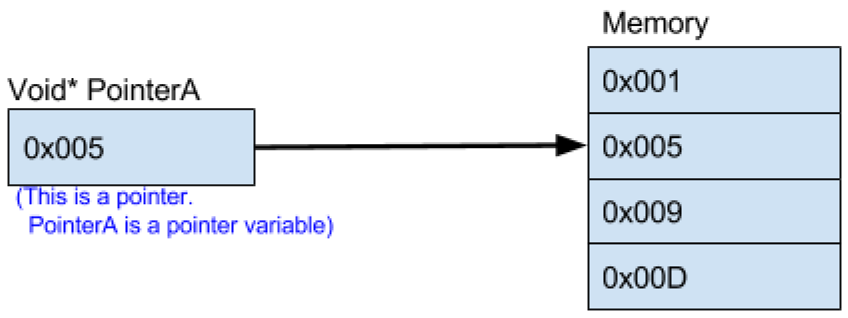
Group 3 (Jennifer Gibson, Benjamin Vacariu, Ju An, Benjamin Day, Ramiro Cabrera)

<u>Pointers</u>

1. Illustrate a pointer variable

(Reference: Stackoverflow, Textbook)



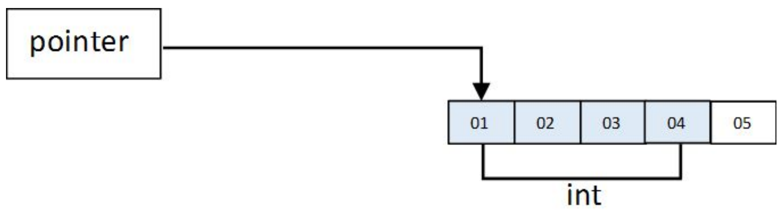
A pointer is a value and pointer variable is a variable where a value is stored in. A pointer variable has a name and it holds a memory address, thus pointing to some sort of data stored in the address.

An example:

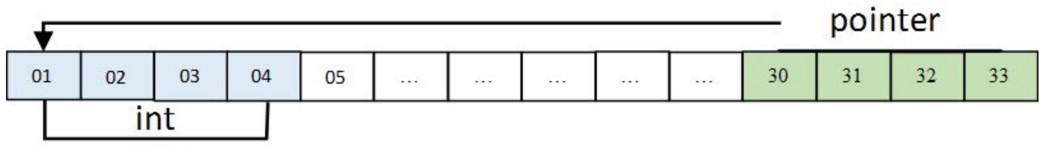
int *ptr; //creates a pointer variable
int num = 5; //regular variable that holds the number 5
ptr = &num; //now the pointer variable holds the address of num. In other words, it points to the data in num.
cout << *ptr; // Outputs the value stored at the address the pointer points to, in this case 5
cout << ptr; // Outputs the address pointed to by the pointer

2. Illustrate a pointer to a storage location]

When a variable is modified, it's storage is modified accordingly. A pointer, on the other hand, represents the starting address and size of a variable in memory. A pointer, as the name implies, points to a specific location in memory that a variable resides in. To point to another variable, the space in memory associated with a pointer holds the address of the variable that the pointer points to (unlike a variable whose memory holds the actual value of the variable).



To point to another variable, the space in memory associated with a pointer holds the address of the variable that the pointer points to (unlike a variable whose memory holds the actual value of the variable).



Just like variables, pointers must have a specific data type to determine how much space in memory the variable it points to occupies. When a pointer points to an integer, it must be declared as an integer pointer. Unlike an integer variable, basic mathematical operations on a pointer alter where the pointer points to in memory. For example, adding 1 to an integer pointer, alters where the pointer points to in memory by the size of the pointer type (so +4 bytes for an integer pointer).

