Pointers

- Class #9 – Pointers
Pointers

- Pointers are among C++’s most powerful, yet most difficult concepts to master.
- We’ve seen how we can use references to do Pass-By-Reference.
- We can use Pointers to create Dynamic Data structures like Linked Lists, Stacks, Queues and Trees.
Pointers

- Pointers are a type of variable, just like int, double, etc., except instead of storing a value, they store a memory address of another variable.

- In this sense, a variable directly references a value, and a pointer indirectly references a value.
Pointers

- Pointers, just like other variables, must be declared before they can be used. For example, the declaration

  ```
  int *countPtr;
  ```

  declares a variable countPtr to be of type int * (a **pointer** to an **int** value)
  - This is read as “countPtr is a pointer to an int”.
  - Each variable being declared as a pointer must be preceded by an asterisk ( * ).
  - Also, although not required, declaring any pointer value with the name ending in Ptr is a good idea.
Initializing Pointers

- A Pointer may be initialized to 0, NULL, or an address.
- NULL is a Symbolic constant defined in `<iostream>` to represent the value 0.
- A Pointer that is assigned 0 or NULL points to nothing.
Initializing Pointers

- The **address operator**, (&), is a unary operator that returns the memory address of its operand.

- This is how we can assign a memory address to a pointer variable.

```c
int y = 5; // Declare an int, assign the value of 5
int *yPtr; // Declare a pointer to an int
yPtr = &y; // Assign yPtr the memory address of y
```
Pointer Operators

- Example

```c
int y = 5;
int *yPtr;
yPtr = &y;    // yPtr gets address of y
```

- *yPtr* “points to” *y*

```
address of y is value of yptr
```
The Dereferencing Operator

- The * operator, referred to as the indirection, or dereferencing operator, returns an alias to what the pointer is pointing to.

- In the previous example, the line
  ```
  cout <<*yPtr;   //Will Print 5
  ```
  Will print the value of the variable that *yPtr points (which is y, which is 5)

- Basically, *yPtr “returns” y

- Operations like below are also legal
  ```
  *yPtr = 7;   //Changes y to 7
  ```
Address of and Dereference

- The address of (&) and dereference (*) operators are actually inverses of each other.

- They cancel each other out

```
 *&myVar == myVar
   and
  &*yPtr == yPtr
```
```
// Fig. 5.4: fig05_04.cpp
// Using the & and * operators
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    int a;        // a is an integer
    int *aPtr;    // aPtr is a pointer to an integer

    a = 7;
    aPtr = &a;    // aPtr set to address of a

    cout << "The address of a is " << &a
         << "\nThe value of aPtr is " << aPtr;

    cout << "\n\nThe value of a is " << a
         << "\nThe value of *aPtr is " << *aPtr;

    cout << "\nShowing that * and & are inverses of "
         << "each other.\n&aPtr = " << &aPtr
         << "\n*aPtr = " << *aPtr << endl;

    return 0;
}
```

The address of `a` is the value of `aPtr`.

The `*` operator returns an alias to what its operand points to. `aPtr` points to `a`, so `*aPtr` returns `a`.

Notice how `*` and `&` are inverses.
Let’s go back old (read: previous test) topic. Passing arguments to functions.

They are two well known ways – passing by value, and passing by reference.

We can do pass by reference two different ways. Passing by Reference using references, or passing by reference using **pointers**.

You actually did this in Lab 3.
Function Passing

- So let’s think about this for a second. We’re going to pass **variables**, by reference, using **pointers**.
- Pointers hold …?
- Memory Addresses
- and what operator did we just see that will give us the memory address of a variable?
- The ampersand ( & ) operator
- This is how we call a function that uses call by reference using pointers – we have to send it the memory address, so the function call looks like

  ```
  myfunc( &my_var );  // calls function myfunc with mem
  // address of my_var
  ```
Function Passing

- So we call a function that uses Pointer arguments with the syntax of

  \[ \text{myfunction( \&my\_var );} \]

- So now we send a memory address to our function. To actually do anything with it, we need to deference it, both in the function definition and in the function body.

  \[
  \text{myfunction( int *my\_varPtr )}
  \]
  \[
  \{ 
  \quad *my\_varPtr = *my\_varPtr * *my\_varPtr;
  \}
  \]
Const Pointers

