The Heap

Introduction: Why do we need a "heap" anyway?
The heap is used to allocate and manage memory. Ordinarily, we can "allocate" the memory that we need simply by defining variables. Each variable we define instructs the compiler (with a little help from the linker) to allocate enough memory locations to store one value. This process works great most of the time. Even when we need lots of memory, we don't ordinarily need to resort to using the heap – we can define an array of variables.

There are two potential problems with ordinary variables and arrays. The first problem is that the compiler must be able to determine the amount of memory necessary to store the variable. In the case of a single variable, this is never an issue. Every **int** takes four bytes, every **double** requires eight bytes, etc. However, in the case of an array of variables, it may be inconvenient to specify the exact size at compile time. Consider the following (slightly contrived) example:

```
void reverse(char p[]) {
  /* step 1: count the characters in the string */
  int length = 0;
  while (p[length] != 0) {
    length += 1;
  }

  /* step 2: store the characters in a temporary array */
  char temp[length]; // an array to temporarily store the characters

  int k = 0;
  while(k < length) {
    temp[k] = p[k];
    k += 1;
  }

  /* step 3: write the characters back into the original array,
   * but in reverse order */
  int k = 0;
  while (k < length) {
    p[k] = temp[length-k-1];
    k += 1;
  }
}
```

There are other ways to write the *reverse* function that do not require a temporary array, but let's assume that this was the algorithm that makes the most sense to us, and hence that's the algorithm we should use. The problem with the program is that the size of the

temp array is not known until the function begins to run. The compiler cannot know what the value of *length* will be. Hence, the compiler can't know how big the array *temp* will be. That's a real nuisance, since the size of the array *temp* may be critical for determining such things as the size of the activation record, or the address of other local variables (like *k* which was declared after *temp*), etc. Because the compiler cannot do its normal activities without knowing precisely the size of all arrays, the C (and C++) programming languages require that the size of an array must be a "compile time constant". Typically, that means that the size of the array must be a "literal" (e.g., a constant like "10" or "1000"). Since the compiler will not allow us to declare an array that has size *length*, our only resort is to declare a pointer variable and to initialize that pointer to hold the address of some *dynamically allocated storage* – i.e., to set the pointer variable so that it points to memory allocated by the heap.

The second problem that comes up that requires dynamically allocated space is the need to find storage for information "created" by a function. There are several examples of how a function can be used to "create" information. A simple example would be for the function to create an array of all the prime numbers smaller than *N*. When I call the function, I may not know how many prime numbers there will be. Hence, I can't declare an array the right size to hold the answer to my own question. The function that answers my question will need to dynamically allocate the array and store the prime numbers. Then, that function can return a pointer to the beginning of the array. Notice since I'll be looking at the array after the function is over, the array can't be part of the activation record for the function – i.e., the array holding the prime numbers can't be a local array to the function that creates it. If the array were local to the function, then those memory locations might be overwritten with some other activation record(s) if I were to call some other function(s). The only solution is for the function to allocate its array by using the heap. Then the prime numbers can be written into the array and safely accessed even after the function has returned.

How do we use the heap?
Keep in mind that we use the heap when we need storage – when we need memory to store some information and it is inconvenient, incorrect or impossible to declare new variables to hold this information. Given that insight, it should be obvious that we'll be accessing this storage using pointers. What we'll do is declare a pointer variable and initialize that variable so that it holds the address of our new storage.

```
void ourFunction(void) {
  int* p; // our pointer variable
  p = … ; // initialize p to hold the address of our new storage
```

When we want to store our information or access it, then we'll use the pointer dereference syntax (e.g., `*p`).

The C standard library includes two functions for helping us with the heap. The first function is called *malloc* which stands for **m**emory **alloc**ate. This function allocates the required amount of storage and returns the address of the first byte of the allocated space.

In our example, we would take the value returned by *malloc* and store that value in our pointer variable. Note that *malloc* has one parameter. The value passed to *malloc* in this parameter is the amount of storage we are requesting (in bytes). For example, if we want to store just one **int**, then we can call *malloc* with the argument 4 (since an **int** is four bytes).

```
void ourFunction(void) {
  int* p; // our pointer variable
  p = malloc(4); // initialize p to hold the address of our new storage
  *p = 42; // store our information in the dynamically allocated memory
```

A lot of small or simple programs will only need to call *malloc*. They may call *malloc* many times – each time allocating additional storage. Since our computers only have a finite amount of memory, we can't keep doing this forever. Sooner or later we either have to deallocate (un-malloc?) the storage or else we'll run out of memory.

