CSCI 2270 – Data Structures

Recitation 2, Sept 2018

Objectives:

1. Pointers
2. Address-of Operator
3. Dereference Operator
4. Structs
5. Pointer to Structs
6. An analogy for pass-by-value VS pass-by-reference
7. Dynamic memory
8. Freeing memory
9. Destructors

1. Pointers

Every variable we declare in our code occupies 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

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;

       // * here denote p is pointer variable
        int* p = &a;
        cout<< a <<endl ;
        cout<< p <<endl ;

       //* is used to dereference p
      cout<< *p << endl ;
      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 putting a '*' before p we are accessing the value stored at the position addressed by p. Hence * is dereferencing the address p.

4. Structs

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

Why struct?

There are some instances wherein you need more than one variable to represent a complete object. As a student of CU, you should provide name, email, birthday, address to the college.

You could do like this:

```
std::string name;
std::string email;
int birthday;
std::string address;
```

But, the problem with this is you have 4 independent variables that is not grouped.

What is a struct?

C++ allows us to declare an aggregated(grouped) user defined data type. This user defined data type can in turn hold multiple variables of different data types. (Imagine like an array that holds multiple values of different data types)

```
struct student
{
	std::string name;
	std::string email;
	int birthday;
	std::string address;
};
```

5. Pointer to a struct

```
struct Distance
{
	int feet ;
	int inch ;
}

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

	//store the address of d in p
	ptr = &d;
	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.

6. An analogy for pass-by-value VS pass-by-reference

Tip!

Pass By Value

If I send a fax to someone to sign, he creates a local copy i.e prints it and then signs. The person makes changes to his local copy that he printed out(The signature won't be reflected in my original copy). He has to scan it and send it back, for me to see his signed document.

Pass by reference

Now consider I send my address to the person who is required to sign. The person comes to my place and then signs. In this case the person is modifying my original document.

7. Dynamic Memory

What is static memory allocation?
When we declare variables, we are preparing the variables that will be used. This way a compiler can know that the variable is an important part of the program. So, the compiler allocates spaces to all these variables during compilation of the program. **The variables allocated using static memory allocation will be stored in *STACK*.**

But wait, what if the input is not known during compilation? Say, you want to pass input in the command line arguments (During runtime). The size of the input is not known beforehand during compilation.

The need for dynamic memory!!
We suffer in terms of inefficient storage use and lack or excess of slots to enter data. This is when dynamic memory play an important role. This method allows us to create storage blocks or room for variables during run time. Now there is no wastage. **The variables allocated using static memory allocation will be stored in *HEAP.***

Tip!
Fine dining restaurant vs drive thru
Most of the times we reserve space at a fine dining restaurant by calling them up or reserving online. (The space is wasted if you do not show up even after reserving). The restaurant can also not accomodate more than its capacity. (Static memory)

Consider McDonalds drive-thru, you don't reserve space, you will somehow find a spot in the queue and just manage to grab your order. Here you are not reserving any seats beforehand, but still you can manage to get a meal. The restaurant can serve multiple customers even if they have not reserved. (Dynamic memory)

Dynamic memory allocation example.

```
int main()
{
   // Dynamic memory allocation
   int *ptr1 = new int;
   int *ptr2 = new int[10];
}
```

8. Freeing Memory

Once a programmer allocates a memory dynamically as shown in the previous section, it is the responsibility of the programmer to delete/free the memory which is created. If not deleted/freed, even though the variable/array is still not used, the memory will continue to be occupied. This causes **memory leak.**

To delete a variable **ptr1** allocated dynamically

```
delete ptr1;
```

To delete an array ptr2 allocated dynamically

```
delete [] ptr2;
```

9. Destructors

Destructor is a member function which destructs or deletes an object.

Destructor is **automatically called** when any object defined in the program goes out of scope. An object goes out of scope when:

1. A function ends.
2. The program ends.
3. A block containing local variables ends.
4. A delete operator is called.

Destructors have same name as the class preceded by tilde(~). They don't take any argument and don't return anything.

Example usage of destructors.

```
class DestructorExample
{
    private:
        char *s;
        int size;

    public:
        DestructorExample(char *); // constructor
        ~DestructorExample();      // destructor
};

DestructorExample::DestructorExample(char *c)
{
    size = strlen(c);
    s = new char[size+1];
    strcpy(s,c);
}

DestructorExample::~DestructorExample()
{
    delete []s;
}
```

10. Quiz

1. Consider an array **int a[3] = {1, 2, 3}.** What is the output for ?
    a. a+2
    b. *(a+2)
    c. *a
    d. *a[0]
2. How come we can pass an array name as an argument to a function and still be able to persist the change ?
3. What is the need of a custom destructor ?
4. 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 ?

11. Exercise

Write a C++ program that will do as follows:

1. It will read from a text file. Each line of the file contains a number. The file name will be passed to your program as a command line argument. The number of lines in the file is not known!
2. Create an array ***dynamically*** of a set starting capacity (e.g. 10) and store each number as you read from the file.
3. When you fill your array completely with numbers, ***dynamically*** allocate a new array of twice the size, copy all the values over, and delete the old array. New numbers should be stored in this new array that's twice as large. Keep doubling the array size as many times as it takes to store all the numbers.

Pseudo code

```
int *getBiggerArray(int *oldArray, int oldCapacity)
{
    int *newArray = // dynamically allocate an array of size 2 * oldCapacity

    // copy all data from oldArray to newArray
    delete [] oldArray;
    return newArray;
}

int main (int argc, char *argv[]) {
    if(/* there is not exactly one command-line argument */)
    {
      // throw error
    }

    // open the filename that was the first command-line argument
    int capacity = 10;
    int *array = // dynamically allocate array with initial capacity
    int numberOfLines = 0;
    for(/* each line in the file*/)
    {
        if(/* numberOfLines >= capacity */)
        {
            array = getBiggerArray(array, capacity);
            capacity = 2 * capacity;
        }
        // add the number in the line into the array
    }
}
```