- Const Pointers, just like any const variable, is unable to be changed once it is initialized.
- Const Pointers are Pointers that always point to the same memory location.
- These must be initialized when they are declared.
- Remember, when your declaring a pointer variable, you have to declare it a type (int, double, etc.) – C++ also makes the distinction between a regular int and a const int.
So basically this leads to some screwy syntax with const pointers. Here it is. Assume x is declared as

```c
int x = 5;
```

- Non Constant Pointer to Non Constant Data
  ```c
  int *myPtr = x;
  ```

- Non Constant Pointer to Constant Data
  ```c
  const int *myPtr = &x;
  ```

- Constant Pointer to Non Constant Data
  ```c
  int *const myPtr = &x;
  ```

- Constant Pointer to Constant Data
  ```c
  const int *const Ptr = &x;
  ```
```cpp
#include <iostream>

int main()
{
    int x, y;

    int * const ptr = &x; // ptr is a constant pointer to an integer. An integer can be modified through ptr, but ptr always points to the same memory location.

    *ptr = 7;

    ptr = &y; // ptr is a constant pointer.

    return 0;
}
```

Error E2024 Fig05_13.cpp 15: Cannot modify a const object in function main()
Pointer Arithmetic

- Welcome to the world of weird programming errors.
- This is another example of the powerful but dangerous nature of pointers.
- Pointer Arithmetic is so error prone, it’s not allowed in Java or C#.
- This doesn’t mean don’t use – you may have, or at least understand it so you can understand other people’s code. So master this.
So again, what is an array really?
That’s right, a const pointer.
So, we can create a Pointer to the first element of an array with code like

```c
int b[5] = { 0 };
int *bPtr;
bPtr = b;  // above line equivalent to bPtr = &b[0];
```
Pointer Arithmetic

- Normally, when we would want to access the 4\textsuperscript{th} element in our array, we’d use notation like
  \[ b[3]; \]

  but, we can also do

  \[ (\ bPtr\ +\ 3); \ //\textit{actually\ does\ address\ of\ bPtr\ +\ 3\ *\ 4} \]

  this is called using \textbf{Pointer/ Offset notation}.

  We can also access pointers using subscripts

  \[ bPtr[3]; \ //\ \textit{same\ as\ above} \]

  this is called, you guessed it, \textbf{Pointer/ Subscript notation}. 
Thoughts on Previous

- Although you can use Pointer/Subscript notation on Pointers, and you can use Pointer/Offset notations with arrays, **try not to unless you have a good reason.**

- No technical reason, it is just confusing for people reading your code.
Null Pointers

- When you begin a program, you may have some pointers declared, but are not yet set to any value, in this case you should set them to NULL.

- NULL is a special value that indicates a pointer is unused.

- For each pointer type, there is a special value -- the "null pointer" -- which is distinguishable from all other pointer values and which is not the address of any object.

- NULL is defined as 0 (zero) in C++.
Arrays of Pointers

- Normally, we’re used to Arrays containing ints, doubles, etc.
- We can also make arrays of Pointers.
- This is most commonly seen with Arrays of C-Style strings.
Array of Pointers

```c
const char* suit =
    {"Clubs","Diamonds","Hearts","Spades"}
```

- This basically says “each element is of type pointer to char”
Why Null Pointers?

- When we declare (but not initialize) *ANY* variable, what value does it contain?
- What can’t we do to a variable if we have no idea what it contains?
- Null Pointers give us a way to compare and see if something is initialized.
Comparing Pointers

To test if a Pointer is null, you can either by either

```c
int *intPtr=NULL;

if (intPtr==NULL)
    or
if (intPtr=='0')
```

These both are equivalent. I like the first convention (more readable), but either is acceptable.
## Arrays of Pointers

Arrays can contain pointers

- Each element of not in the array, only pointers to the strings are in the array
- suit is a pointer to a `char *` (a string)
- The strings are

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="suit[0] diagram" /></td>
<td><img src="image" alt="suit[1] diagram" /></td>
<td><img src="image" alt="suit[2] diagram" /></td>
<td><img src="image" alt="suit[3] diagram" /></td>
</tr>
</tbody>
</table>

- suit array has a fixed size, but strings can be of any size