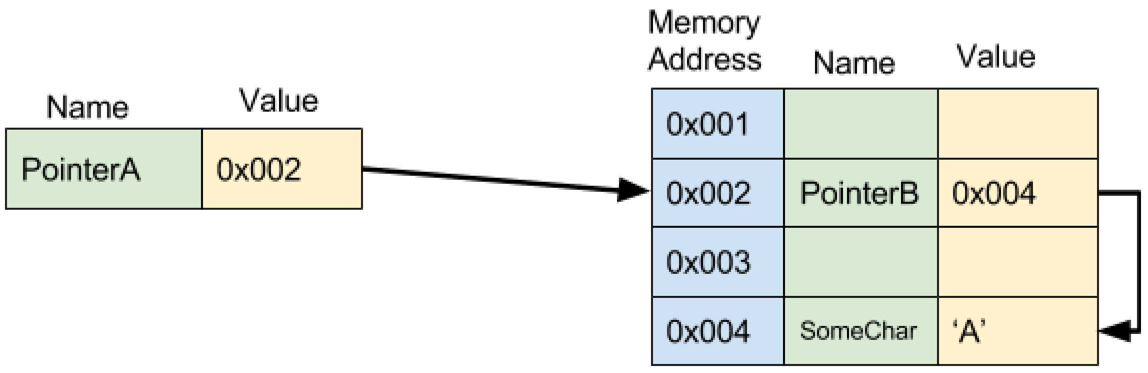
To use a pointer to access the variable value that a pointer points to, the dereference operator '*' is used. For example, to add 1 to the value of the variable the pointer points to, *pointer += 1 can be used (but again without dereferencing it first this would only alter where the pointer points to in memory).

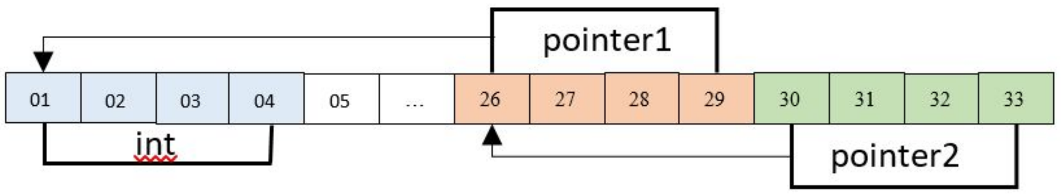
An example:

```
int *ptr;
int num = 5;
ptr = &num;
cout << ptr;  //displays the address not a value - the address in pointer is the address of num.
cout << &num;  // displays the same as above.  it displays the address, not the value.
```

3. Illustrate a Pointer to a Pointer

The pointer refers to a memory address that stores a memory address of another variable.





A pointer points to an object at some memory address and is written type* name or type *name. If the object that it points to happens to be a pointer the declaration will be type** name or type **name.

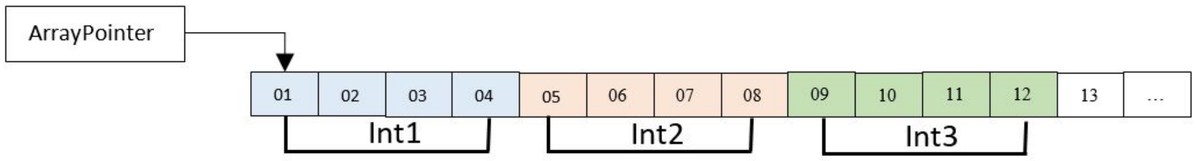
The pointer to a pointer stores the memory address of another pointer variable that stores the memory address of the object or variable.

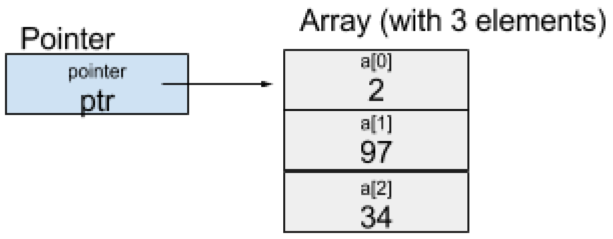
Dereferencing a pointer accesses the value at the address the pointer points to. Dereferencing a pointer to a pointer returns the address the first pointer points to. The following example helps make more sense of this:

```
int *ptr1;
int **ptr2;
int num = 5;
ptr1 = &num // now ptr1 points to num which holds a value 5
ptr2 = &ptr1 // now ptr2 points to ptr1 which points to num whose value is 5;
```

