

**”DO NOT MISTAKE
THE POINTING
FINGER FOR THE
MOON.” – ZEN
SAYING**

ADAM SWEENEY

CS 211

INTRODUCTION

- We should be familiar with the fact that variables have memory addresses
- We have been circling this topic for a while now; finally time to dive in

AGENDA

- What is a pointer?
- How to declare a pointer
- How to use a pointer
- The real power of pointers



**WHAT IS A
POINTER?**

WHAT EVEN IS A POINTER?

- A pointer is a variable
- Its value is always a memory address
- What is the data type for a pointer?
 - It depends what it's pointing to
- We NEVER have “just” a pointer
 - Pointer to an int
 - Pointer to a double
 - ...etc.

HOW WE ACCESS COMPUTER MEMORY


- We visualize RAM as a series of bytes
- When a variable is declared, a properly sized piece of memory is set aside for that variable
- We access that memory by using the name of the variable
 - We know that every byte has an address
 - If we know the address of a variable, we can use it to gain access as well
- Pass-by-reference uses memory addresses to pass variables to functions

LOOKING BEHIND THE CURTAIN

- Using variable names or pass by reference hide their memory access abilities from us
 - The concept they give us is more important
- But now we will dive in ourselves

A PITFALL

- The value of a pointer variable is a memory address
- A memory address is an integer
- **A pointer is not an integer**
- This is intended
- When we deal with pointers, we are expected to deal in pointers
- Unlike `char` and `int`, it is not possible to mix types

A decorative wavy line in yellow and white on the left side of the image.

HOW TO DECLARE A POINTER

IT'S NOT SO BAD

- Declaring pointers is fairly straightforward

```
int *intPtr, x = 42, *y;
```

- The `*` character denotes that the variable being declared is a pointer to that data type
- In the line above, three variables were declared
 - A pointer to an integer named `intPtr`
 - An integer named `x` with an initial value of 42 (not a pointer!)
 - A pointer to an integer named `y`

NOW LET'S ACTUALLY POINT TO SOMETHING

- Pointer declaration is not so bad
- How do we tell them to point to something?

```
int *intPtr, x = 42, *y;  
y = x;
```

- That does NOT work (It will fail to compile)
- The types don't match
 - You are trying to assign the value of an integer into a pointer
 - Pointers only hold addresses

ACTUALLY POINTING

- To assign to a pointer I must give it the “address of” something

- ```
int *intPtr, x = 42, *y;
y = &x;
```

# OPERATORS FOR DEALING WITH POINTERS

- The “address of” operator, `&`, helps us assign addresses to pointers
- Typically an operator in C++ has one job, and one job only
- The star, `*`, now has two jobs
  - The first is multiplication
  - But we’ve seen it used with pointers as well
    - And even with pointers, it does two different things
- We use the `*` to declare a variable as a pointer to that type
- We also use the `*` to *dereference* a pointer



# **HOW TO USE A POINTER**

# USING POINTERS

- We have been using pointers in disguise (arrays)
- Go over a few examples to show how certain pointer assignments behave, and to illustrate the difference between the value of a pointer and the value of the data being pointed to

# A COUPLE QUESTIONS

- What mis-representation can occur with the following declaration?

```
int* intPtr1, intPtr2;
```

- Judging by the names of the variables, it appears that both are intended to be pointers to integers. Attaching the \* to the data type does not work like that. Only `intPtr1` is a pointer to an integer, `intPtr2` is just an integer
- With regards to pointers, what are the two uses of the \* operator?
  - At the time of declaration, the \* indicates that you are declaring a pointer to the data type
  - Outside the declaration, the \* de-references the pointer, meaning it goes to address and gets the value held there





# **THE REAL POWER OF POINTERS**

# IT'S OVER 9000!

- All programs that execute on a computer have access to two types of memory
- Up to this point, we have only been accessing one of those memories
- With pointers, we can access the other

# THE STACK

- It's what we've been using so far
- The stack is very compact, very organized memory
  - Building it requires knowing all the pieces ahead of time
- Items in the stack cannot change their size, and items in the stack typically exist for the duration of the program's execution

# STACK MEMORY

- All functions and variables go into the stack
- There is no wasted space
  - This inhibits our ability to grow/shrink/create/destroy items “on the fly”
- This is why array capacities cannot change
- We can also say that this memory is static. i.e., once set it cannot be changed
- All items going to the stack must be known at compile time

# THE HEAP

- It is only accessible through pointers
- The heap is far less organized
- Items can come and go while a program is being executed

# HEAP MEMORY

- It is not as compact or organized as the stack
- Things like array size do not have to be known at compile time
  - Instead, we can figure it out during run time
- Items can be created and destroyed while a program is running
- This memory is also called dynamic memory

# ACCESSING HEAP MEMORY

- We need some new syntax
- The idea is that we are requesting, and subsequently borrowing heap-allocated memory

```
int *dynInt = new int;
int *dynInt = new int(42);
```

- The keyword `new` requests memory from the heap, and it is given to us in the form as a pointer

# ABOUT THAT BORROWED MEMORY

- `int *dynInt = new int;`
- The integer we borrow from the heap has no name
  - Our only access to that integer is through the pointer
- It was mentioned (a few times) that this memory is borrowed
- When we borrow things, we should give them back when we're done



# RETURNING MEMORY TO THE HEAP

- `int *dynInt = new int;`
- We learned some new syntax to request memory from the heap
- It follows that there will be some new syntax to give it back  
`delete dynInt;`
- The keyword `delete` destroys the dynamically allocated object and returns the memory to the heap

# BORROWING MEMORY REQUIRES MANNERS

- If we borrow memory using `new` and do not return it using `delete`, the compiler does not complain
- But we have done a bad thing
- It's called a memory leak
- Modern OS's can help mitigate them, but it's still terrible practice to leak memory
- If we borrow memory, we should give it back when we're done

# THE HEAP IS (SOMEWHAT) LIMITED

- If there is memory in the heap to give, you will always receive that memory
- The heap can run out of memory
- You will see an exception thrown while the program is running, `std::bad_alloc`
- We will likely not see that in this class, but we should be aware
- Exception handling is not discussed in this course

# THE HEAP IS PASSIVE

- The heap will NOT take back memory it has lent out
- If we remove all pointers to that memory, it now exists in a purgatory state
  - It's impossible to get it back, and it will not return to the heap until the program ends
- Modern OS's make it possible for that memory to be returned when a program ends
- Users used to be required to reboot their computer to reclaim that memory

# DELETING IS NOT QUITE ENOUGH

- Deleting memory returns the dynamic memory to the heap
- But what about the pointer?
- The pointer still has a value
- But the memory at that address “no longer exists”
- What happens if we try to de-reference that pointer?
- The C++ bogeyman, undefined behavior
  - Typically a compile or runtime error



This Photo by Unknown Author is licensed under [CC BY-NC-ND](#)

# DON'T LEAVE ME HANGIN'

- A pointer in this state is called a dangling pointer
- It is our responsibility as programmers to make sure pointers don't dangle

```
int *dynInt = new int;
// DO STUFF
delete dynInt;
dynInt = nullptr;
```

- You might see other syntax; don't use it

# GOOD PRACTICES (FOR INTRO)

- If you request heap memory using the keyword `new`, immediately write the `delete` and `set-to-nullptr` statements
- Write your code between the allocation and de-allocation
- When you're done with the memory, you don't have to remember to delete it, since it's been done already
- Once you start writing Object-Oriented (or functional) programs, this is neatly handled in other ways