Motivation for Dynamic Allocation

Recall the task of writing a function that takes an integer, creates an array of the size specified by the integer, initializes each field, and returns the array back to the caller. Given the tools we had thus far, our code (which would not work!) would look like this:

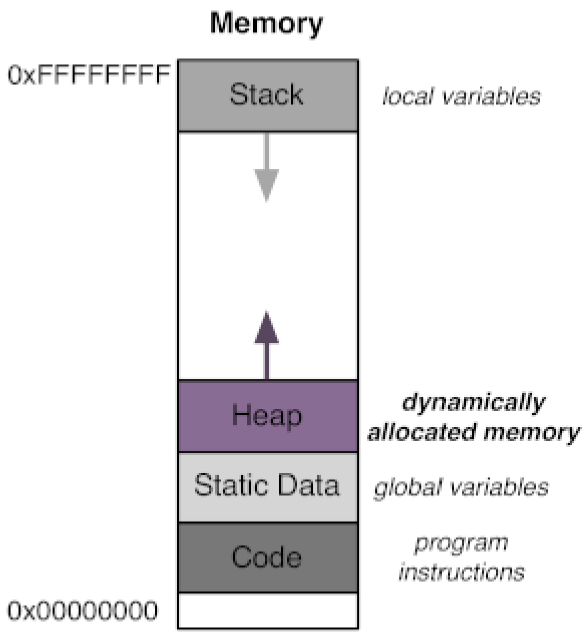
```
  int myArray[howLarge];
  for (int i = 0; i < howLarge; i++) {
    myArray[i] = i;
  }
  return myArray;
}
```

The reason that this code will not work is that the array is created on the stack. Variables on the stack exist only until the function ends, at which point, the stack frame essentially disappears.

While it may not seem that important from this example to be able to create an array and return it, programmers often want to write functions which perform more complex creation tasks (and have the results persist beyond the function that created them). Suppose you needed to read information from a file (possibly with a complex format). You might want to write a function to read the information, allocating memory to hold the results in some combination of arrays and structs, and return the result to the caller. Fortunately, there is a way to do exactly this: *dynamic memory allocation*.

Dynamic memory allocation allows a programmer to request a specific amount memory to be allocated *on the heap* (highlighted in purple in the figure below)—not the stack. Because the memory is not in the stack frame, it is not freed when the function returns.

Instead, the programmer must explicitly free the memory when she is done using it.



Dynamic memory allocation involves both allocating memory (with the **malloc** function), and freeing that memory when you no longer need it (with the **free** function). Programmers may also wish to reallocate a block of memory at a different size (for example, you may think an array of 8 elements is sufficient, then read some input and find that you need 16). The **realloc** function allows a programmer to do exactly this—asking the standard library to allocate a new (larger or smaller) block of memory, copy the contents of the original to the new one, and free the old one (although the library *may* optimize this procedure if it can expand the block in place). For all three of these functions, include ***<stdlib.h>***. We will also see a wonderful function, **getline**, for reading strings of arbitrary length using dynamic allocation.

✓

Completed