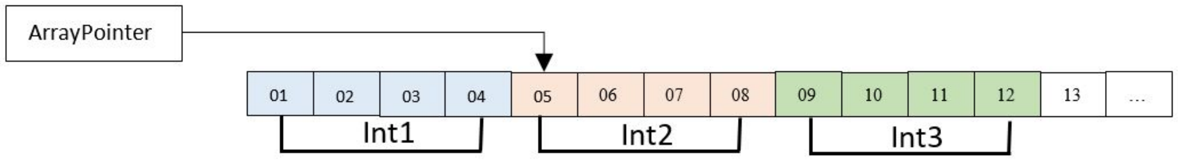
4. Illustrate an array with 3 elements

An array is a pointer to a collection of similar data types all stored within contiguous memory. For example, the diagrams below illustrate an array of 3 integers:





Just like a normal pointer, incrementing an array pointer increments where the pointer points to by the number of bytes corresponding to the pointer type. So adding 1 to ArrayPointer results in:

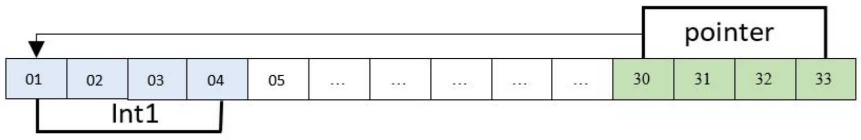


Pointers can also be manipulated using a method called indexing. We begin at the address of 01 again (see first diagram). Now, to access the value stored by Int1, we can dereference our pointer *ArrayPointer to manipulate the value (not address) of Int1. Another way to accomplish this is with using indexes. When ArrayPointer points to Int1, we can also access Int1's value using the following notation: ArrayPointer[0]. The benefit of using index notation is that the address that ArrayPointer points to is not changed. So to access the value of the second element in our array (Int2) we can just use ArrayPointer[1]. Again using ArrayPointer[1] doesn't alter the address that ArrayPointer points to, it just gives the value of the memory block offset by ArrayPointer's address plus the index amount. Another way to think about this is with the following equations.

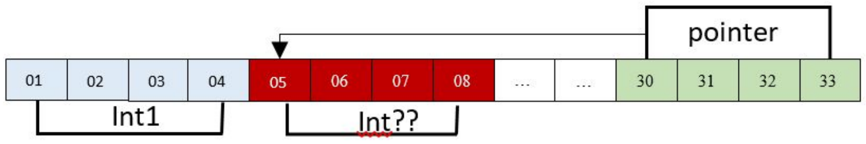
*ArrayPointer = ArrayPointer[0] = Int1's Value
*(ArrayPointer+=1) = ArrayPointer[0+1] = Int2's Value
*(ArrayPointer+=2) = ArrayPointer[0+2] = Int3's Value

The problem with the equations above is that when we increment the ArrayPointer (not using index notation) we are changing where the ArrayPointer points to. So if we take ArrayPointer +=1, then ask for *ArrayPointer we get the value of Int2, but to go back to Int1's value we'd have to move the pointer again (ArrayPointer -= 1). Whereas using index notation we can just say Int1's value = ArrayPointer[0], and Int3's value = ArrayPointer[2], Int2's value = ArrayPointer[1], and each time we don't have to mess with where ArrayPointer is actually pointing at to access the value.

Another thing to note is that if we have a normal integer pointer pointing to an int as follows:

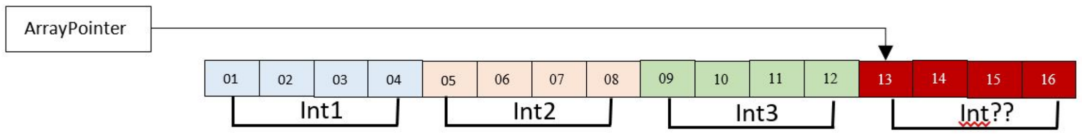


We can access the value of int via *pointer, but we can also access its value via pointer[0].



