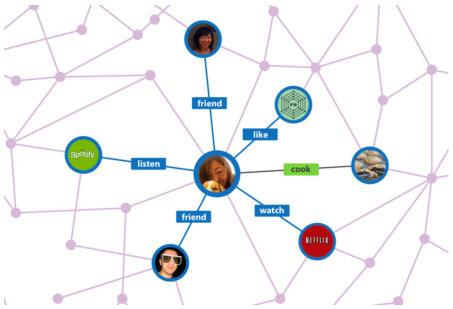
Dynamic Memory Allocation

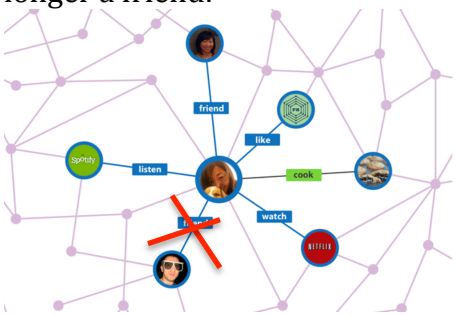
Consider the following scenarios:
Ex 1:
You read text in from a file, and you know ahead of time the maximum number of lines you're going to see, so if you're storing the information you read in you know exactly how much memory you need to allocate. We see this on this week's assignment where you know there are a maximum of 100 items in the array.

Ex 2:
You have a network of objects, or other variable type, where there are a fixed number of objects in the network and you will never need to add anything to the network. Perhaps we're talking about a computational way of representing this graph:



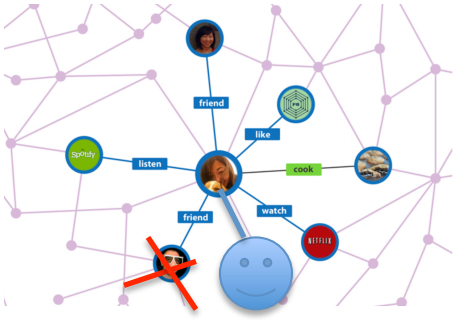
Each of these nodes in the graph lives somewhere in memory in your program, and when the program runs, memory is allocated for the node.

However, the more likely case in computing is that you don't know how many lines are in the file, or how much data you have. These are the cases when the memory allocation needs to change while the memory is running. We need to change memory allocation dynamically. For example, let's say the dude in the picture is no longer a friend:



Whatever representation we have that links two nodes together needs to change. We need to free up the memory currently allocated to the dude node to make room for someone else. If we want to add a new connection while the program is running,

we need to add a new node, and dynamically allocate the memory to store that information.



In some languages, there are interfaces that will handle this memory management for you. In Python, you can use a list, and then do list.add(), which allocates memory for another item and you as the developer don't have to think about it. In C++, there is a Vector that has a push() method that allows you to add a an element dynamically. (Behind the scenes, both of these probably use some sort of linked list data structure, which we will start discussing next week.)

It's an important distinction here that Vectors and Lists are not data structures, but rather, interfaces that have been created by those languages. If you were working in some other language, such as C, you wouldn't have access to Vectors. Arrays and dynamic array manipulation is the best you can do. What we're concerned with in this class is not interfaces, such as Vectors, but rather, the data structures and algorithms underlying those interfaces. We're going to do this using C++ with as little overhead as possible. To do this, we need to be interacting as closely as possible with memory and memory management.

It's important to talk about arrays for a few reasons. This is how we generally think about a sequence of data, and the array data type is available in most programming languages. There are also common operations involving arrays that you will see in complicated algorithms for searching and sorting, such as indexing, shifting, and doubling. Having a solid grasp of array manipulation will help you in understanding these algorithms.

Arrays do also have their limitations: primarily they have a fixed size. With any data structure, it's important to understand the strengths and limitations as well as how to work with the limitations if that data structure is what you have available in a given language. For arrays, the most common algorithm for handling array re-sizing is called array doubling. This algorithm is used when you have more data, but your array is full and you need a larger array for holding the additional data. The algorithm uses the size of the current array and generates a new array dynamically that is twice the size of the current array, then copies the values from the current array into the first half of the new array. This copying over of data into the new array has an O(n) complexity.

Memory background information
There are two areas of memory that we are most concerned about in this class, the call stack and the heap.

The Stack
This is where local variables are stored. When you create a regular variable inside a function it is a local variable. There is limited space available on the stack, so it's important to know how much space you need.

The Heap
The heap is a large pool of free memory, much larger than the stack that is used for storing variables that are created dynamically while the program is running. When you create variables using pointers they are created on the heap.

When we start involving the heap in our programs, there are some additional responsibilities we take on with memory management. Unlike the stack where memory is allocated and de-allocated for you, you have to explicitly do these tasks yourself when you're working with the heap.

The advantages and disadvantages of the heap are:

1) Allocated memory stays allocated until it is specifically deallocated (beware memory leaks).
2) Dynamically allocated memory must be accessed through a pointer.
3) Because the heap is a big pool of memory, large arrays, structures, or classes should be allocated here.