Memory Management

1 Introduction

There are two types of memory a programmer has access to, Heap and Stack. The management of stack memory is handled by function calls before and after they execute, this means that after a function exits the memory associated with it is lost. Stack memory includes variables declared in your functions, global variables and even the parameters to your functions.

If we need an object to stay around after a function call, or if the data needs to dynamically change size we may need to use heap memory. The heap is a linear piece of memory which can be used a reused for any purpose the programmer sees fit, but this region needs to be organized so that the chunks of data can be kept track of. This allows for the program to use the space efficiently reducing fragmentation in memory.

Normally all of this process is hidden from the programmer through the use of new and delete. Back in the 'C' days memory was reserved by using the malloc function, and returned to the heap using the free function. These functions however did not modify the data underneath, i.e. they do not do any constructing or destructing. The new and delete operators will auto matically call constructors and destructor for you.

2 Rewriting Malloc

If we wish to control memory allocation for ourselves we need to write our own malloc and free. We may in fact use the original malloc to reserve space to manage ourselves. Then if we rewrite new and delete our new allocator will integrate seamlessly into our code.

2.1 Operator Overloading

The operators 'new' and 'delete' can be overloaded just like most other operators. If we were to re-implement the default new and delete it would look like:

```
1  void* operator new(size_t size) {
2     return malloc(size);
3  }
4
5  void operator delete(void* pointer) {
6     free(pointer);
7  }
```

Operating Overloading

We also need to override the array versions of these functions. The array versions do not need any special treatment:

```
8  void* operator new[](size_t size) {
9     return malloc(size);
10 }
11
12 void operator delete[](void* pointer) {
13    free(pointer);
14 }
```

Operating Overloading

We can even add these as member functions of a class such that it will only apply to that class and its subclasses. This can be usefull for allowing just your own classes to use the modified 'new' and 'delete'.

Extra parameters can be added to new and delete. This can be used in a couple of ways, one would be to manually choose which 'new's go to which allocators like with the case for classes, but with finner control. Another example is the following:

```
15 void* operator new(size_t size, void* location) {
16    return location;
17 }
```

Operating Overloading

With this version of new we can call the constructor on any location in memory we choose even on the stack. And example invocations would be:

```
18 char *text = new char[sizeof(Dog)];
19 new((void*)text) Dog("Fido");
20
21 Cat cat("Fluffy");
22 new((void*)&cat) Cat("Bob");
```

Operating Overloading

In this example the cat fluffy has be replaced with the cat called bob, but in this example fluffy did not have its deconstructor called on it. This could cause memory leaks if the Cat class keeps data on the heap, this can be fixed by explicitly calling the destructor on the object fluffy:

```
23 Cat cat("Fluffy");
24 cat.~Cat();
25 new((void*)&cat) Cat("Bob");
```

Operating Overloading

That last code example is valid but can easily be misunderstood, (so please do as I say, not as I do).

The delete operator can also be overloaded with extra parameters, but there is no way to call it except for writing:

```
26 operator delete(pointer, extra_parameter);
```

Operating Overloading

even though it cannot be called easily it needs to be implemented for when a class throws an exception inside its constructor. In this case it will automatically call the corresponding delete operator.

2.2 Bin Algorithm

The bin algorithm splits large regions of memory 'bins' into equally sized chunks which can be used by the program. Its a simple algorithm but is useful for applications which regularly use the same sized chunks again and again.

2.3 The Code

We will now look at the code for an allocator. First we will some constants which will make it easier to tweak the algorithm later.

```
1 #include <stdlib.h>
2 #include <iostream>
3
4 #define MEMORY_SIZE (10*1024*1024)
5 #define BIN_SIZE (1024)
6 #define NUM_BINS (128)
7 #define BIN_SIZE_INC (16)
```

Allocator.cpp

To encapsulate all the code for the allocator we will define it as a class. Our new malloc and free will be implemented as methods of this class. Also defining the new operator as a member will allow us to control the allocators own allocation of memory. This will rule out any circular dependence and therefore the allocator can be created before it is use. Another advantage of using a class is that we can use inheritance and polymorphism like any other class (this example is too small to take advantage of this).

```
8  class BinAlloc {
9  public:
10    void* malloc(size_t size);
11    void free(void* location);
12
13    static BinAlloc* const allocator;
14    void* operator new(size_t size) {
15       return ::malloc(MEMORY_SIZE);
16    }
```

Allocator.cpp

In order to stop multiple instances of this class being made the constructors of this class are made private and a single static const pointer is made for the single instance. The copy constructor and assignment operator are also ways in which the class could be copied so they are also hidden. This is called the singleton pattern.

```
17 private:
18    BinAlloc(): max(0) {
19       for (int i = 0; i < NUM_BINS; i++)
20          bins[i] = NULL;
21    }
22    BinAlloc(const BinAlloc& other) {}
23    BinAlloc operator=(const BinAlloc& other) {
24       return *this;
25    }
```