Even if this new memory location does not store an integer the pointer will create an integer value from the data that is stored there and return it (or manipulate it if that's what you're doing).

ArrayPointer[0], ArrayPointer[1], ArrayPointer[2] correspond to the values of Int1, Int2, and Int3. What does the value of ArrayPointer[3] correspond to?



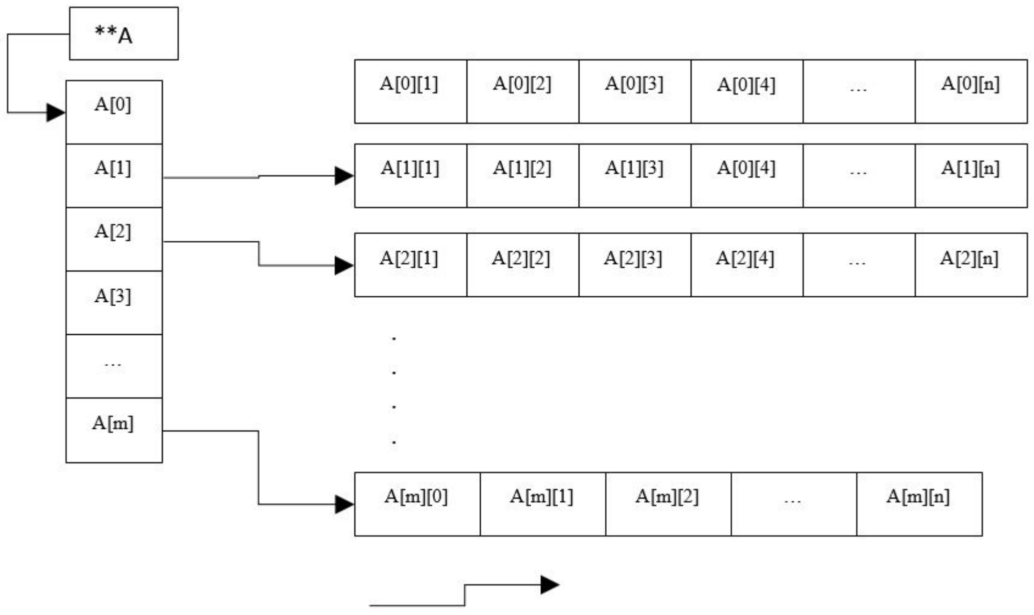
So with arrays knowing the size of the array is critical so a value outside of the array boundary is not accidentally used. This is especially dangerous, because what if that chunk of memory that you're manipulating is being used by the system for something else? It is possible to easily corrupt other programs or cause the system to crash if you're not careful.

An example:

```
int array[3] = {2,97,34};  // create and initialize array
int *ptr;  //declare pointer
ptr = array;  //assign array address (which is the address of array[0]) to ptr
cout << *(ptr +2);  // prints 34
cout << *(array+1); //prints 97
cout << array[0]; // prints 2
```

5. Illustrate a two-dimensional array with 2 rows and 3 elements in each row.

2D array is a 1-Dimensional array that we're used to, where every element is a pointer to another 1D array. For example:

