.
    std::set<BitMapEntry*> FreeMapEntries;
    std::map<void*, ArrayMemoryInfo> ArrayMemoryList;

  private:
    void* AllocateArrayMemory(size_t size);
    void* AllocateChunkAndInitBitMap();
    void SetBlockBit(void* object, bool flag);
    void SetMultipleBlockBits(ArrayMemoryInfo* info, bool flag);
  public:
    MemoryManager( ) {}
    ~MemoryManager( ) {}
    void* allocate(size_t);
    void   free(void*);
    std::vector<void*> GetMemoryPoolList();
  };
```

The `ArrayMemoryList` holds information about memory allocated for arrays of `Complex` objects. It is basically a mapping of the start address of an array to a structure maintaining the `MemPoolListIndex`, the `StartPosition` of the array in the bitmap, and the size of the array, as shown in Listing 11.

Listing 11. ArrayInfo structure definition

```
typedef struct ArrayInfo
  {
  int   MemPoolListIndex;
  int   StartPosition;
  int   Size;
  }
ArrayMemoryInfo;
```

To facilitate easy manipulation of the bitmap, it is maintained as a `BitMapEntry` object that stores some extra metadata information, as shown in Listing 12. Provisions for setting single or multiple bits in the bitmap are provided via the `SetBit` and `SetMultipleBits` application program interfaces (APIs). The `FirstFreeBlock()` API retrieves the first free block pointed by the bitmap.

Listing 12. BitMapEntry class definition

```
typedef struct BitMapEntry
  {
  int      Index;
  int      BlocksAvailable;
  int      BitMap[BIT_MAP_SIZE];
  public:
    BitMapEntry():BlocksAvailable(BIT_MAP_SIZE)
      {
      memset(BitMap,0xff,BIT_MAP_SIZE / sizeof(char));
      // initially all blocks are free and bit value 1 in the map denotes
      // available block
      }
    void SetBit(int position, bool flag);
    void SetMultipleBits(int position, bool flag, int count);
    void SetRangeOfInt(int* element, int msb, int lsb, bool flag);
    Complex* FirstFreeBlock(size_t size);
    Complex* ComplexObjectAddress(int pos);
    void* Head();
  }
BitMapEntry;
```

The memory manager initializes an 'n' bit array (which acts as the bitmap) such that each bit of this array corresponds to a block of the requested memory. All entries are set to 1. This is done in the constructor for the `BitMapEntry`.

On a call to the non-arrayed version of `new` or `malloc` for the `Complex` class, the memory manager checks whether a block is available in the `FreeMapEntries` data structure. If any such block is found there it is returned, otherwise a new chunk of memory is requested from the operating system and its corresponding `BitMapEntry` is set. The block of memory to be returned to the user is carved out from this chunk, its availability bit is set to 0 to indicate it is no longer free, and the pointer is returned to the user as evident from the relevant code shown in Listing 13.

Listing 13. MemoryManager::allocate definition

```
void* MemoryManager::allocate(size_t size)
  {
  if(size == sizeof(Complex))    // mon-array version
    {
    std::set<BitMapEntry*>::iterator freeMapI = FreeMapEntries.begin();
    if(freeMapI != FreeMapEntries.end())
      {
      BitMapEntry* mapEntry = *freeMapI;
      return mapEntry->FirstFreeBlock(size);
      }
    else
      {
      AllocateChunkAndInitBitMap();
      FreeMapEntries.insert(&(BitMapEntryList[BitMapEntryList.size() - 1]));
      return BitMapEntryList[BitMapEntryList.size() - 1].FirstFreeBlock(size);
      }
    }
  else  // array version
    {
    if(ArrayMemoryList.empty())
      {
      return AllocateArrayMemory(size);
      }
    else
      {
      std::map<void*, ArrayMemoryInfo>::iterator infoI =ArrayMemoryList.begin();
      std::map<void*, ArrayMemoryInfo>::iterator infoEndI =
        ArrayMemoryList.end();
      while(infoI != infoEndI)
        {
        ArrayMemoryInfo info = (*infoI).second;
        if(info.StartPosition != 0)  // search only in those mem blocks
          continue;             // where allocation is done from first byte
        else
          {
          BitMapEntry* entry = &BitMapEntryList[info.MemPoolListIndex];
          if(entry->BlocksAvailable < (size / sizeof(Complex)))
            return AllocateArrayMemory(size);
          else
            {
            info.StartPosition = BIT_MAP_SIZE - entry->BlocksAvailable;
            info.Size = size / sizeof(Complex);
            Complex* baseAddress = static_cast<Complex*>(
              MemoryPoolList[info.MemPoolListIndex]) + info.StartPosition;

