Groups 12 Members:
Brent Irwin
Nhan Ngo
Jason Silber
Peter Moldenhauer

Group Activity 2: Pointers
Oct 23 by 11:59pm

1. **Illustrate a pointer variable.** *Remember that a pointer variable is a variable that holds a memory address, not a data value.*

The pointer variable holds an address to a certain data type. It does not hold the data value.

```
int x = 25;         // set x to 25
int *ptr;           // initialize pointer variable to an integer
ptr = &x;           // set pointer to the address of x, which is 0x7e00
```

This pointer ptr holds the value 0x7e00, which is where x is located.

ptr	0x7e00

x	25

To find the value stored at the pointer's location, you dereference the pointer variable with an asterisk. *ptr would then be equal to 25.

*ptr	25

```
x = 30;             // change x to 30
```

If the value of x is changed, its location stays the same. ptr still points to the same location, so the updated value of x will be reflected when ptr is dereferenced again.

`ptr = 0x7e00`

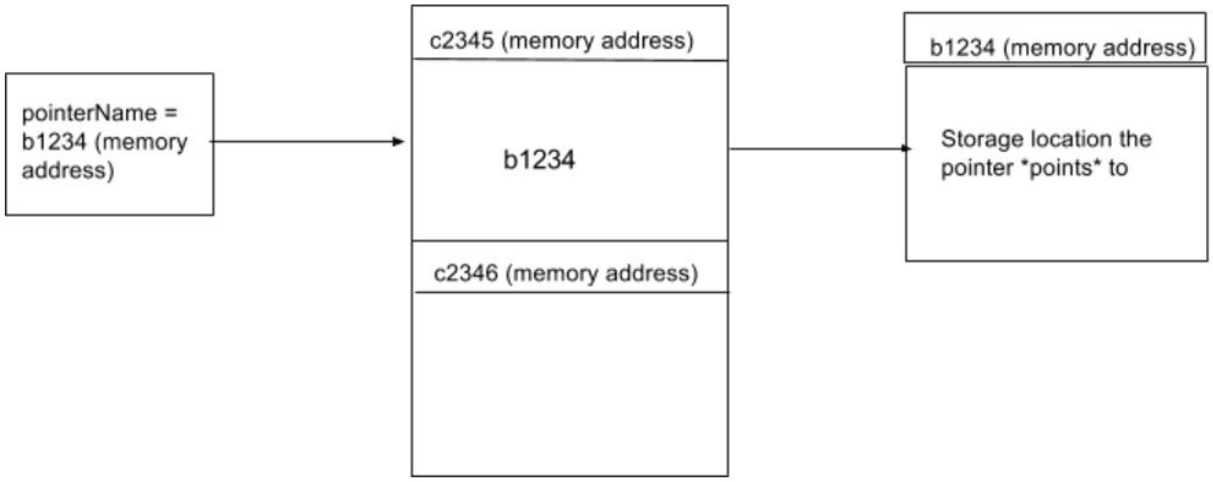
ptr	0x7e00

x	30

*ptr	30

2. **Illustrate a pointer to a storage location.** *A pointer is a variable, so it also has a name, but it holds a memory address, not a value of a standard data type.*

A pointer is a variable of the memory address type with a memory address of its own. This is where it stores the address the pointer *points* to. In the diagram below, the pointer variable pointerName represents the memory address at c2345, and at this memory address, it is also storing the memory address, b1234. Thus, pointerName is a pointer variable because it is pointing to some value stored in memory b1234.



3. **Illustrate a pointer to a pointer.** *Similar to #2, but there is an additional pointer variable that have a name.*

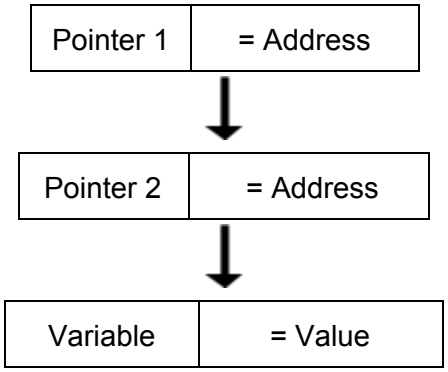
A pointer to a pointer relationship can be a complex relationship in the sense that it is actually 2 reserved memory locations that is used to store addresses in both pointers.