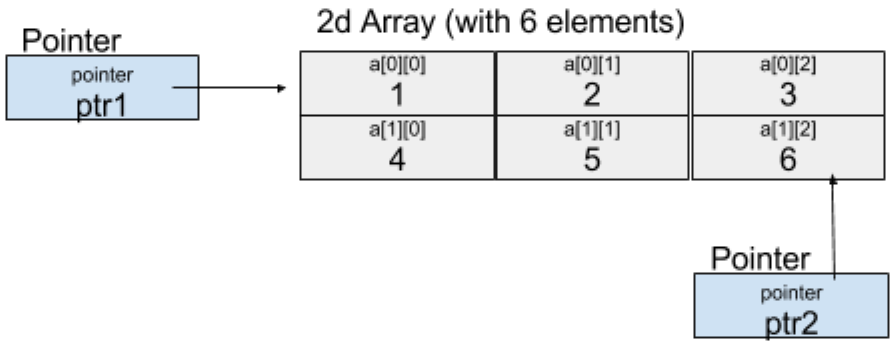


**A
A[0]
A[1]
A[2]
A[3]
...
A[m]
A[0][1] A[0][2] A[0][3] A[0][4] ... A[0][n]
A[1][1] A[1][2] A[1][3] A[0][4] ... A[1][n]
A[2][1] A[2][2] A[2][3] A[2][4] ... A[2][n]
A[m][0] A[m][1] A[m][2] ... A[m][n]

Since *ptr can point to a single integer or an array of integers, then **ptr points to another pointer or an array of pointers. Also, just like a pointer we can use index notation to access values. For a single pointer *ptr we can use ptr[0] to access the first element value. So what can we expect for a two dimensional array? Well as can be guessed from the above diagram, two indices can be used to access a specific value. So if **A is a pointer to a 2D array, we can use A[0][0] to get the value at the first element. Or likewise, we can use A[0] to get the pointer to the first 1D array.

Just like with regular pointers we can use pointer manipulations with double pointers. If A points to A[0], then A+=1 points to A[1].

An example:



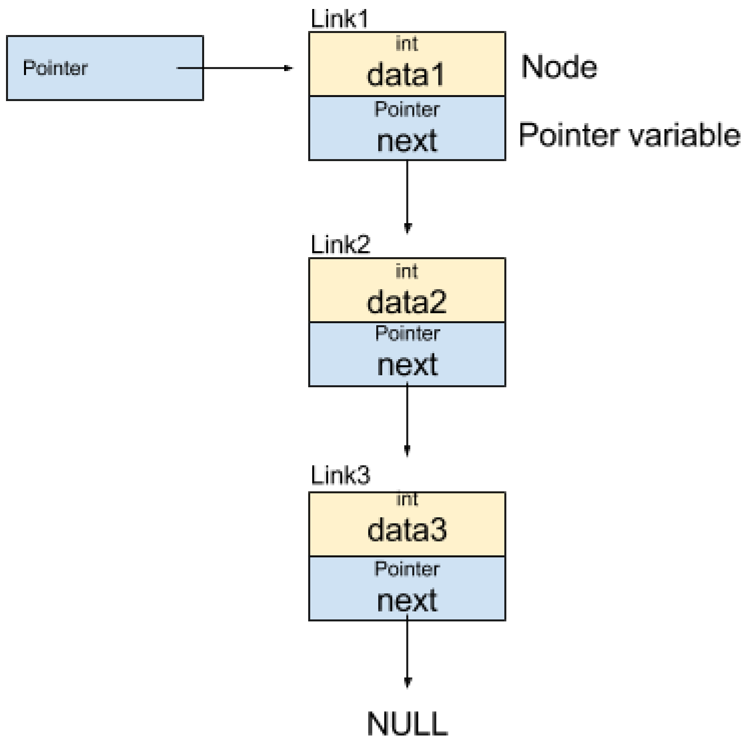
```
int array2D[2][3] = {
   {1, 2, 3},
   {4, 5, 6}
};
int *ptr1 = *array2D;         // Points to the array at it's starting address.
int *ptr2 = &array2D[1][2];   // Pointing to a specific element of the array.

cout << array2D << endl;     // Prints the array's starting address
cout << ptr1 << endl;        // Prints the array's starting address
cout << *ptr1 << endl;       // Prints the value stored at array2D[0][0] in this case 6
cout << ptr2 << endl;        // Prints the address of the element array2D[1][2]
cout << *ptr2 << endl;       // Prints the value stored at array2D[1][2] in this case 6
cout << *(ptr1 + 3) << endl; // Prints the value stored at array2D[1][0] in this case 4

// The last line demonstrates that the values in a 2d array are stored consecutively
// As such, a 2d array is not too different from a 1d array with the same number of elements.
// They both store values the same way, but the 2d array is indexed and can be easier to use in some
// cases.
```

