“Do not mistake the pointing finger for the moon.”

– Zen Saying

CS 211

Adam Sweeney

October 27, 2017

Wichita State University, EECS

Introduction

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

Agenda

- Section 9.1
- What is a pointer?
- How to declare a pointer
- How to use a pointer
- The real power of pointers

What is a pointer?

What is a pointer?

- First and foremost, a pointer is a variable
- Its value is always a memory addresses
- What is the data type for a pointer, then?
  - It depends what it's pointing to
- This is an extremely important distinction
- We never have "just" a pointer
  - Pointer to an `int`
  - Pointer to a `double`
  - Pointer to a `char`
  - . . . etc.

How We Access Computer Memory

- We can conceptualize our computer memory as a series of bytes
- When we declare any variable, an appropriately sized chunk of memory is allocated to contain the value of that variable
- We then have access to to that value through the name of the variable
- However, we know that each byte has an address
- This means that we can also access that variable by using that address
- We have been doing this when we pass-by-reference, or use any array
- Those methods of accessing by address were covered up to make the greater concept easier to understand
- Now we're diving in ourselves

A Pitfall

- The value of a pointer variable is an 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`, there will be no mixing of types to accomplish a goal

How to Declare a Pointer

How to Declare a Pointer

- 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 example above, three variables were declared
  - A pointer to an integer named `intPtr`
  - An integer named `x` with an initial value of 42
  - A pointer to an integer named `y`

Assigning an Address to a Pointer

- So now we've seen that declaring pointers isn't so scary
- How do we tell them to point to something?

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

- That does **not** work
- You will receive a compile error
- This is because you are attempting to assign the value of an integer to a pointer to an integer
- Remember, they are not the same thing
- What we need to do is assign the "address of" the integer `x` to the pointer

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

Operators for Dealing with Pointers

- The "address of" operator, &, helps us assign addresses to our pointers
- Typically, an operator in C++ has one job, and one job only
- With pointers, the * ends up serving double duty, on top of being the operator for multiplication
    - When declaring a variable, the * indicates that we are declaring a pointer to that data type
    - When using it outside of declaration with a pointer, we are *dereferencing* the pointer to get at the value being pointed to

How to Use a Pointer

Using Pointers

- We have been using pointers in disguise for a few weeks now (arrays, pass-by-reference)
- How do we actually use a pointer?
- Go over a few examples on the board to show how certain pointer assignments behave, and to illustrate the difference between the value of the pointer and the value of the data type being pointed to

A Couple Questions

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

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

- With regards to pointers, what are the two uses of the * operator?

A Couple Questions

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

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

  - Judging by the names of the variables, it appears that both variables were intended to be pointers to integers. However, attaching the * to the data type does not work like that. Only the variable `intPtr1` is a pointer to an `int`, `intPtr2` is a regular 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 of declaration, the * dereferences the pointer, meaning it goes to the address being pointed to and actually looks at the value at that address

The Real Power of Pointers

The Real Power of Pointers

- 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 types
- With pointers, we can access both

The Stack

- The first type of memory that programs can access is called the stack
- It is what we've been using so far on all of our assignments
- We will use large crates as an analogy
- A stack of crates is compact, and well organized
- To get to crates in the middle or at the bottom of the stack, we would first have to remove crates from the top
- We cannot alter a crate in the stack once it has been placed, because it will break the stack

Stack Memory

- The memory stack is very similar
- All of our functions and variables go into the stack
- There is no wasted space, which means there is no space to grow or shrink "on the fly"
- This is why we have always had to make sure that we declare how big an array is; it cannot go on the stack unless we know what its size is
- Another way we can describe this memory is static
- Once set, it cannot change

The Heap

- The second type of memory that a program can use access is called the Heap
- We are only able to access this memory through pointers
- Continuing with the crate analogy, we can imagine the heap as a room full crates, not stacked
- We can access any crate at any time we want
- We can place a crate wherever we have room for it to fit
- If a crate needs to grow or shrink, it can do so, assuming that there is room

Heap Memory

- The memory heap behaves like this
- It is not as compact, and there is generally going to be wasted space
- However, we are able to grow or shrink "on the fly"
- Array size does not have to be known at compile time, it can be allocated at run-time
- Another way we can describe this memory is dynamic
- We can get only what we need, when we need it, on demand

Accessing Heap Memory

- We need some new syntax
- The idea is that we are requesting, and subsequently borrowing heap-allocated memory
- And because it's borrowed, when we're done, we should give it back

```
int *dynInt = new int;
int *dynInt = new int(32);
```

- The first declaration simply gets a dynamically allocated integer
- The second declaration gets a dynamically allocated integer and intitializes it with a value of 32
- The keyword `new` requests memory from the heap, and it is given to use in the form of an address

Accessing Heap Memory

```
int *dynInt = new int;
```

- What is the name of the integer variable that we have allocated on the heap?

Accessing Heap Memory

```
int *dynInt = new int;
```

- What is the name of the integer variable that we have allocated on the heap?
- It doesn't have a name

Accessing Heap Memory

```
int *dynInt = new int;
```

- What is the name of the integer variable that we have allocated on the heap?
- It doesn't have a name
- Because it doesn't have a name, the **only** way we can access this dynamic integer is through the pointer
- It was mentioned that this memory is borrowed, and we should give it back when we're done
- How?

Returning Memory to the Heap

```
int *dynInt = new int;
```

- We learned some syntax to request this memory from heap, so it follows that there is some syntax for giving it back

```
delete dynInt;
```

- The keyword `delete` destroys the dynamically allocated object and returns the memory to the heap
- If we borrow memory using `new` and do not return it using `delete`, we will not receive any compiler warnings or errors
- **But we have done something bad**

The Heap is Limited

- As long as there is memory in the heap to give, you will always receive that memory
- It is possible for the heap to run out of memory
- You will see an exception thrown while the program is running, `std::bad_alloc`
- This should not be an issue in this class, but we should be aware of the possibility
- Exception handling will not be taught in this course, it will be covered in Object-Oriented Programming
-

The Heap is Passive

- The heap will never take back memory it has lent out
- So, if we do not return it and our program ends, that memory is now in a purgatory state
- It can never go back to the heap, and nothing else can access it
- This is known as a memory leak
- The *only* way for that memory to make its way back to the heap is by restarting your computer

Deleting is not Quite Enough

- Deleting the dynamically allocated memory ensures that it goes back to the heap
- But we aren't quite done yet
- What about our pointer?
- After a delete command is used, the memory it was pointing to "doesn't exist" anymore
- What would happen if we try to dereference that pointer?
- The C++ bogey-man, undefined behavior

Don't Leave Me Hangin'

- A pointer in this state is known as a dangling pointer
- It is up to us to make sure that it does not dangle

```
int *dynInt = new int;
delete dynInt;
// The next three lines are different ways to do the same thing
dynInt = NULL;      // Requires <iostream> || <cstdlib> || etc.
dynInt = 0;         // NULL is simply an alias for 0
dynInt = nullptr;   // C++11 method which is more robust
```

Good Practices for Learning to Use Dynamic Memory

- If you request heap memory using the keyword `new`, immediately write the `delete` command to return the memory
- If you delete dynamically allocated memory, immediately null the pointer so it is not dangling