Allocator.cpp

The following methods are convenience methods for calculating useful information. For example the next compatible size for a memory allocation, or the start of memory for the n'th chunk etc. there is also a 'make_chunk' method that will be used to split up new chunks of memory for use by the allocator. The variables bins is an array of pointers to the beginning of 'free lists' which store the memory address of free chunks (using one memory location to store the location of the next, NULL indicating the end of the list). There is also the variable max which indicates which chunk of memory can be split up next.

```
26    void* chunk(int num);
27    int chunk_index(void* loc);
28    int bin_for_size(size_t size);
29    size_t size_for_bin(int bin);
30    int bin_for_location(void* location);
31    void make_chunk(size_t size);
32
33    int max;
34    void* bins[NUM_BINS];
35 };
```

Allocator.cpp

Now we overide the global new and delete operators to use the allocator class.

```
36 BinAlloc* const BinAlloc::allocator = new BinAlloc();
37
38 void* operator new(size_t size) {
39    return BinAlloc::allocator->malloc(size);
40 }
41
42 void* operator new[](size_t size) {
43    return BinAlloc::allocator->malloc(size);
44 }
45
46 void operator delete(void* location) {
47    BinAlloc::allocator->free(location);
48 }
49
50 void operator delete[](void* location) {
51    BinAlloc::allocator->free(location);
52 }
```

Allocator.cpp

Next are the convenience methods. In particular in the 'bin_for_size' method we subtract one from the size so that the integer arithmetic will round down for perfect multiples of 16. This way chunks that exactly fit won't be placed in the next size up. The rest of the methods can be thought of as rearranging the equation:

$$chunk = num * BINSIZE + sizeof(BinAlloc) + (char*)this \tag{1}$$

```
53 void* BinAlloc::chunk(int num) {
54    return num*BIN_SIZE + sizeof(BinAlloc) + (char*)this;
55 }
56
57 int BinAlloc::chunk_index(void* location) {
58    return (int)((char*)location - sizeof(BinAlloc)
59       - (char*)this)/BIN_SIZE;
60 }
61
62 int BinAlloc::bin_for_size(size_t size) {
63    return (size - 1)/BIN_SIZE_INC;
64 }
65
66 size_t BinAlloc::size_for_bin(int bin) {
67    return (bin + 1)*BIN_SIZE_INC;
68 }
69
70 int BinAlloc::bin_for_location(void* location) {
71    int index = chunk_index(location);
72    void* chunk_start = chunk(index);
73    return *(int*)chunk_start;
74 }
```

Allocator.cpp

The main body of the algorithm is in the next three methods. 'make_chunk' splits then next chunk of memory into blocks of memory at the next bin increment, it then calls free on these chunks to add them to the bin. We do not allocate the very first piece of memory from these chunks and reserve it to store a size value. 'malloc' will check whether there is anything in the appropriate bin and create a new chunk if the bin is empty. Then it will pop an element off the front of the list. 'free' works by pushing items on the list, but it needs to fin the corresponding bin. This is where the 'bin_for_location' method is useful. Rounding down to the beginning of the bin we can read the size value we wrote when creating the bin.

```
75  void BinAlloc::make_chunk(size_t size) {
76     if ((max + 2)*BIN_SIZE + sizeof(BinAlloc) > MEMORY_SIZE) return;
77     int bin = bin_for_size(size);
78     size_t actual_size = size_for_bin(bin);
79
80     char* chunk_ptr = (char*)chunk(max++);
81     *(int*)chunk_ptr = bin;
82     chunk_ptr += actual_size;
83
84     while (chunk_ptr + actual_size <= chunk(max)) {
85        free(chunk_ptr);
86        chunk_ptr += actual_size;
87     }
88  }
89
90  void* BinAlloc::malloc(size_t size) {
91     int bin = bin_for_size(size);
92     if (bins[bin] == NULL)
93        make_chunk(size);
94     if (bins[bin] == NULL) return NULL;
95     void* result = bins[bin];
96     bins[bin] = *(void**)result;
97     return result;
98  }
99
100 void BinAlloc::free(void* location) {
101    int bin = bin_for_location(location);
102    *(void**)location = bins[bin];
103    bins[bin] = location;
104 }
```

Allocator.cpp

Now all thats left to do is to test whether it works. You may notice something interesting about the address of the variables a, b and c. Also this allocator is not complete, it will not allocate chunks of arbitrary size, and once a region of memory stores a certain size it is fixed only being able to store data at that size.

```
105 using namespace std;
106
107 void main() {
108
109    int* a = new int(3);
110    int* b = new int(4);
111    int* c = new int(5);
112
113    int* arr =  new int[20];
114    for (int i = 0; i < 20; i++)
115       arr[i] = i*i;
116
117    cout << "a = " << a << ", *a = " << *a << endl;
118    cout << "b = " << b << ", *b = " << *b << endl;
119    cout << "c = " << c << ", *c = " << *c << endl;
120    cout << "arr = " << arr << ", arr[9] = " << arr[9] << endl;
121
122    delete[] arr;
```

Allocator.cpp

```
123     arr =  new int[20];
124     for (int i = 0; i < 20; i++)
125        arr[i] = 1 << i;
126 
127     cout << "arr = " << arr << ", arr[9] = " << arr[9] << endl;
128 
129     delete a;
130     delete b;
131     delete c;
132 
133     system("Pause");
134 }
```

Allocator.cpp