MTH 4300, Lecture 8

Towers of Hanoi; Pointers; Pointers and Arrays; Dynamic Memory; Lifetime and Deallocation

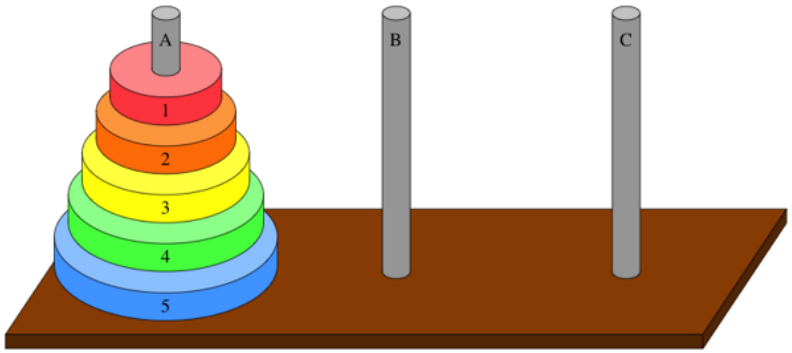
E Fink

February 25, 2025

1. Towers of Hanoi

This is one of the most famous recursive problems. There are n disks on the left pole, stacked in order of decreasing size. The goal is to move this stack to the right pole.

The rules: you can only move one disk at a time, and you can never place a disk on to of a smaller disk.



https://www.mathsisfun.com/games/towerofhanoi.html

Towers of Hanoi

How do we solve this? Here's a strategy:

- Move the smallest $n-1$ disks temporarily to the pole we're **not** targeting, leaving the biggest disk uncovered.
- Move the biggest disk from the original pole to the pole we **are** targeting.
- Move the smallest $n-1$ disks from the pole they're on to the target pole.

But how do we move the smallest $n-1$ disks? That's what recursion is for. (What would the base case be? The instructions only make sense when $n > 1$, so $n = 1$ ought to be the base case.)

Let's write a program that gives explicit instructions for the disk moves. Every instruction will be of the form

Move disk d from pole x to pole y.

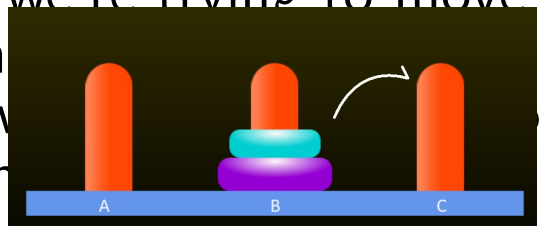
L8x1_hanoi.cpp

Towers of Hanoi

One of the trickiest parts of this problem is understanding what the signature line of the function should be. I have written it as:

```
void towers(int num_d, char f, char t, char aux)
```

where `num_d` represents the number of disks we're trying to move; `f` represents the "from" pole (A, B or C) wh
disks from, and `t` represents the pole which v
And `aux` represents the third pole, which can
temporary storage.



For example, for this picture, I would call

and it would print

2. Pointers

One of the reasons for using C++ is that it allows you to work with memory in a fairly direct manner, so that you can design data structures that are complex, but can still be manipulated efficiently. To do this, we need to be able to work with addresses.

A pointer is a variable that is meant to hold the *address* of a variable of a particular type. For example, suppose you wanted a variable `p` that contains the address of an `int`. Then you would declare

```
int* p;
```

The `*` means you're not declaring an `int` variable, but rather a variable which holds the address of an `int` (a "pointer to an int"). You could then store the address of `int x` to that variable by writing

```
p = &x;
```

Note that the declaration of the pointer variable can also be written as

```
int *p;
```

with the asterisk attached to the variable rather than the data type. This is more common, in fact, because it allows declarations like

```
int *p, q, r, *s;
```

where `q` and `r` are regular `int` variables, and `p` and `s` are pointers to `int`s.

Pointers

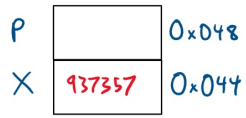
In addition to playing a role in declaration, the * symbol also is an operator: the dereference operator.

> *If you have a pointer variable p whose value is an address, then *p is the value stored at that address.*

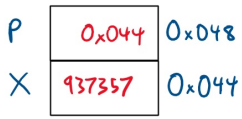
In other words, `*p` means: look at the address in `p`; go to that address; and read the value that is there.

L8x2_pointer.cpp

When we declare `int x = 937357` and `int *p`, here's what's getting created in the stack:



Then, when we set `p = &x`, here's how things get updated:



Finally, when we `cout << *p`, we first read the content of `p` – which is an address, 0x044 – then go to that address and read the contents *there*: 937357.

Pointers

Pointer variables can of course be used as the left side or as the right side of an assignment. E.g., if `p` and `q` are both pointers of the same type, then

```
p = q;
```

would cause the address stored in `q` to be copied into `p`.

Likewise, `*p` and `*q` can be used on both sides of an assignment:

```
*p = *q;
```

would take the value stored at address `*q`, and assign it to the memory space whose address is `*p`.

L8x3_value.cpp

(We sometimes say that `*p` is an *l-value* since it can be used on the left side of an assignment.)

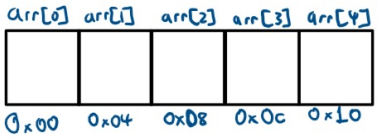
Whenever the name of an array is referenced without referring to a particular element, it is treated as a pointer. This phenomenon is known as the array *decaying* to a pointer.

For example: suppose I declare `int arr[5]`. As we discussed,

`cout << arr`

WON'T print out the entire array – instead it will print out the address of the first element. (`char` arrays are an exception to this behavior.)

Recall that arrays are stored in memory in consecutive addresses:



Then, `arr` would be `0x00`. Also, if you add an integer `n` to an array, you obtain the address of the entry `n` positions down the array. So, for example, `arr + 1` would be the address of `arr[1]`, which would be `0x04`; `arr + 2` would be `0x08`, etc. In fact, `arr[i]` is just a shorthand way of writing `*(arr + i)`. This is where 0-based indexing comes from!

L8x4_array.cpp

Note that attempting to access out-of-bounds memory is undefined behavior. One possibility is a *segmentation fault*, where a program attempts to access a memory location outside of that reserved for the program.

Pointers and Arrays

The following example illustrates a couple of points.

- If a formal parameter of a function is declared as type `T *` for any data type `T`, then that function can be called with arrays of type `T` as parameters.
- Pointers are frequently used to traverse an array. For example, within the code

```
int arr[3] = {14, 25, 36};
int *end = arr + 3;
for(int *item = arr; item != end; ++item){ ...
```

the variable `item` would hold the address of each element of `arr` in succession.

L8x5_function.cpp

4. 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)`!

L8x6_dynamic.cpp

5. Lifetime and Deallocation

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.

L8x7_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.

L8x8_leak.cpp