CSCI-2270 Data Structures Recitation 2 Instructors: Hoenigman/Zagrodzki/Zietz

January 2018

Learning Objectives:

- Pointers
- Referencing and de-referencing
- Pointer to structures
- call by value vs. call by reference

1 Pointers

Every variable we declare in our code grabs a space in the memory. So it has an address in memory where it resides. A pointer is a memory address. By means of pointer variable we can access the address and the value at that address.

2 Address-of operator

Consider the following lines of code-

```
        int main()
        {
            int a = 10;
            cout<< a <<endl;
            cout<< &a << endl;
            return 0;
        }
```

Output

```
10
0x7ffccbbcd804
```

So the first `cout` is quite straight forward. It is simply printing the value of *a*. But what is being printed by the second `cout`. It is the address of the variable *a*. The operator '&' is used to get the address of the variable *a*. Hence it is *referencing* operator or *address-of* operator.

3 Dereference operator

Consider the following lines of code-

```
        int main()
        {
            int a = 10;
            int* p = &a; // * here denote p is  pointer variable
            cout<< a <<endl;
            cout<< p << endl;
            cout<< *p << endl; //* is used to dereference p
            return 0;
        }
```

Output

```
10
0x7ffccbbcd804
10
```

This code is a bit trickier than the previous one. Let's see what is happening in the code. Note now we are storing the address-of *a* in another variable *p*. Note the type of *p*. It is *int** instead of just *int*. So int* p suggest that p is a **pointer variable**(* denote it as pointer) and it will store the address of a **int** variable.

So the second `cout` prints *p*, the address of variable *a*. Now what the third `cout` is doing? Remember *p* has the address of *a*. By puting a '*' before *p* we are accessing the value stored at the position addressed by *p*. Hence * is dereferencing the address *p*.

4 Pointer to struct

We can have address (pointers) to any type of variables even for structures.

```
    struct Distance
    {
        int feet;
        int inch;
    }
    int main()
    {
        Distance d;
        Distance* ptr; // declare a pointer to Distance variable
        d.feet = 8;
        d.inch=6;

        ptr = &d; //store the address of d in p
        cout<<"Distance="<< ptr->feet << "ft" << ptr->inch << "inches";
        return 0;
    }
```

Output
Why don't you try this one yourselves?
You may be wandering about '$->$'. Recall to access members of a struct variable we use '.' operator (e.g. `d.foot = 8`). When we have a pointer to a struct variable to use the member variables we will use $->$ operator.

5 Call by Value vs. Call by address

Consider the following code-

```
    void add2(int num)
    {
        num = num + 2;
    }
    int main()
    {
        int a = 10;
        add2(a);
        cout<< a;
    }
```

What do you think the output will be? 12?
To your surprise it will be just 10. When we pass a variable as argument to a function in the system

stack the function creates the local copy of the variable and performs the operation on that local copy. The caller function has no knowledge of that local copy. Hence the change is not persisted. Now consider this version-

```
        void add2(int* num)
        {
            *num = *num + 2;
        }
        int main()
        {
            int a = 10;
            add2( &a );
            cout<< a;
        }
```

In this case we are passing the address of a. The function will again create a local copy of the address. However since both of this addresses are same they will refer to to the variable *a*. Hence changing the value at the pointer will change the value of *a* and that change will be persisted. **Now examine the following code and try to find why it is persisting the change?**

```
        void add2(int a[], int len)
        {
            for(int i=0;i<len;i++)
                a[i]+= 2;
        }
        int main()
        {
            int a[] = {1,2,3};
            add2( a, 3 );
            for(int i=0;i<3;i++)
                cout<< a[i] << endl;
        }
```

Is this one similar/different to the one given below?

```
        void add2(int* a, int len)
        {
            for(int i=0;i<len;i++)
                a[i]+= 2;
        }
        int main()
        {
            int a[] = {1,2,3};
            add2( &a[0], 3 );
            for(int i=0;i<3;i++)
                cout<< a[i] << endl;
        }
```

6 new and delete

Non-static and local variable get memory allocated in stack. Dynamically allocated memory is allocated in heap. We will use **new** operator to dynamically allocate memory. For example-

```
        int* a = new int[10];
```

will dynamically allocates memory for 10 integers continuously.
Advantage We can take the size of the array as user input during run time and dynamically create an array. **Cons** Once programmer allocates memory dynamically it is his/her responsibility to deallocate it. Otherwise even if the array is no longer required the space will be occupied. That will cause a memory leak. To delete a space pointed by a pointer **p** we need to use-

```
        delete p;
```

If **p** points to an array then to free the space-

```
        delete[] p;
```

6.1 A small quiz

1. What is the difference between `int* a = new int( 25 )` and `int* a = new int[ 25 ]`?

2. How come we can pass an array name as an argument to a function and still be able to persist the change?

3. In the exercise given in next section can you think of a situation (or in absence of which line) when there will be a memory leak?

7 Exercise

Write down c++ program that will do as follows-

1. It will read from a text file. Each line of the file contains a number. Provide the file name as a command line argument. *Number of line in the file is not known.*

2. Create an array dynamically of a capacity (say 10) and store each number as you read from the file.

3. If you exhaust the array but yet to reach end of file dynamically re-size the array and keep on adding.

Here is pseudo code for the program.

```
void resize(int* oldArray, int* capacity)
{
    int newCapacity := *capacity * 2;
    int* newArray := dynamically allocate an array of size newCapacity;
    copy all data from oldArray a to newArray;
    delete[] oldArray;
    oldArray:= newArray;
    *capacity := newCapacity;
}
int main(int argc, char* argv[])
{
    if( argc!=2 )
        return error;
    string filename:= argv[1];
    open the file filename;
    capacity:=10
    array := dynamically allocate space of length capacity;
    numOfElement := 0;
    if(file is open)
    {
        while(line can be read)
        {
            tobeInserted := read a line from the file;
            if( numOfElement == capacity)
            {
                 resizeArray( array, &capacity)
            }
            array[numOfElement++] := tobeInserted;
        }
    }
}
```