Running out of memory

If you ever do run out of memory, then *malloc* will return the address 0. Recall that 0 is a special address in C, no information can be stored at address 0. So, technically, you should probably check to make sure that *malloc* returns a real address (anything other than zero) before you try to store something there. In my opinion, it's somewhat questionable whether you should check or not. After all, if you have run out of memory, what are you going to do about it? For most programs, the best you could do at this point is print out some (fairly useless) error message like "Sorry bloke, but I just ran out of memory, go buy a bigger computer why don't you, and in the meantime, I'm just going to shut down." OK, that's an especially annoying error message, but in all seriousness, what are you going to do about it if you've run out of memory? On the other hand, think about what happens if we didn't check to see if we'd run out of memory. In that case, assuming we ever do really run out of memory, *malloc* will return the address 0. Our program, since it didn't check, will go ahead and try to store information at address 0. The operating system will then get very pissed off at our program and kill it. Instead of our program printing a cheeky error message, the operating system will display something like "Unhandled Exception" or "Segmentation Violation" or something fairly cryptic. The bottom line still is, we get a useless error message and our program shuts down.

```
void ourFunction(void) {
  int* p;
  p = malloc(4);
  if (p == 0) { // malloc returns zero if we're out of memory
    printf("out of memory");
    exit(-1); // causes our program to immediately shut down
  }

  /* if we reach this point, we know for certain that malloc returned OK */
  *p = 42; // store our information in the dynamically allocated memory
```

In our programs, we won't worry about running out of memory. Our programs are pretty small, and our computers have lots of memory. In real professional software, we'd have to be concerned about this possibility, but in that case, we should be using a language like C++ or Java that has proper "Exception Handling" support anyway. If it were my program, I still wouldn't be checking the return value from *malloc*.

Deallocating Memory

In most programs, when we allocate storage, we only need to use that storage for a short period of time. For example, in our original *reverse* function, we needed storage to hold a string. Once we'd finished reversing the string, we no longer needed the storage. Hence, if we could somehow tell *malloc* that this memory was no longer being used to hold our string, then *malloc* could maybe use that memory for something else. The way we "deallocate" memory is by calling a function named *free*. The *free* function doesn't (normally) do anything to the memory, and there is no return value. What *free* does do is to indicate somewhere internally in the computer that we're no longer using the memory that was allocated to us (and hence that memory is available for other purposes). The argument to *free* is the address of the first byte in the region of memory that we are deallocating. **NOTE: the address passed to *free* must be EXACTLY the same address as that returned by *malloc*.** If we call *malloc* many times (to allocate storage for different things), then we will need to call *free* several times. You will usually want to ensure there's a one-to-one correspondence between calls to *malloc* and calls to *free*. Here's how it goes: you call *malloc* and take the address returned and store that address in a pointer variable. Then, you use your pointer variable to store and access your information for as long as you need to. Once you're done with your storage, you call *free* and use the value of your pointer variable as the argument to *free*. From this point onward, you should not access the memory any more (i.e., you should not dereference the pointer).

```
void ourFunction(void) {
  int* p;
  p = malloc(4);
  *p = 42; // store our information in the dynamically allocated memory
  *p = *p + 1; // use our information for a while
  …
  /* now we're done with the information */
  free(p);  // tell the heap that we're done
  p = 0; // optional: forget the address where the memory was
```

That's about all there is to it. There are a few rules that you are expected to follow when using *malloc* and *free*.

1. The argument to *malloc* is the maximum number of bytes you will require to store your information. If you exceed this amount of memory, then your program contains a **buffer-overflow error**.

2. When you have finished using the memory, you should call *free* using the same address as was returned by *malloc*. If you fail to call *free* for each call to *malloc* then your program contains a **memory-leak error**.
3. Once you call *free* you should not access the memory any longer. If you access memory that has been *free*'d, then your program has a **use-after-free error** (this is the noun form of "use", the one pronounced "yoos").

