Memory: New, Delete, & Arrays

So we are talking about memory management. In the last lecture, we talked about stack memory and heap memory, and the fact that local variables are stored on the heap, whereas persistent data allocated by the `new` operator is allocated on the heap.

Note that I use the term `variable` in the case of local variables, but not for heap data. A variable is something that, among other properties, has a name. Data on the heap does not; it is just data. I point this out because Java programmers often confuse an object on the heap (which is not a variable) with its reference, which has a name and is a local variable. In C++, an object can be a variable in local memory (i.e. on the stack), or it can be a data object on the heap.

Before we ask that, why would we want to put data on the heap? Two reasons:

1. Data on the stack falls out of scope and is reclaimed when the function that allocates is returns. This means the data cannot persist longer than the function. Data on the heap, on the other hand, is persistent.
2. The size of the stack frame is determined at compile time. Therefore, if data is to be put on the stack, the compiler must know its size at compile time. The size of data on the heap, on the other hand, can be determined at run time.

In the last lecture, I also introduced pointers as typed virtual memory addresses. The value of a pointer is a virtual address. Nothing more, nothing less. And a virtual address is just a data value like any other.

References, of course, are similar to pointers. They are a way of accessing data in memory. References, however, are an abstract language concept. They provide named access to data allocated earlier. You do not know how they are implemented (although they are often implemented as pointers). They are not recursive (no references to references). Pointers are lower level: they are manipulate-able machine addresses, nothing more and nothing less.

Note that there is no direct analog to pointers in Java (just references).

Why introduce pointers? Because is C++ you have complete control over memory, and to do that we need pointers.

Having direct access to memory addresses is sometimes useful, but always dangerous. Here are some things that can go wrong:

- After I delete an object, I still have the pointer. Good style says to set the pointer to NULL after calling delete, because if I use the pointer again, the object is gone…
- If I have two pointers to an object and I use one to delete it, the other pointer is now invalid (but only you know that). Having two pointers to a single object is called aliasing. It has uses, but never do it accidentally.
- If I allocate an object on the heap and loose its pointer, then I can never reuse that memory. This is called a memory leak.
- If I create a pointer to a local variable and the function returns, my pointer is invalid (but only you know it). This is the more common case of the “return a local reference” bug mentioned earlier.
• If I create a pointer to a local variable and delete it, expect the machine to crash (since you are trying to give the stack back to the heap)

So why am I harping on where data lives, when it is created and destroyed, and when constructors and destructors are invoked?

Because the number #1 factor that separates C++ programmers from hackers is memory management. You have to manage your own data. Do it well and consistently, and you will have fast C++ programs. Do it badly, and you will be bedeviled by hard to find bugs.

Today’s new topic is arrays. If we want to use the heap to store dynamically sized data structures, we need to have arrays. After all, if we want to store a Quagga somewhere, we can just put it on the heap. But what if have a heard of Quagga’s, and we don’t know how many Quagga’s are in it until we read it from a file?

The answer: we make an array of Quaggas. The thing to realize about an array is that it is not an object, like std::vector or Java’s ArrayList. It is a primitive.

Arrays are just blocks of contiguous memory. For example, if a write new int[14] I am allocating enough memory on the stack to hold 14 ints. I can allocate an array on the stack, too, by writing int foo[5]; but the size of the array must be a compile-time constant. I cannot put a structure of unknown size on the stack. With the new operator, the size can be determined at run-time.

Note, too, that the memory is typed – but only at compile time by the compiler. In other words, if I write new int[14] the compiler knows that the memory allocated contains integers. But it doesn’t write anything into the memory about its data type; the data type of the memory is just used to compile your source code.

What is the value returned by the new operator? It is a pointer to the first element in the array. For example, int* foo = new int[14]; As a result, foo[0] is the first element of the array, and foo[1] is the second. The reasons array indices in C++ start from 0 is that they are offsets added to the pointer to the first element. The first element needs no offset, so the offset value is 0.

Why are indices offsets? Pointer arithmetic! Foo[2] is the same thing as *(foo+2). Pointer arithmetic allows you to add or subtract integers to/from pointers in order to move through memory. Note that pointers are typed, so it knows how big an element of the array is. For example, if q_array is an array of Quaggas, then q_array+1 is the address of the 2nd Quagga, not one word past the start of the 1st Quagga.

The problem most people have with C++ arrays is in believing how simple they are. They are just blocks of memory. The array variable is just a pointer to the first element in the array. Arrays are not objects, and therefore have no methods. You cannot ask an array how big it is (or anything else).

Arrays on the heap are allocated using new, just like any other data type, and new returns the pointer to the first element. What happens when you want to delete the array? Since the array is just a pointer to an element, if you write delete foo (where foo is an array of Quaggas), you are just asking it to delete a single element (the first element in the array). This is probably a bug.
Assuming your goal is to delete the whole array, you should write: `delete [] foo;` The square brackets tell the system to delete not just the first element, but the whole array.

One type of array is special (although only by convention). Before objects were added to C to make C++, programmers needed to manipulate strings. Since C had arrays, strings were just arrays of char, and pointers to strings had type char*. But there was a problem. If you were handed a char*, how did you know how long the string was? After all, any character could be part of a string, so it could just keep going and going…

To determine the length of strings, a (strong) convention was adopted. To note when a string ended, it was decided that all strings would end with the NULL character (ASCII 0).

Today, C++ has the std::string class. If you want to represent a string, that’s what you should use. But C-style strings of type char* persist. For example, look at the arguments to main: the second argument is of type char**, because it is an array of C-style strings.

Even beyond main, C-style strings remain common. For example, the constructor to ifstream takes a filename as a char*. And most legacy code is full of char*.

So you should know the C-style string convention of ending in NULL. In particular, if you ever have to allocate a C-style , make sure to allocate an extra byte for the NULL byte. For example, if you want to store the string “foo” in a C-style manner on the heap, allocate an array of 4 chars, and then write four bytes: ‘f’, ‘o’, ‘o’, NULL.