            ArrayMemoryList[baseAddress] = info;
            SetMultipleBlockBits(&info, false);

            return baseAddress;
            }
          }
        }
      }
    }
  }
```

Code to set or reset the bit for a single block is in Listing 14.

Listing 14. MemoryManager::SetBlockBit definition

```
void MemoryManager::SetBlockBit(void* object, bool flag)
  {
  int i = BitMapEntryList.size() - 1;
  for (; i >= 0 ; i--)
    {
    BitMapEntry* bitMap = &BitMapEntryList[i];
    if((bitMap->Head <= object )&amp;&amp;
      (&(static_cast<Complex*>(bitMap->Head))[BIT_MAP_SIZE-1] >= object))
      {
      int position = static_cast<Complex*>(object) -
      static_cast<Complex*>(bitMap->Head);
      bitMap->SetBit(position, flag);
      flag ? bitMap->BlocksAvailable++ : bitMap->BlocksAvailable--;
      }
    }
  }
```

Code to set or reset the bits for multiple blocks is in Listing 15.

Listing 15. MemoryManager::SetMultipleBlockBits definition

```
void MemoryManager::SetMultipleBlockBits(ArrayMemoryInfo* info, bool flag)
  {
  BitMapEntry* mapEntry = &BitMapEntryList[info->MemPoolListIndex];
  mapEntry->SetMultipleBits(info->StartPosition, flag, info->Size);
  }
```

For the array version, the handling is a bit different. Memory blocks for arrays are allocated from different memory chunks than those used for single-object memory allocations. This reduces the overhead of searching for the next free available memory in both types of calls to `new`. The memory manager looks up the `ArrayMemoryList` data structure to obtain information regarding a chunk that provides memories for arrays. When found, it retrieves a suitable subsection in this chunk and provides its address to the user and the representing bits are set to 0. If for any reason there is a failure, a new chunk is requested from the operating system and the memory is carved out. Memory chunks for both types of new calls are stored as part of `MemPoolList` in the `MemoryManager` class.

On a call to the non-arrayed version of `delete` or `free` for a pointer to an object of type `Complex`, the memory chunk containing this pointer is determined first (see Listing 16. The `BitMapEntry` for this chunk is then consulted to set the corresponding bit back to 1, thereby marking it as available. The whole operation takes O(1) time. For `delete []`, the associated information is retrieved from `ArrayMemoryList`, and when the start position and the size of the bitmap is known, those many bits are marked available.

Listing 16. MemoryManager::free definition

