CSCI-2270-Data Structures
Instructor-Hoenigman/Jacobson
Recitation 4

In this recitation, you will learn about pointer declaration, passing arrays in and out of functions, and an introduction to building a linked list. First, let's review the concepts of pointers.

Pointers:

Pointers are similar to normal variables as they can store data. When the variable is declared, memory is allocated to that variable at a specific location, but your program doesn't have to worry about the physical address. For a pointer variable, the value stored at that address is the address of another location in memory.

Declaring pointers:

```
type*  ptr;
type  *  ptr;
type  *ptr;       // these declarations all have the same meaning
```

Initializing pointers using the address-of operator(&):

When a pointer is declared, it does not refer to a valid address unless it is initialized. Therefore, it is important to initialize pointers. This is done using the address-of operator (&) or the new keyword. The & operator returns the address of a variable. The new keyword allocates memory on the heap and returns that heap address.

```
int x = 100;
int *ptr = &x;    // assign the pointer to point to a valid address, the address of x.

Int *x = new int; //x stores the address of the memory on the heap
```

Accessing the address and data in a pointer:

In order to access the value/data stored in a pointer we use the dereferencing operator - (*). Remember that *ptr* stores the address location that holds the content and **ptr* refers to the content itself at that location.

```
cout << ptr << endl;          //prints the address
cout << *ptr << endl;         //prints the value
```

Reference Variables:

As seen above, when *&* is used in an expression, it behaves like an address-of operator and returns the address of the variable on which the operator is used. However, C++ attaches an additional meaning to the *&* operator. It can be used to create reference variables when used in declaration statements. In such cases, it acts as an alias to an already existing variable.

```
type &reference_variable = existing_variable;
cout << reference_variable << endl;          //prints the value
cout << &reference_variable << endl;                 //prints the address
```

Examples:

```
int var = 5;
int *p1 = &var;                 // p1 points to the address of var1

cout << var << endl;            // prints the value 5
cout << &var << endl;           // prints the address: for example, 0x7fff5b2fcb18
cout << &p1 << endl;            // prints the address: 0x7fff5b2fcb08
cout << *p1 << endl;            // prints the value 5
cout << p1 << endl;             // prints the address: 0x7fff5b2fcb18
```

Passing pointers in/out of functions:

Pointers are especially useful in situations where we want to modify the original copy of the variable directly. Recall that by default C++ uses the pass by value mechanism-when normal variables are passed to a function a clone copy is passed as an argument and any changes made to it inside the function do not reflect on the original copy outside the function.

When pointers are passed to a function it is called **pass-by-reference**

Pass by reference using pointer variables as arguments:

```
void AddTwo(int *p)
{
        *p=*p+2;

}
int main()
{
        int num =100;
        int *v = &num;
        int *ex = AddTwo(v);   // calls the AddTwo function by passing the address of the variable
        cout << ex << endl;    // prints the address of p
        // AddTwo(&num)        // same as the above call
}
```

Pass-by-reference using reference variables as arguments

```
void AddTwo(int &p)
{
        p=p+2;

}
```

```
int main()
{
        int num = 100;
        AddTwo(num);
        cout<<num<<endl;
}
```

Arrays as Pointers

An array's name is a pointer that point to the first element of the array, i.e, the element at index 0. Suppose that the array NumArr is an int array, then NumArr is a pointer that points to the first element of the array and is equivalent to &NumArr[0].

For example:

```
int a[2] ={10,11};
cout << a << endl;              //print to the address of the first element in the array
```

To dereference the name of the array we add the dereferencing operator '*':

```
cout << *a << endl;              //points to the value of the first element in the array

cout << a[0] << endl;           // prints the value at 0th index of a
cout << &a[0] << endl;                  // prints the address of 0th index of a
```

Could this work ?

```
cout << *a[0] << endl;
```

No, a[0] is already accessing the value stored in the array at index 0. Remember that the dereferencing operator is used to access the value.