One special form of the use-after-free error is to invoke the *free* function twice using the same address. This case is frequently called a "double-delete error", so if you hear someone talking about "double delete", you'll know what they mean. Notice that in my program above, I set the pointer *p* to zero so that I could not possibly commit any use-after-free errors. I think this is a great habit. It won't catch/prevent all your errors, but it will help.

The Heap Implementation – Chunks and Signatures

The heap is nothing more than a large contiguous collection of memory locations – i.e., an array. This array is allocated by the operating system as part of the memory for the program to run. When *malloc* is called the first time, *malloc* initializes the array to indicate that none of the locations within the array have been used yet. In other words, the entire array is available as storage. Each time that *malloc* is called, it selects a set of locations from within this array that can be used to satisfy the current request. Then *malloc* returns the address of the first location from this set.

As *malloc* and *free* are called, different sets of locations are allocated and deallocated for use as storage. Note that any kind of information can be stored in the memory locations inside this array. As far as *malloc* is concerned, the array may be an array of **int**s. When *malloc* returns, it will return the address of one of those **int**s. You may decide to store **char**s in that location, or **float**s or anything you'd like. It's just a memory location, after all, and can hold any type of information.

Somehow, *malloc* and *free* need to coordinate which memory locations are being actively used to store information, and which locations are still available. We will refer to the collections of consecutive locations as "chunks" of memory. A chunk could be as small as just one location in the array, or perhaps several thousands of locations. The memory of the heap will be broken down into one or more of these chunks. Each call to *malloc* may split one large chunk into two smaller chunks, and each call to *free* may end up combining two (or perhaps three) adjacent chunks into one larger chunk.

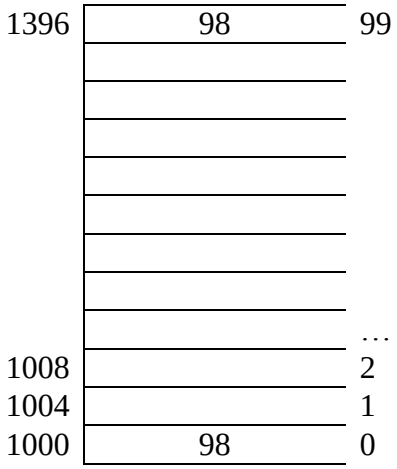
To keep track of the status of each chunk, *malloc* and *free* use two memory locations for each chunk to store "signatures". The first location in the chunk will be the "bottom signature" and the last location in the chunk will be the "top signature". The value stored in the two signatures will always be the same. The value will be determined as follows:

- The signature will be negative if the chunk is actively being used to store information
- The absolute value of the signature will be the size of the chunk

Each time *malloc* is called, it looks through the heap, examining each of the signatures. Based on the value of the signature, *malloc* can determine if the chunk is available, and if it is large enough to satisfy the current request.

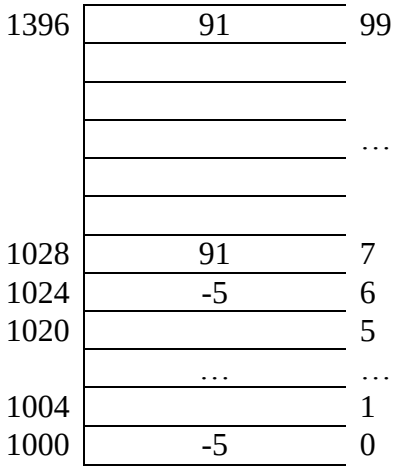
Examples

Consider the following diagram showing an array of 100 memory locations (all ints). This particular array happens to start at address 1000 (addresses are shown on the left of each location, the indices are shown on the right). Since the array is 100 elements long, and since each element is four bytes, the last position in the array is address 1396. Two of the array elements have been assigned values. These are the signatures for one large chunk. Based upon our conventions for signatures, we can see that the chunk is 98 locations long, and it is available to use for storage. Note that the chunk is only 98 locations long and not 100. Two of the locations are wasted holding the signatures.



Now consider what happens if someone were to call *malloc* in an attempt to allocate 20 bytes of memory. In this case, *malloc* takes the value of the parameter and immediately divides it by the size of an **int**. In this case, 20 is evenly divisible by 4, so we clearly need to use 5 array locations to satisfy this request. If the size requested had not been evenly divisible by 4, then we would need to be sure to round up to the next integer.