```
void MemoryManager::free(void* object)
  {
  if(ArrayMemoryList.find(object) == ArrayMemoryList.end())
    SetBlockBit(object, true);         // simple block deletion
  else
    {//memory block deletion
    ArrayMemoryInfo *info = &ArrayMemoryList[object];
    SetMultipleBlockBits(info, true);
    }
  }
```

The full listing of the bitmapped memory manager code is in the Downloadable resources section. It is a working example of the memory manager discussed in this section, and it includes all the listings we discussed in this section.

Problem with inheritance

The size of a derived class is quite likely to be different from that of its base class. For example, consider a derived variant of the `Complex` class, called `ComplexT`, meant for tracking the value of a complex number versus time, which contains an additional `unsigned long` to store time. The `new` or `delete` operator that was overridden earlier is not going to work now, unless we define them again specifically for the derived variant as well. There are three possible solutions to this problem:

- Let the `MemoryManager` maintain two different pools for these two different classes. This, in effect, means two variants of the allocation/deallocation routines and two pointers to store the two pool heads. This works, but the scalability is limited.
- Maintain a single pool based on the size of the largest derived class. The `allocate` routine always returns the size of the largest derived class, irrespective of which object in the base-derived hierarchy is requesting memory. This works fine but is not a very useful approach because the memory footprint of the program would increase.
- Create a general-purpose memory manager for variable-size allocation.

Free-lists based memory manager

In a typical program, the majority of the requested memory blocks are of certain specific sizes. This memory manager takes advantage of this heuristic. To implement it, you maintain lists that contain addresses of the free blocks of those sizes. These lists are known as *free-lists* in technical jargon. For example, consider a program where memory requests are made for sizes of 8, 16, 24, 32, 48, and 64 bytes. For such purposes, the memory manager contains six different free-lists and requests for a specific memory size are provided from the corresponding list. Similar to the bitmapped memory manager, memory is requested from the operating system on startup, and the memory manager internally partitions this requested block from the operating system into lists. Whenever a user requests memory for an object of, say, 'm' bytes, the size of object is converted to the nearest block size 'n', for which there is a free-list that stores blocks of size 'n'.

A brief introduction of guard bytes

The 'n' byte block not only stores the object but also stores some metadata information. Each block contains four extra bytes towards its end that contain what are known as *guard bytes* in the heap memory operations paradigm. Typically, `memcpy` and `memset` functions are capable of intruding into bytes of memory that are logically out of bounds for the allocated memory. This causes heap memory corruption. Many compilers add special characters around an allocated `mem` block that act as sentinels (or guard bytes) for the block. Any accesses to such blocks of allocated memory check for these special characters around the allocated block. If such characters are not found, then it is assumed that somehow their value was changed during program execution in an intrusive way, which implies that the heap memory is no longer in an ordered state and is, hence, corrupted. It is a good mechanism for catching some of those nasty memory bugs that programmers are aware of. In the following example, two out of four bytes act as the guard bytes, the next byte stores the size of this object, and the last byte stores a flag indicating whether this object is available or already occupied.

Creating a free-lists memory manager

The memory manager maintains lists of pointers to blocks of variable sizes, as described earlier. It also maintains the chunks of memory requested from the operating system as a `memoryPool`. This pool can be used to free the entire chunks of memory when `MemoryManager` invokes its destructor. Listing 17 shows the data structure.

Listing 17. MemoryManager class definition for free-lists based implementation

```
class MemoryManager : public IMemoryManager
  {
  private:
    VoidPtrList     Byte8PtrList;
    VoidPtrList     Byte16PtrList;
    VoidPtrList     Byte24PtrList;
    VoidPtrList     Byte32PtrList;
    VoidPtrList     Byte40PtrList;
    std::vector<void*>    MemoryPoolList;

    . . . . . . . . . //helper routines may go in here

  public:
    MemoryManager( ) {}
    ~MemoryManager( ) {}
    void* allocate(size_t);
    void   free(void*);
  };
```

Our example uses three classes of 16, 28, and 32 bytes and, thus, requires blocks of 24, 32, and 40 bytes to hold them with guard bytes. Hence, `Byte24PtrList`, `Byte32PtrList`, and `Byte40PtrList` are the ones used most frequently, and the code is centered on them. However, the design is flexible enough to also add lists of other sizes that you desire.

Operators `new` and `delete` are overloaded for these classes, which delegate the calls to `allocate` and `free` routines that take care of memory allocation and deallocation. These routines are discussed in detail here. We work with a chunk size of 1024 as the initial count of objects. Also, the size of each class used in the example is stored as a predefined constant, as shown in Listing 18.

Listing 18. Predefined constants used in this implementation