Function Definitions

An array is passed as a pointer into a function. This can be done in the following ways:

```
int MaxElement(int NumArr[30], int size)

int MaxElement(int * NumArr, int size)

int MaxElement(int NumArr[], int size)
```

All of these above declarations are equal and are recognized by the compiler as:

```
int MaxElement(int*, int)
```

The array is always passed as a pointer to index 0 of the array and therefore we don't need to pass the size of the array in the function. Any value given inside the square brackets is ignored

by the compiler.

Any modifications made to the values in the array inside of a function are reflected outside of the function as well. This is because the array is passed to the function as a reference to the memory location of the array instead of a clone copy.

Returning Arrays from a Function

Arrays are returned as pointers, similar to how they are passed into a function. In order to indicate that the function returns a pointer to an array, we use the * operator in the function definition.

```
int* AddOne(int *arr, int n){
        for(int i = 0;i<n;i++){
                arr[i]++;
        }
        return arr;
}
```

OR

```
int* AddOne(int arr[],int n){
        for(int i = 0;i<n;i++){
                arr[i]++;
        }
        return arr;
}
```

OR

```
int* AddOne(int arr[5],int n){
        for(int i = 0;i<n;i++){
                arr[i]++;
        }
        return arr;
}
```

***Calling the function*:**

```
Int *b=AddOne(arr,5);
```

Linked Lists

One of the limitations of arrays is that they have a fixed size. As studied in the previous recitation, allocating memory to store additional data, once the array is full, can be addressed

with an array-doubling algorithm. Even if only one or two additional elements need to be added, array doubling still allocates double the memory. This can be computationally expensive.

A list is a data structure that allows for individual elements to be added and removed as needed. In a typical list implementation, called a linked list, memory is allocated for individual elements, and then pointers link those individual elements together.

Singly Linked Lists

There are two types of linked lists, singly linked lists and doubly linked lists. Today, we will talk about singly linked lists. In a singly linked list, each element, also called a node, contains the *data* stored in the node and *a pointer* to the next node in the list (shown in Figure 1).

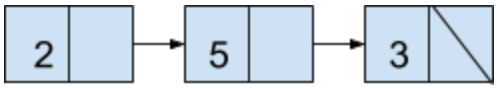


Figure 1. Singly linked list with three elements, called nodes. In this example, each node has an integer key value and a pointer to the next node in the list.

In the Figure 1 example, the node *data is the integer key*. The first node has a key value of 2, the second node has a key value of 5 and the third node has a key value of 3. The next pointer for the final node in the list is set to NULL, which is shown by the slanted line.

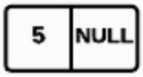
All variables stored in memory have a memory location that can be accessed using a pointer variable. A linked list node can be implemented in C++ using a class or a struct and the next and previous pointers in the node reference another instance of the node.

```
//node implementation for singly linked list
struct node{
        int key;
        node *next;
};
```

Building a Singly Linked List

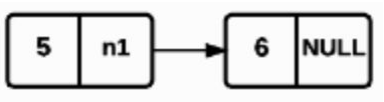
```
node *x = new node          //Create a new node dynamically
x->key = 5                          // set the key to point to the first node
x->next = NULL                      // set next pointer to null (list is empty)
node *head = x                      // store new node as the head of LL and point to x
```

This makes the linked list look like:



```
node *n1 = new node          // Create a new node to add to the list
n1->key = 6
n1->next = NULL
x->next = n1                                    // set x.next pointer to the next node
```

Now the linked looks like:



Try to add the next node to the list. What will the outcome look like?

In class work:

For the in class work you will build a singly linked list using the values 2, 5, 3. You will write the pseudocode for *addNode* function, which adds the node to the end of a linked list. The function takes two arguments, the first node in the list, also called the head of the list, and the value to be added. You will show your TA the algorithm for building a linked list and a drawing of what the linked list looks like at each step and the end result of the linked list.

Recitation 4 Programming Assignment

There is a link on Moodle to a Recitation 4 Programming exercise. In the quiz, you are asked to implement your *addNode* function. You have until Sunday at 5pm to complete the exercise.