Memory

Computers have a variety of memory types. Inside of the CPU there are at least three levels of cache memory, which store data and chunks of files, allowing each thread to run as fast as it can. Modern graphics cards have a phenomenal amount of memory, normally limited to use on the card. However, you can access that extra memory (and processor power) to increase the speed of your application. Even more memory can be added with a USB memory stick using Flash technology.

The memory you will be using most often is RAM (random access memory). This is available to your processor via the memory bus. Depending on the architecture of the chip, this bus may also have data and addresses flowing along it. The operating system sees RAM two different ways: as stack memory or heap memory. Stack memory is used when you create an array in your application, or when you use recursion in a subroutine. Heap memory is the most commonly used memory, directly accessible by the programmer. It is manipulated through the use of these commands: malloc() and free() in C, or new() and delete() while using C++.

When you launch your application the operating system gives it access to a multi-gigabyte chunk of memory. One half of that memory is used by the OS for loading linked libraries and other things necessary to keep the application and the OS in communication. The other half is available for you, the programmer, to use for your application. Your chunk of working memory is allocated in two different ways. First as a stack, the second as a heap. The stack memory is used for arrays, which are accessed with an index variable, statically allocated at compile time. The heap is accessed in chunks, which are dynamically allocated at run time.

While there is a vast amount of heap memory, stack memory space is limited. The OS provides a certain amount when a process begins. Each stack frame contains a number of values about the state of the CPU. These include: return address, function parameters, and local variables of the function. When a function recurses another stack frame is built, and populated with data about the new CPU state. Make sure any recursive functions are not overly deep, or you will hit the stack's memory limit since there is only so much stack space available. Creating very large arrays is another way to hit that OS defined limit. If you want to solve large problems use heap memory, then access it via pointers mimicking an array address – such as temperature[i][j]. Using this style:

```
temperature = (double *) malloc( sizeof( double ) * ( n+1 ) * ( n+1 ));

for (i=0; i<900; i++)
  for (j=0; j<900; j++)
    temperature[i*n + j] = 995.0;  // n = width of space
```

you can access memory allocated from the heap using an array indexing technique. Thus far I've been able to allocate 4 GB of space as one 'pseudo' array to use for my simulations. I have used much more as independently allocated objects.

I have focused on teaching you about using memory in chunks since the very beginning of this course. Hopefully, this has helped you learn how and when to use pointers. Memory is very cheap these days. Have fun using as much of it as you can.

Memory deallocation is performed differently for each language. It is called garbage collection. Java and Python garbage collect automatically, but have their own little gotchas. In the middle of something they may decide to garbage collect, instead of doing the programmer's bidding. Bogs things down immensely. In C you are responsible for doing it yourself. It is left to the OS to recover the memory in applications which do something, then exit. In a long running application the programmer needs to be more careful. It is best to release the memory which you have allocated. In C, use malloc() to allocate heap memory, and use free() to release that memory back to the free pool. In C++ things are a little different, it requires new() and delete().

You want to set any 'floating' pointers to NULL. This really helps while debugging. Yes, you immediately assign it to your newest node but, consistency really helps. When you are creating a new node begin with all unassigned nodes pointing at NULL. There are a lot of pointers getting changed when you delete a node from a doubly-linked list,. Keeping all that straight takes a bit of drawing, and some discipline. Standardizing your methods helps you find any bugs you may have introduced. Make sure to keep track of your heap memory allocation. Both in their allocation, and during their release back to the OS. This makes your life more serene.