```
  const int JOB_SCHEDULER_SIZE = sizeof(JobScheduler);
  const int COMPLEX_SIZE = sizeof(Complex);
  const int COORDINATE_SIZE = sizeof(Coordinate);
  const int POOL_SIZE = 1024; //number of elements in a single pool
            //can be chosen based on application requirements

  const int MAX_BLOCK_SIZE = 36; //depending on the application it may change
                //In above case it came as 36
```

These constants are used in the `allocate` and `free` routines.

Allocate routine

For a given size of the block, the `allocate` routine determines which list to work on in the memory manager. It consults the requisite list to see if any blocks are available. Note that this list stores only the pointer to the block and not the block itself (the block is part of the `MemoryPoolList`).

If the free list is empty, an additional chunk of memory is requested from the operating system and organized as partitioned blocks by the `InitialiseByte24List`, `InitialiseByte32List` and `InitialiseByte40List` routines.

When a free block address is available, the corresponding block is marked as unavailable, its guard bytes are set, and the address of the block is returned. Consider the implementation as shown in Listing 19.

Listing 19. MemoryManager::allocate definition

```
void* MemoryManager::allocate(size_t size)
  {
  void* base = 0;
  switch(size)
    {
    case JOB_SCHEDULER_SIZE :
      {
      if(Byte32PtrList.empty())
        {
        base = new char [32 * POOL_SIZE];
        MemoryPoolList.push_back(base);
        InitialiseByte32List(base);
        }
      void* blockPtr =  Byte32PtrList.front();
      *((static_cast<char*>(blockPtr)) + 30) = 32; //size of block set
      *((static_cast<char*>(blockPtr)) + 31) = 0; //block is no longer free
      Byte32PtrList.pop_front();
      return blockPtr;
      }
    case COORDINATE_SIZE :
      {
      if(Byte40PtrList.empty())
        {
        base = new char [40 * POOL_SIZE];
        MemoryPoolList.push_back(base);
        InitialiseByte40List(base);
        }
      void* blockPtr =  Byte40PtrList.front();
      *((static_cast<char*>(blockPtr)) + 38) = 40; //size of block set
      *((static_cast<char*>(blockPtr)) + 39) = 0; //block is no longer free
      Byte40PtrList.pop_front();
      return blockPtr;
      }
    case COMPLEX_SIZE :
      {
      if(Byte24PtrList.empty())
        {
        base = new char [24 * POOL_SIZE];
        MemoryPoolList.push_back(base);
        InitialiseByte24List(base);
        }
      void* blockPtr =  Byte24PtrList.front();
      *((static_cast<char*>(blockPtr)) + 22) = 24; //size of block set
      *((static_cast<char*>(blockPtr)) + 23) = 0; //block is no longer free
      Byte24PtrList.pop_front();
      return blockPtr;
      }
    default : break;
    }
  return 0;
  }
```

Free routine

The `free` routine takes the address of the block and searches for the byte containing size information (in the guard bytes) that resides at the end of the block. It's the second-to-last byte in the block. After the size is obtained, the object is marked as available (last byte set to 1) and the corresponding list is fetched. The object is then pushed into this *free* list marking the end of the deletion phase. Consider the implementation in Listing 20.

Listing 20. MemoryManager::free definition

```
void MemoryManager::free(void* object)
  {
  char* init = static_cast<char*>(object);