A pointer to a pointer is defined as such examples:

char** prt; int** prt; double** prt;

The first pointer reserves memory at a given address for the given type. The second pointer also reserves memory at a different memory location for address storage. Importantly, in this example, the second pointer is used to store the address of the variable initialized to it, while in contrast the first pointer is used to store the address of the second pointer.

The connection between them is such that it representative of a chain of pointers where the first pointer pointer is chained to the second pointer and the second pointer is chained to the variable.



A pointer must dereferenced to access the contents of the of the variable address that was passed to it. A pointer to pointer relationship must be accessed in similar way with a dereferencing at each level of pointer depth. The order intuitive in the alignment to the declaration of the pointer to pointer.

int pointer dereferenced example**	**Pointer pointed to dereferenced example**
**pointer	*pointer

An example of declaration, initialization and assignment in syntax is as follows in a standard pointer to pointer relationship:

Single Pointer Example	Pointer to Pointer Example

int* pointer;
Int exampleValue = 0;
pointer = &exampleValue;
cout << exampleValue; // outputs 0
cout << *pointer; // outputs 0
	int** pointer;
	Int exampleValue = 0;
	pointer = &exampleValue;
	pointer = &pointer;
	cout << exampleValue; // outputs 0
	cout << *pointer; // outputs 0
	cout << **pointer; // outputs 0

A Pointer to pointer is particularly useful for dynamically created 2d array's. They allow for the first pointer to accept the address of a newly created array. The second pointer can then accept the next depth of newly created array forming a 2d array. A very important item to be wary of with pointer to pointer dynamically created types is they require careful attention to the deletion of the type to free up the memory. An example of both processes is listed below.

Pointer to Pointer Dynamically Created 2d Array Example	Pointer to Pointer Dynamically Created 2d Array Deletion Example

char** dynamic2dArray;
 	
dynamic2dArray = new char*[arrayTotalSize];
 	
for(int counter = 0; counter < arrayTotalSize; ++counter)
dynamic2dArray[counter] = new char[arrayTotalSize];
 	
for(int row = 0; row < arrayTotalSize; row++)
{
for(int column = 0; column < arrayTotalSize; column++)
{
dynamic2dArray[row][column] = variable;
}
}
	for(int counter = 0; counter < Size; ++counter)
	delete [] dynamic2dArray[counter];
	
	delete [] dynamic2dArray;
	dynamic2dArray = 0;

This illustrates the complexities of stepping through the different levels of a pointer to pointer relationship when assigned addresses of more complex things like 2d array's and maintaining an understanding of where and how deep you are in the chain of a pointer to pointer.

4. **Illustrate an array with 3 elements.** *Arrays in C/C++ are implemented with pointers, but you can still use the bracket notation. The body of the array will still be contiguous storage locations. Use both the bracket notation and a pointer to indicate that array.*

Pointers and arrays have a unique relationship that can be easily misunderstood. When we look at an array in detail, we can see that the name of the array (without brackets or subscripts) is actually the starting address of the array. In other words, the array name by itself is actually a pointer. This is important to know because this allows the contents of the array to be accessed either through the bracket notation (array name with brackets and subscripts) OR the array contents can be accessed through pointer notation (dereferencing pointer and using pointer arithmetic). Let's look at examples of both of these methods to get a better understanding of how this works:

First, let's look at accessing an array and the contents of that array in bracket notation. Consider an array that is defined as follows:

```
int myArray[ ] = {3, 5, 7};
```

We know that this is an array of three integers, specifically the integers 3, 5 and 7. To access each individual element of this array through standard array notation or "bracket notation", we need to use the array name with the square brackets and the specific subscript. Since the elements in arrays start at the index 0 and then increment up from 0, we know that the subscript for the integer 3 in myArray is 0, the subscript for integer 5 in myArray is 1 and the subscript for the integer 7 in myArray is 2. Therefore, to access the second element (5) in myArray we need to use the array name with the brackets and the specific subscript inside of the brackets to get the desired array element:

```
cout << myArray[1];  // this will display 5
```