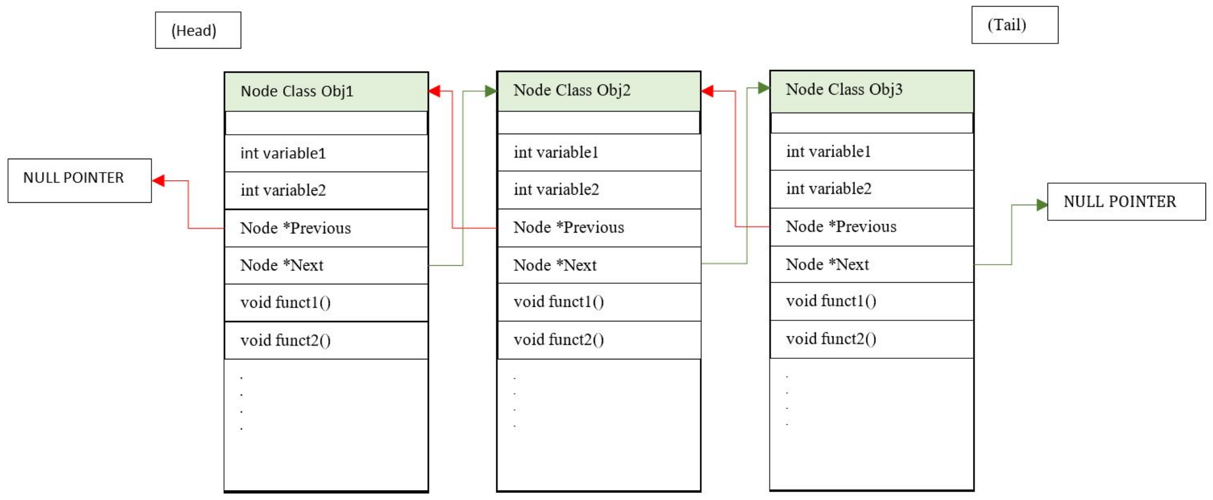
6. Illustrate a linked list with 3 nodes.
(Reference: Chapter 17 of C++ Early Objects 8th edition.)

A linked list is a list constructed using pointers. In other words, it is a collection of structures that contain a pointer to one or more structures of the same type forming a chain (or other possible topologies). The simplest linked list is similar to a 1 dimensional array, the first element in the linked list is called the "head" and the last element is called the "tail". The boxes are called nodes and the arrows represent pointers. Each of the nodes below contains a data and a pointer variable "next". In order to access the nodes, you have to have a pointer to the head of the list. Then each node has a successor pointer pointing to the next node in the list.



In a one-way linked list each structure is linked such that each structure has a single pointer that points to the next structure in the chain. Traversal of a one-way linked list can only be done in one direction (from the head to the tail object). In a two-way linked list, each structure has two pointers, one that points to the next object and another pointer that points to the previous object. Traversal in a two way linked list can be done in both directions (head to tail, or tail to head object).

The example below is a two way linked list (using classes). Each structure contains a Next pointer that points to the next object in the list, the last object in the list points to NULL to signify that it's the last component in the list. The ‘Previous’ pointer points to the structure prior to itself within the list. If the object is the head of the list, the ‘Previous’ pointer points to NULL.



To traverse the above example starting at Obj1:

```
Node *tmp = &Obj1;
while(tmp != NULL)
{
        tmp = tmp->Next;
}
```

or to go backwards:

```
Node *tmp = &Obj3
while(tmp != NULL)
{
        tmp = tmp->Previous
}
```

Also note that linked lists are not stored contiguously in memory. Linked lists provide the benefit over other standard containers (like vectors) is that the programmer can control how much memory is being allocated to the list as it grows and shrinks. Linked lists also provide the ability to add an item to the middle of the list without having to shift memory around (just change where the pointers point to). A downside to linked lists is that they add a layer of complexity that standard containers don’t have. Another downside is that standard containers often have a lot of pre-built functionality that would have to be manually re-created for a linked list.