To allocate 5 array locations, we need to first decide which 5. In this case, we'll pick locations 1 through 5 (since there doesn't seem to be any obvious reason not to pick them). We'll need to create two signatures to mark the beginning and end of this new chunk, and we'll need to create new signatures to properly indicate the remaining size of the heap. The resulting state of the heap is:

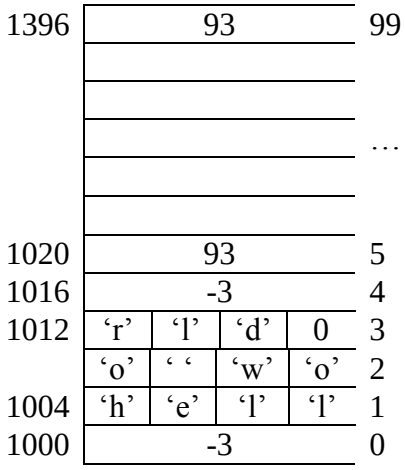


Note that the heap now contains two chunks. The first chunk occupies a total of seven locations (including the two signatures) and the second chunk occupies the remaining 93 locations in the array. The last thing that *malloc* must do is to return the address of the first byte of available storage. As we can see in our diagram, this address is 1004.

In each of the examples that follows, we'll use the same set of assumptions each time. We'll assume that the heap is an array of 100 **int** locations that happen to begin at address 1000. On a real computer, the heap would be a much larger array, of course, and it could start at just about any address. The examples show a sequence of calls to *malloc* and *free* and illustrate the state of the heap after that sequence.

Example 1

```
char* p;
p = malloc(12); // 12 bytes requires 3 int locations, p is assigned 1004
strcpy(p, "hello world");
```



Note that my diagram shows four bytes in each row. Since characters take only one byte, I can fit four of the characters in the string in each row. The first character is stored in location 1004. The zero marking the end of the string is stored in location 1015.

Example 2

```
int* p = malloc(4); // p gets the value 1004
*p = 42;
int* q = malloc(4); // q gets the value 1016
*q = 17;
free(p);
```

1396	92	99
		…
1024	92	6
1020	-1	5
1016	17	4
1012	-1	3
1008	1	2
1004	42	1
1000	1	0

Notice that p is still equal to 1004, and that the 42 is still stored in that location. The call to *free* does not change anything other than the signatures.

Notice that the first chunk is available (the signatures are positive), so if *malloc* were to be called again, it might use this chunk to satisfy the request. So, for example, if we continue the example with two more calls to *malloc*.

```
int* r = malloc(4); // r is set to 1004, note that p = = r
int* s = malloc(4); // s is set to 1028
```

1396	89	99
		…
	89	9
1032	-1	8
s points here 1028	???	7
1024	-1	6
1020	-1	5
q points here 1016	17	4
1012	-1	3
1008	-1	2
r points here 1004	42	1
1000	-1	0

Note that the value of `*s` is undefined – we have no idea what value might be stored in memory location 1028. However, `*r` is equal to 42. The old value is still sitting in location 1004, and nothing has been done to change it.

In general we should always assume that the value stored in our dynamically allocated memory (e.g., the value stored in location 1028 or location 1004) is undefined. A slight change to the order in which *malloc* or *free* is called can completely change which memory locations get used for each allocation request, so it's essentially impossible to predict which memory locations will get reused. For example, if we move the call to *free(p)* down so that it is done between the calls to *malloc* for *r* and *s* then, *r will be undefined and *s will be 42.

```
int* p = malloc(4); // p gets the value 1004
*p = 42;
int* q = malloc(4); // q gets the value 1016
*q = 17;
int* r = malloc(4); // *r is undefined
free(p);
int* s = malloc(4); // *s is 42
```

Most good programmers assume that calling *free* destroys the values stored in memory. Of course, it doesn't do this at all. Calling *free* merely updates the signatures, and all the old values are still in memory. But, if we assume that those values are gone, we'll never be tempted to try and write programs that access those values. Accessing values in memory locations that have been *free*'d is so likely to result in bugs in our programs, it's best to do what we can to avoid it.