MTH 4300, Lecture 9

Dynamic Memory; Lifetime and Deallocation; Some Hazards of Pointers; Jagged Arrays

E Fink

March 3, 2025

1. Dynamic Memory

Pointers are also essential for managing *dynamic memory*.

Remember how with arrays, you needed to know the size at compile time? That's because the memory for all the variables we've used so far has been allocated on the *stack*. The allocation of memory slots in stack frames is very difficult without knowing how much memory is needed in advance.

However, you may also allocate memory dynamically – that is, during execution of a program. This is great if you don't know how much data you're going to keep track of in your program. Dynamically allocated memory resides in a different portion of memory, known as the *heap*.

Dynamic Memory

To declare a new dynamic variable, we would write a line like

```
int *x = new int;
```

The `new` operator:

- searches the heap memory for enough consecutive, *unreserved* memory to hold an `int` object;
- uses that memory for a new object, "marking the memory as in-use";
- and returns the address of this memory (which would be stored to `x` in this example).

So, the new object won't have a name, but that's okay, because it will be accessible via the pointer `x`.

Dynamic Memory

If we wish to store an *array*, very little changes: your line would simply be something like, e.g.,

```
int *x = new int[10];
```

with a pair of brackets, and a length – *and that length CAN be a variable!* This line would search for enough space to hold 10 consecutive ints in memory.

You could then fill this array by any of the following methods:

```
*x = 4; // Fills the first entry
*(x+2) = 6; // Fills the third entry
x[3] = 10; // Fills the fourth entry
```

For the last one, remember what we said before, `x[3]` is the same as `*(x+3)`!

L9x1_dynamic.cpp

Stack-allocated variables have an *automatic* lifetime – they cease to exist when their enclosing function completes execution.

On the other hand, heap-allocated objects have an extended lifetime: they need to be *explicit deallocated*.

If `p` is a pointer of some type, you can deallocate the memory reserved at `p` by

```
delete p;   // Deallocate a single variable
```

or

```
delete[] p; // Deallocate an array
```

depending upon whether `p` pointed to a single variable or an array.

L9x2_delete.cpp

Note: the `delete` operator does not erase any of the contents held at the address in question; it simply marks it as "no longer in use," making it free for later allocations.

Lifetime and Deallocation

What happens if you fail to call `delete` or `delete[]` on your heap-allocated objects? If you only do it once: probably nothing. Most modern operating systems will automatically deallocate all memory after your program has finished running.

However, if you repeatedly fail to deallocate large quantities of memory:

- your program could slow down, since finding remaining free space in the heap will get harder;
- or you could run out of space in the heap, at which point the behavior is undefined, but your program would likely crash.

L9x3_leak.cpp

A *dangling pointer* is a "pointer without an object." More accurately, it is a pointer whose contained address is either complete uninitialized garbage, or which holds the address of an old heap-allocated object that has been deallocated using `delete`.

The latter can happen easily when the value of one pointer is assigned to another.

L9x4_dangle.cpp

Some Hazards of Pointers

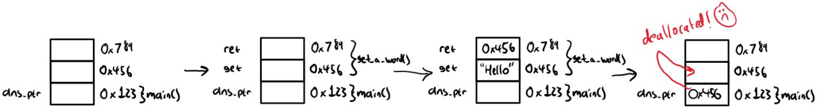
An *inaccessible object* is the opposite – it is an "object without a pointer." More accurately, it is a heap-allocated object with no pointer containing its address.

This can happen if the value returned by `new` isn't assigned to a pointer variable; but more realistically, it can happen when the object's old pointer gets assigned the address of a new object.

L9x5_inaccess.cpp

When functions return pointers, it is very important that those references are to heap-allocated objects. If you return a reference which points to a local variable, that will cause a problem, because *local variables get deallocated automatically once the function returns!*

L9x6_return.cpp



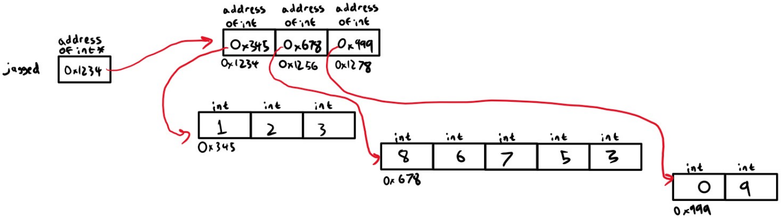
We've seen 2D arrays before, which have a fixed “height” and “width”. In particular, the arrays contained within the array has the same length. But what if you want a *jagged array*: an array containing differently-sized arrays?

One can use a **double** pointer: e.g.

```
int **jagged;
```

To interpret this, `jagged` will be a pointer, but instead of being a pointer to an `int`, it is a pointer to an `int*`. Therefore:

- `jagged` can hold the address of (the first entry of) an array of `int*`s
- Each entry holds the address of (the first entry of) an array of `int`s



L9x7_jag.cpp

Jagged Arrays

L9x8_tri.cpp

Create an array `triangle` with 5 entries, each of which is an array of `int`s. The first entry is an array of length 1, the second entry is an array of length 2, and so on. I recommend a SINGLE for-loop for this.

0				
1	2			
3	4	5		
6	7	8	9	
10	11	12	13	14

Then, fill the array with entries like the picture shows. For this one, I recommend a NESTED for-loop.

Oh, and we should probably deallocate the array as well!