Now let's look at how we can use a pointer to indicate an array and access the array contents. As mentioned above, an array name by itself is actually a pointer. So using the array demonstrated above, we can see the address of the array if we do:

```
cout << myArray;
```

When this line is run, it will display something like 0x4a00 (or of similar nature) which is the starting address of myArray. Because the starting address of myArray contains the first element of the array (3), to access the 3 we need to dereference the myArray pointer. Therefore, if we do:

```
cout << *myArray;
```

This will display 3 because we are dereferencing the array name pointer. Now, to access the elements beyond just the first element in the array, we need to use pointer arithmetic. It is

important to know that the body of the array will still be contiguous storage locations even with pointer notation. Since *myArray displays the first element in the array, if we add 1 to *myArray, then the second element (5) will be displayed. It must be noted though that due to operator precedence, we need to make sure that the pointer is added and THEN dereferenced. So it should follow the format of:

```
cout << *(myArray + 1);  // this will display 5
```

Similarly, to access the last element of the array (7) we would need to add 2 to the myArray pointer because this last element is located at index 2 of the array:

```
cout << *(myArray + 2);  // this will display 7
```

5. **Illustrate a two-dimensional array with 2 rows of 3 elements in each row.** *Multi-dimensional arrays in C/C++ are arrays of arrays. Consider a mxn array. There are m rows of n elements. In this case there are m locations in memory that consist of n contiguous memory locations. The m sets are not required to be contiguous with each other! Use both the bracket notation and pointers to indicate that array.*

A two dimensional array is basically one array that contains an array as each one of its elements. This two dimensional array then forms a “table” of rows and columns of all of the different elements in the array. A two dimensional array with 2 rows of 3 elements in each row can be defined using bracket notation as follows:

```
int my2DArray[2][3] = {{1,2,3},{4,5,6}};
```

Here, we can see that the array of rows (2) is created and that each of “row elements” contain an array itself of 3 elements. The first array is rows and the second array is the columns in each row. This 2D array can be represented visually as seen in the table below. Notice how just like in a 1D array, in a 2D array, each specific element of the array can be accessed through the subscripts. In a 2D array however, we will not need two subscripts to access both the row and the column “coordinates” in the array:

my2DArray[0][0]	my2DArray[0][1]	my2DArray[0][2]
my2DArray[1][0]	my2DArray[1][1]	my2DArray[1][2]

Two dimensional arrays just like one dimensional arrays can also be indicated through pointers and can use pointers to access each specific element of the two dimensional array. It is important to know that a two dimensional array is defined with a pointer as only a single pointer (not a pointer to a pointer). This is because the data encoded in the array is sequential.

Therefore, the array used above:

```
int my2DArray[2][3] = {{1,2,3},{4,5,6}};
```

can also be written as a pointer as:

```
int *my2DArrayPointer = (int*)my2DArray;
```

To access the specific elements of the two dimensional array through a pointer is a little unique. In general, the element at my2DArray[x][y] can be accessed using pointer arithmetic by:

```
my2DArrayPointer + x*ySize + y
```

where ySize is the size of the first array dimension or number of columns (in this case 3). Therefore, to access the last element in the 2D array (6) we need to substitute in the x and y "coordinates" into the line of code above. We then get:

```
cout << *(my2DArrayPointer + 1*3 + 2);
```

Note the fact that we need to actually dereference the pointer to access the contents. If we do not dereference the pointer, we will simply get the address at which the element is located at.

6. **Illustrate a linked list with 3 nodes.** *Please draw the memory diagram. Later in this and later courses you will use links. A link is a node with a pointer variable to another node. Think of it like a train. You have a node structure which holds data and a pointer variable "next", indicating the memory location of the next node in the link list. The last node will have a NULL pointer. You also need a pointer variable to the first node of the linked structure.*