  while(1)
    {
    int count = 0;
    while(*init != static_cast<char>(0xde))
          //this loop shall never iterate more than
      {                  // MAX_BLOCK_SIZE times and hence is O(1)
      init++;
      if(count > MAX_BLOCK_SIZE)
        {
        printf ("runtime heap memory corruption near %d", object);
        exit(1);
        }
      count++;
      }
    if(*(++init) == static_cast<char>(0xad))  // we have hit the guard bytes
      break;  // from the outer while
    }
  init++;
  int blockSize = static_cast<int>(*init);
  switch(blockSize)
    {
    case 24: Byte24PtrList.push_front(object); break;
    case 32: Byte32PtrList.push_front(object); break;
    case 40: Byte40PtrList.push_front(object); break;
    default: break;
    }
  init++;
  *init = 1; // set free/available byte
  }
```

The full listing of the free-lists memory manager code is in the Downloadable resources section. It is a working example of a free-lists memory manager implementation, and it includes all the listings we discussed in this section.

Advantages and disadvantages

There are several advantages to this design:

- Variable size memory allocation is handled gracefully.
- The design is flexible and can be extended by determining the size of lists per the software requirements.
- The ability to add guard bytes allows a lot of metadata information that can be stored for an object. We already used it to determine runtime heap memory corruption.

While there are gains in performance, the disadvantage of this design is that it is memory intensive.

Multithreaded memory manager

So far, the memory managers we've created don't take into account a concurrent environment. In a multithreaded environment, memory allocation and deallocation could potentially be attempted by multiple threads at the same time. This means we must ensure that the allocation and free operations in our memory manager are atomic. That is, we must provide for a mechanism that guarantees mutual exclusion among two threads when they attempt these operations simultaneously. The standard way to ensure that the allocation and free methods are atomic is to put a lock-based mechanism in place. Whenever a thread calls either of these two methods, it must get ownership to a single exclusive lock before it can actually get access to or free memory. If the lock is owned by another thread, then the current thread is blocked until it owns this lock. Listing 21 denotes the signatures of functions used as locking mechanisms.

Listing 21. pthread mutex methods

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

For our purposes, we use pthread mutexes defined in the standard header, `pthread.h`, that comes with the GNU compiler collection installation to simulate the locking or unlocking mechanism. The code in the `allocation` and `free` routines is put between the `pthread_mutex_lock` and `pthread_mutex_unlock` methods. If a second thread calls either of these routines when a thread is already accessing the memory pool, it is made to wait till the lock is available. Listing 22 is a modified snippet of the methods.

Listing 22. Concurrent versions of the allocation and free methods

```
void* MemoryManager::allocate(size_t size)
  {
  pthread_mutex_lock (&lock);
  ... // usual memory allocation code
  pthread_mutex_unlock (&lock);
  }

void* MemoryManager::free(size_t size)
  {
  pthread_mutex_lock (&lock);
  ... // usual memory deallocation code
  pthread_mutex_unlock (&lock);
  }
```

The memory manager class now needs to include a `pthread_mutex_lock` object. The class constructor needs to be adequately modified to call `pthread_mutex_init` to initialize the lock, and the class destructor must call `pthread_mutex_destroy` to destroy the mutex object. Listing 23 is the revised code for the `MemoryManager` class.

Listing 23. MemoryManager class modified to handle threaded allocation and deallocation

```
class MemoryManager : public IMemoryManager
  {
  pthread_mutex_t lock;
  ... // usual MemoryManager code follows
  };

MemoryManager::MemoryManager ()
  {
  pthread_mutex_init (&lock, NULL);
  ... // usual constructor code
  }

MemoryManager:: ~MemoryManager ()
  {
  ... // usual destructor code
  pthread_mutex_destroy (&lock);
  }
```

Now run the multithreaded version of the memory manager. In our tests, it showed that it is a lot slower than the single-threaded version, which demonstrates why it often pays to have a specialized custom-coded memory manager.

Summary

This tutorial explained the following concepts:

- The need to have a memory manager in user code.
- The design requirements for a memory manager.

- How to create a fixed-size allocator or memory manager.
- How to create a variable-sized allocator.
- Memory allocation in a multi-threaded environment.

For more information about this subject, see the Related topics section.

Downloadable resources

Description	Name	Size
Source files for bitmapped memory manager	bitmapped_memory_manager.zip	4KB
Source files for free-lists based memory manager	free_list_mem_mgr.zip	3KB

Related topics

- The Memory Management Reference: This Ravenbrook Web site is a good place to get an introduction to memory management
- "A memory allocator" (Doug Lea): Discusses techniques and algorithms.
- "Inside memory management" (developerWorks, November 2004): This article provides an overview of the memory management techniques that are available to Linux programmers.
- IBM trial software: Build your next development project with software for download directly from developerWorks.